11-2020

Enhancing developer interactions with programming screencasts through accurate code extraction

Lingfeng BAO
Zhejiang University

Shengyi PAN
Zhejiang University of Technology

Zhenchang XING
Australian National University

Xin XIA
Monash University

David LO
Singapore Management University, davidlo@smu.edu.sg

Author

Lingfeng BAO, Shengyi PAN, Zhenchang XING, Xin XIA, David LO, and Xiaohu YANG

Enhancing Developer Interactions with Programming Screencasts through Accurate Code Extraction

Lingfeng Bao
Zhejiang University
Hangzhou, Zhejiang, China
lingfengbao@zju.edu.cn

Shengyi Pan
Zhejiang University of Technology
Hangzhou, Zhejiang, China
shengyipan@outlook.com

Zhenchang Xing
Australian National University
Canberra, Australia
zhenchang.Xing@anu.edu.au

Xin Xia
Monash University
Melbourne, Victoria, Australia
Xin.Xia@monash.edu

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Xiaohu Yang
Zhejiang University
Hangzhou, Zhejiang, China
yangxh@zju.edu.cn

ABSTRACT

Programming screencasts have become a pervasive resource on the Internet, which is favoured by many developers for learning new programming skills. For developers, the source code in screencasts is valuable and important. However, the streaming nature of screencasts limits the choice that they have for interacting with the code. Many studies apply the Optical Character Recognition (OCR) technique to convert screen images into text, which can be easily searched and indexed. However, we observe that the noise in the screen images significantly affects the quality of OCRed code.

In this paper, we develop a tool named *psc2code*, which has two components, denoising code extraction from screencasts and enhancing programming video interaction. Experiment results on 1142 programming screencasts from YouTube show *psc2code* can effectively identify frames containing valid code region with a F1-score of 0.88 and improve the quality of OCRed code by fixing 46% of the errors. We also conduct a user study to evaluate the applicability of *psc2code* in enhancing video interaction, which shows it helps participants learn the knowledge in tutorials more efficiently.
Demo video: https://youtu.be/Ju1JC78NEb8
Replication Package: https://github.com/baolingfeng/PSC2CODE

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

Programming Videos, Code Extraction, Computer Vision

1 INTRODUCTION

Programming screencasts, such as programming video tutorials on YouTube, have become a popular resource for developers to learn coding. They provide authors an efficient way to introduce programming skills, and help viewers better learn the knowledge by watching a developer's coding in action (e.g., how errors are being fixed step by step) [9]. However, the streaming nature of programming screencasts, i.e., a stream of screen-captured images, limits the choice that video watchers have for interacting with its content, such as navigating the video and reusing the source code.

An intuitive way to enhance the interaction is using the Optical Character Recognition (OCR) technique to convert the video content into text, which can be easily searched and indexed. Besides, the OCRed code can be directly copied and reused in one's own program. Several approaches have been proposed to extract code from programming screencasts [2, 8, 13, 17], but none of them explicitly address the following three "noisy" challenges: 1) Programming screencasts are not all about presenting the process of code writing in IDEs. Sometimes the authors will use other software applications as well, for example, visiting API documentation in web browsers and introducing concepts in Power Point slides. There is no need to extract such non-code contents. 2) Apart from the code editor, IDEs also include many other parts (e.g., console, tool bar). Even within the code editor, there is interference from code completion suggestion window, popup menu, etc. The mix of source code and interference often leads to poor OCR results. 3) Due to the special characteristics of GUI (e.g., overlap of UI elements, code highlights) and low resolution of screen images, the text produced by the OCR technique is not 100% accurate even for a clear code region.

In this paper, we propose a tool named *psc2code* to address three "noisy" challenges and enhance developer interactions by leveraging the more accurate code extracted from programming screencasts. Figure 1 presents an overview of *pcs2code*. Given a programming screencast, *psc2code* first removes the nearly-identical frames by calculating the normalized root-mean-square error (NRMSE) in terms of pixel matrices. Then, it leverages a CNN-based classifier to further remove the non-code and noisy-code frames (e.g., frames
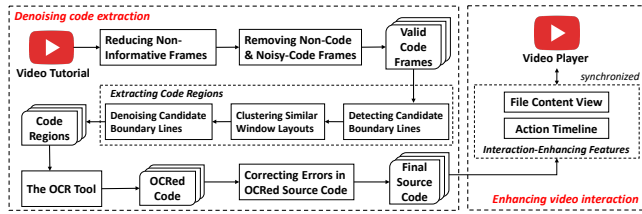
Figure 1: Framework of *psc2code*

containing pop-up windows). Next, it detects the boundary of sub-windows in valid frames and crops the regions that are likely to be the code-editor window. To reduce the noise in the detected sub-window boundaries, *psc2code* clusters the nearby boundary lines and frames with similar window layout. Finally, it uses the OCR technique to extract code from the cropped regions, and further corrects the errors. Based on the extracted code, *psc2code* enhances the navigation and exploration of programming screencasts.

