CodeSCAN: ScreenCast ANalysis for Video Programming Tutorials

Alexander Naumann[1,2], Felix Hertlein[1,2], Jacqueline Höllig[1,2], Lucas Cazzonelli[1,2] and Steffen Thoma[1,2]

[1] *FZI Research Center for Information Technology, Karlsruhe, Germany*

[2] *Karlsruhe Institute of Technology, Karlsruhe, Germany*

Keywords: Computer Vision, Optical Character Recognition, Object Detection, Image Binarization, Datasets.

Abstract: Programming tutorials in the form of coding screencasts play a crucial role in programming education, serving both novices and experienced developers. However, the video format of these tutorials presents a challenge due to the difficulty of searching for and within videos. Addressing the absence of large-scale and diverse datasets for screencast analysis, we introduce the CodeSCAN dataset. It comprises 12,000 screenshots captured from the Visual Studio Code environment during development, featuring 24 programming languages, 25 fonts, and over 90 distinct themes, in addition to diverse layout changes and realistic user interactions. Moreover, we conduct detailed quantitative and qualitative evaluations to benchmark the performance of Integrated Development Environment (IDE) element detection, color-to-black-and-white conversion, and Optical Character Recognition (OCR). We hope that our contributions facilitate more research in coding screencast analysis, and we make the source code for creating the dataset and the benchmark publicly available at a-nau.github.io/codescan.

1 INTRODUCTION

In the ever-evolving landscape of programming education and knowledge dissemination, coding screencasts on platforms such as YouTube have emerged as powerful tools for both novice and experienced developers. These video tutorials not only provide a visual walkthrough of coding processes but also offer a unique opportunity for learners to witness real-time problem-solving, coding techniques, and best practices. As the popularity of coding screencasts continues to soar, the possibility to augment traditional video content with additional information to improve the learner's experience, becomes increasingly interesting and relevant. The source code extracted from a screencast, that replicates the full coding project up to the given position in the video, is one such element that can empower learners with the ability to delve deeper into the presented material.

Incorporating such code extraction into the realm of coding screencasts enables learners to benefit from a dual advantage. Firstly, it empowers video search algorithms to utilize the full source code, enhancing the precision and relevance of search results. By indexing and analyzing the extracted code, search engines can pinpoint specific programming concepts,

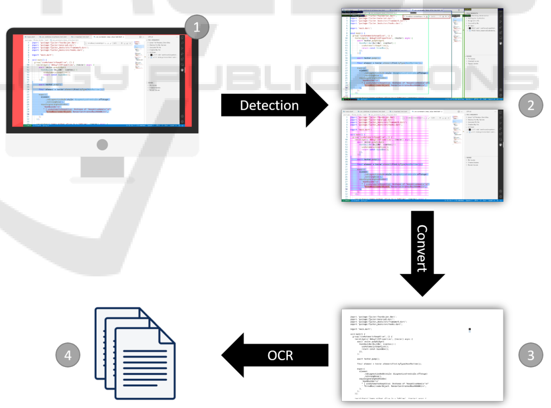


Figure 1: Source code extraction pipeline: Given an image from a coding screencast (1), the IDE elements need to be detected (2). This is followed by an optional binarization step (3) which converts the color image into a black-and-white image, and finally OCR is applied (4) to read out the source code.

language features, or problem-solving techniques, leading learners to the most pertinent videos that align with their educational needs. Rather than watching an entire video in search of a specific code segment, learners can navigate directly to the relevant sections, streamlining the learning process and increasing overall comprehension. Secondly, having access to the

Naumann, A., Hertlein, F., Höllig, J., Cazzonelli, L. and Thoma, S.
CodeSCAN: ScreenCast ANalysis for Video Programming Tutorials.
DOI: 10.5220/0013093100003912
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2025) - Volume 2: VISAPP*, pages 269-277
ISBN: 978-989-758-728-3; ISSN: 2184-4321
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

full code at any given point in a coding screencast also empowers learners to engage in hands-on experimentation seamlessly. By providing learners with the entire codebase in its current state, irrespective of their position in the video timeline, the educational experience is transformed into an interactive playground for exploration and experimentation, where the code can be executed and experimented with at any stage within the video.

However, the task of retrieving the full source code of a project at any given point in a coding screencast is challenging: Instructors frequently navigate between different files within a project, introduce small modifications or copy-paste large amounts of code and additionally, the character distribution of source code differs strongly from standard text paragraphs. To cope with this it is crucial to understand the visible IDE components in addition to being able to extract the visible source code as text. To this end, we present a novel large-scale and high-quality dataset for coding screencast analysis called CodeSCAN, in this work. It contains more than 12,000 screenshots of coding projects in 24 different programming languages. These screenshots are independent and cannot be composed into coherent videos. While full programming tutorials would enable better end-to-end testing of the pipeline, it is important to note, that all current approaches only use single frames as input. CodeSCAN is the first dataset of its kind that sets itself apart by its unprecedented size, diversity and annotation granularity. We use Visual Studio Code with approximately 100 different themes and 25 different fonts. Through automated interaction with Visual Studio Code, we achieve large visual diversity by e.g. editing code, highlighting areas, performing search, and much more. In addition to the dataset, we present a detailed literature review and analyze the performance for text recognition on integrated development environment (IDE) images in detail. More specifically, we analyze the influence of different OCR engines, image binarization and image quality. An overview of a potential source code extraction pipeline is outlined in Fig. 1.

To summarize, the main contributions of our work are

- we introduce CodeSCAN, a novel high-quality dataset for screencast analysis of video programming tutorials with 12,000 screenshots containing a multiplicity of annotations and high diversity regarding programming languages and visual appearance,
- we evaluate a baseline CNN on diverse IDE element detection and propose the usage of a so-called *coding grid* for source code localization,
- we benchmark five different OCR engines, and analyze the influence of image binarization and image quality on performance.

The remainder of this work is organized as follows. We present an overview of related work in Sec. 2. Subsequently, we present the details of our dataset generation approach in Sec. 3. Sec. 4 presents the evaluation for IDE element detection and OCR. Finally, Sec. 5 concludes the paper.

2 RELATED WORK

We review the existing literature on approaches for code extraction, tools using code extraction, and, finally, available datasets.

2.1 Approaches

A common pipeline to extract the source code from a programming screencast as text looks as follows: (1) the video is split up into images and duplicates are removed, (2) the image is classified as containing or not containing source code, (3) the source code is located by predicting a bounding box, and finally (4) OCR is applied within this bounding box to extract