2 DENOISING CODE EXTRACTION

In this section, we introduce the code extraction part of *psc2code*, which consists of following steps: 1) reducing non-informative frames; 2) removing non-code and noisy-code frames; 3) extracting code regions; 4) correcting errors in OCRed source code.

2.1 Reducing Non-Informative Frames

Most of the consecutive frames from a programming screencast are nearly the same, and these nearly-identical frames are of no use for further analysis. Hence, similar to existing techniques [1, 11, 13, 14], the first step of *psc2code* is to remove such non-informative frames.

First, *psc2code* samples the given programming screencast by extracting the first frame of each second as an image using FFmpeg[1]. The sequence of the extracted frames is denoted as $\{f_i\}(1 \le i \le N, N\ being\ the\ last\ second\ of\ the\ screencast)$. Then, to filter out non-informative frames, *psc2code* computes the normalized root-mean-square error (NRMSE) between consecutive frames at pixel level [16], which ranges from 0 (identical) to 1 (completely different). Starting with f_1, only if the dissimilarity between f_i and $f_j(j \ge i+1)$ is above an empirically determined threshold (0.05 in this work), f_j will be retained and selected as a new starting point. Otherwise, it will be removed as a non-informative frame.

2.2 Removing Non-Code & Noisy-Code Frames

In this step, *psc2code* removes frames that do not contain code (e.g., web pages with API documentation) or contain code regions with noise (e.g., code completion popups that block the real code). However, the non-code and noisy-code frames vary greatly (e.g., diverse visual features, different window properties), making it impossible to propose effective rules for removing these frames. We follow the approach of Ott et al. [12] to use a CNN-based classifier for identifying the non-code and noisy-code frames. Specifically, it can be viewed as a binary classification problem, i.e., to predict whether a frame is valid or not.

- **Valid frames**: frames with at least an entire code editor window and the contained code content is completely visible.

- **Invalid frames**: frames with no code editor window or the contained code content is partially visible.

Labeling Training Frames Fifty videos are randomly selected from our programming screencasts dataset (Section 4.1). We make sure that the selected videos cover all the playlists. After removing the non-informative frames and manually labeling, we get 3324 valid frames and 1864 invalid frames, respectively.

Building the CNN-based Classifier We use a pretrained VGG model since it has been shown to perform well in detecting the source code in programming screencasts [12]. Specifically, we only train the top layer of the VGG model for a maximum iterations of 200. The 10-fold cross validation is applied in training and testing. Besides, to ensure the input of CNN have fixed size, all frames are rescaled to 300×300 pixels before sending into the network. The validation results suggest that our CNN-based classifier can effectively identify valid frames with an overall accuracy of 97.3%.

2.3 Extracting Code Regions

In this step, *psc2code* extracts the code region from valid frames. It first divides the whole frame into several sub-windows by rectangle boundaries, and then identifies the sub-window that is most likely to be the code editor. However, the original results of boundary detection are very noisy (see Figure 2). To reduce such noise, *psc2code* clusters nearby lines and frames with similar window layouts.

Detecting Candidate Boundary Lines First, *psc2code* extracts the edge map of a valid frame by Canny edge detector [5]. Then, Probabilistic Hough transform [10] is applied to detect horizontal and vertical lines. Lines that are incline and short (less than 60 pixels) are filtered out as noise. Figure 2(b) shows the candidate boundary lines after filtering. To further reduce the detection noise, *psc2code* applies the density-based clustering algorithm DBSCAN [6] to cluster nearby lines based on their overlap and distance, and uses the longest line in each cluster to represent any other lines.

Clustering Frames with Similar Window Layouts Clustering the nearby lines effectively reduces the noise, but there is still other interference, for example, boundaries of selected line highlights and scrollbars. These noisy lines are difficult to eliminate since their styles and positions vary between different frames. However, we notice that the window layouts of most frames from a programming screencast remain the same. Therefore, by selecting the lines that are shared by the majority of frames, we can eliminate the differences between frames and get the unified window layout. Based on the above observation, *psc2code* uses DBSCAN to cluster frames according to the distribution of contained boundary lines. Each cluster of the clustering results represents a unique window layout. Only the lines shared by the majority of frames in the cluster will be selected as the sub-window boundaries for the corresponding layout. Figure 2(c) shows that by clustering nearby lines and frames with similar window layouts, the noise in the original detection results is successfully eliminated.

Detecting Code Regions To detect sub-windows, *psc2code* forms the clear boundary lines obtained by the above two steps into rectangles. If several rectangles overlap, only the smallest one will be kept. By doing so, *psc2code* only crops the main content region while ignoring the potential window decorators such as headers and rulers. The code region can be easily located based on the

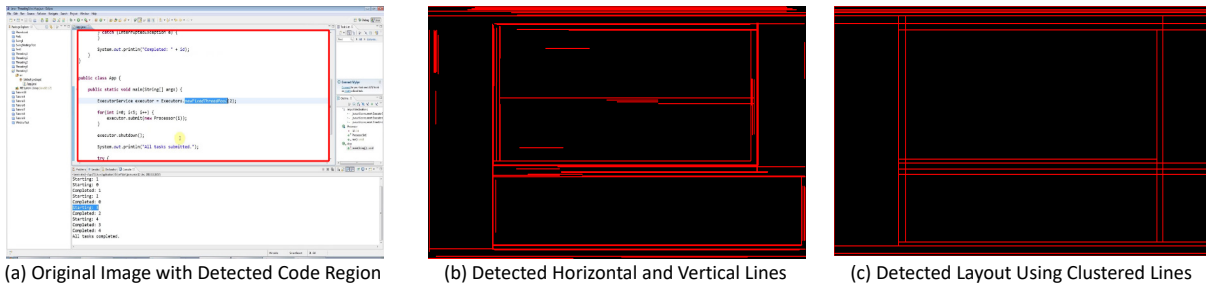| (a) Original Image with Detected Code Region	(b) Detected Horizontal and Vertical Lines	(c) Detected Layout Using Clustered Lines

Figure 2: Illustration of Sub-window Boundary Detection

observation that the code editor sub-window usually takes the largest part in the frames of programming screencasts. The detected code region is highlighted in red box in Figure 2(a).

2.4 Correcting Errors in OCRed Source Code

Given an image of the cropped code region, *psc2code* utilizes the Google Vision API [7] to extract source code. However, the OCR errors are very common, for example, symbol '=' is recognized as '-' and cursor is recognized as 'I'. To correct OCR errors, *psc2code* first removes line numbers and fixes Unicode errors based on heuristics of Kandarp and Guo [8]. Then, inspired by the work of Yadid and Yahav [17], it utilizes cross-frame information to further correct OCR errors. Specifically, *psc2code* learns a statistical language model from a large corpus of source code collected from 300 GitHub Java projects as a domain-specific spell checker. For the incorrect words and line structures detected by the spell checker, *psc2code* corrects them by finding the correct one in other related frames. For example, `Properties prop In - new Properties();` is an incorrect code line, whose line structure is denoted as IDU IDL IDU – new IDU ( ) ;, where IDU and IDL are identifiers beginning with upper and lower characters, respectively. By finding the closest correct line in other frames based on edit distance, the incorrect line can be fixed as `Properties propIn = new Properties();`, which has a correct line structure, i.e., IDU IDL = new IDU ( ) ;.

3 ENHANCING VIDEO INTERACTION

Based on the extracted code, *psc2code* enhances the navigation and exploration of programming screencasts. Different from our prior work [3], which enhances the video by collecting the author's human-computer interaction data through system level instrumentation during the recording, *psc2code* can be directly applied to the large amount of existing programming videos on the Internet since it only requires the automatically extracted code. Figure 3 shows the screenshots of our designed web application prototype, which supports the following features for interaction enhancements:
Video Analysis Given a YouTube programming tutorial, *psc2code* will automatically analyze the video once the viewers click the button (① in Figure 3(a)). The analyzing process includes three steps: 1) downloading the video using pytube[2]; 2) extracting the source code from the video (Section 2); 3) initializing the following interaction-enhancing features based on the extracted code.
File Content View This feature provides watchers a clear view of all code contents that have already appeared in the programming

tutorial till the current time. To detect different files, DBSCAN is applied to cluster frames based on the similarity between their lines of code (LOC). For two frames, the similarity is measured by computing the normalized Longest Common Sublines (similar to Longest Common Subsequence) between their LOC. The content shared by the frames in each cluster is regarded as an independent file and named after the contained Java class (② in Figure 3(b)). Moreover, the current file and its content (③ in Figure 3(b)) are synchronized with video playing. With the help of this feature, viewers can easily copy the extracted source code for reuse in their own program and clearly view any code content by simply clicking the file name instead of navigating the video to the specific frame.
Action Timeline This feature tells viewers when the tutorial author does what to which file. We compare the extracted code between the adjacent valid frames to detect the following two actions: a) *edit*: adjacent frames belong to the same file but have different code contents. b) *switch*: adjacent frames belong to two different files. A brief description of each action is presented together with its type and timestamp. For action *edit*, a summary including the number of inserted and deleted code lines will be shown. By clicking to expand the summary, viewers can get the detailed information of the adjusted code contents (④ in Figure 3(c)). For action *switch*, the name of the original and target file will be presented. By clicking the timestamp of the interested action, tutorial watchers can directly navigate the video to the corresponding frame.

4 EVALUATION

4.1 Programming Video Tutorial Dataset

In our study, we focus on Java programming. However, *psc2code* can be easily extended to other programming languages by rebuilding the corresponding spell checker (Section 2.4). We build our dataset by using YouTube APIs[3] to get videos from 50 most popular playlists for Java tutorial. By manually removing those without live coding, we collect 23 playlists with 1142 videos in total[4]. Except for 50 videos that have already been used for training the CNN-based classifier (Section 2.2), we randomly sample two videos from each playlist for evaluation. After removing the non-informative frames, there are 4828 frames from 46 videos involved in testing.

4.2 Effectiveness of Code Extraction

We evaluate the effectiveness of three main steps in code extraction (Section 2), identifying valid frames, locating code regions and

---

[2]https://github.com/nficano/pytube

[3]https://developers.google.com/youtube/v3/

[4]The whole dataset can be found: https://github.com/baolingfeng/psc2code

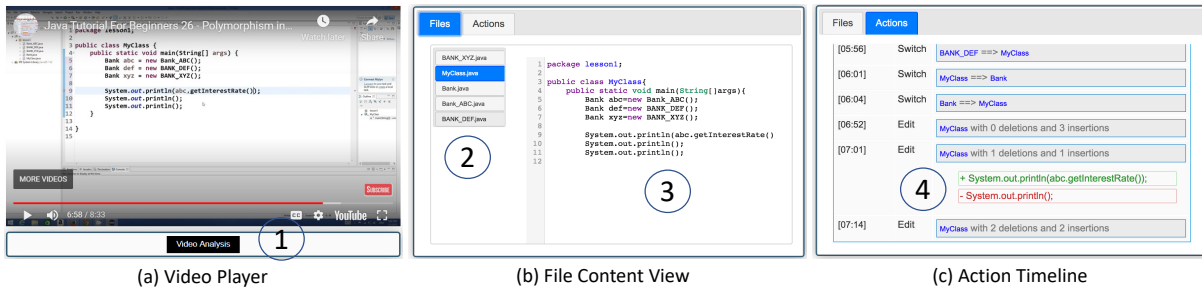(a) Video Player (b) File Content View (c) Action Timeline

Figure 3: Screenshots of *psc2code*

Table 1: Performance Comparison with the Baseline

Approach	Accuracy	F1-score @valid	F1-score @invalid	IOU
Ours	0.85	0.88	0.83	0.92
Baseline	0.73	0.74	0.72	0.64

Table 2: Average Completion Time, Answer Correctness and Usefulness Rating of Three Videos

Video	Time		Correctness		Rating	
	baseline	ours	baseline	ours	baseline	ours
1	367.0	322.8	0.92	0.96	2.4	4.6
2	532.4	300.8	0.92	0.96	2.2	4.2
3	714.6	415.2	0.80	0.92	2.2	4.2
Avg.	538.0	346.3	0.88	0.95	2.3	4.3

correcting errors in OCRed code. For the former two, we use the approach of Alahmadi et al. [1] as the baseline, where the You Only Look Once (YOLO) network [15] is applied to identify the valid frames and locate the code region simultaneously. We use the same training set in Section 2.2 to train the baseline model and apply it on the testing set for performance comparison with our approach. We adopt accuracy and F1-score for the evaluation of valid frames identification. To measure the accuracy of predicted code region bounding box, we use the Intersection over Union (IOU) metric [18], which is also adopted in Alahmadi et al.'s work. The comparison results presented in Table 1 suggest that *psc2code* greatly surpasses the baseline. Although YOLO is powerful in general object detection, our approach is more suitable for this particular task.

To evaluate the ability of *psc2code* in correcting OCR errors, we calculate the ratio of incorrect words that are corrected by *psc2code* for each frame in the testing set. However, since our approach may choose wrong candidate words in fixing errors, we further manually check whether a wrong word is truly corrected or not. The testing results show that 88% of the corrected errors and 46% of all the incorrect words in the original OCRed code can be truly corrected. For errors *psc2code* dose not attempt to correct, most of them are partial words that are still being typed.

4.3 User Study

We conduct a user study to evaluate the applicability of *psc2code* in enhancing developer interactions with programming screencasts

(Section 3). Three videos with different topics are chosen from our dataset. Then, we design five questions for each video with the goal of covering the workflow and different kinds of information (e.g., code content, API usage). Answering these questions requires the participants to carefully watch and explore the video content. Ten undergraduate students from the College of Computer Science of Zhejiang University are recruited as participants, and none of them are familiar with the programming tasks used in the study. Following the between-subject design, we randomly divide ten participants into two groups for each video. Participants in the experimental group use our prototype tool, while those in the control group use a regular video player. Participants in both groups are asked to answer the same questions and rate the usefulness of the corresponding tool. The ratings are on a 5-points liker scale with 5 being the best. We evaluate the helpfulness of our tool by comparing the completion time, answer correctness and usefulness rating from two groups of participants. As shown in Table 2, participants generally agree that our tool is more helpful for learning knowledge from video tutorials compared to the baseline (i.e., regular video player), which is also confirmed by the fact that participants in the experimental group do give more correct answers in less time.

5 CONCLUSION

We propose a tool named *psc2code* to enhance developer interactions by leveraging the accurate code extracted from programming screencasts. For the effectiveness of denoising code extraction, our experimental results indicate that *psc2code* can effectively locate the code content and correct errors in OCRed code. For the applicability of enhancing programming video interaction, our user study confirms that *psc2code* can help viewers learn the programming tutorials more efficiently compared to the regular video player. In the future, we will collect more tutorials with different programming languages to evaluate and strengthen the robustness of *psc2code*.

ACKNOWLEDGMENTS

REFERENCES

[1] Mohammad Alahmadi, Jonathan Hassel, Biswas Parajuli, Sonia Haiduc, and Piyush Kumar. 2018. Accurately Predicting the Location of Code Fragments in Programming Video Tutorials Using Deep Learning. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2–11.

[2] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. 2015. Reverse engineering time-series interaction data from screen-captured videos. In *Proc. SANER*. IEEE, 399–408.

[3] Lingfeng Bao, Zhenchang Xing, Xin Xia, and David Lo. 2018. VT-Revolution: Interactive Programming Video Tutorial Authoring and Watching System. *TSE* (2018), 1–1.

[4] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Minghui Wu, and Xiaohu Yang. 2020. psc2code: Denoising Code Extraction from Programming Screencasts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–38.

[5] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698.

[6] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[7] GoogleVision 2018. Google Vision API. https://cloud.google.com/vision/.

[8] Kandarp Khandwala and Philip J Guo. 2018. Codemotion: expanding the design space of learner interactions with computer programming tutorial videos. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. ACM, 57.

[9] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, camera, action: how software developers document and share program knowledge using YouTube. In *Proc. ICPC*. IEEE Press, 104–114.

[10] Jiri Matas, Charles Galambos, and Josef Kittler. 2000. Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding* 78, 1 (2000), 119–137.

[11] Parisa Moslehi, Bram Adams, and Juergen Rilling. 2018. Feature location using crowd-based screencasts. In *Proc. MSR*. ACM, 192–202.

[12] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *Proc. MSR*. 376–386.

[13] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too long; didn't watch!: extracting relevant fragments from software development video tutorials. In *Proc. ICSE*. ACM, 261–272.

[14] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, Sonia Cristina Haiduc, Barbara Russo, and Michele Lanza. 2017. Automatic identification and classification of software development video tutorial fragments. *IEEE Transactions on Software Engineering* (2017).

[15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proc. CVPR*. 779–788.

[16] Da-Chun Wu and Wen-Hsiang Tsai. 2000. Spatial-domain image hiding using image differencing. *IEE Proceedings-Vision, Image and Signal Processing* 147, 1 (2000), 29–37.

[17] Shir Yadid and Eran Yahav. 2016. Extracting code from programming tutorial videos. In *Proc. Onward! 2016*. ACM, 98–111.

[18] C Lawrence Zitnick and Piotr Dollár. 2014. Edge boxes: Locating object proposals from edges. In *ECCV*. Springer, 391–405.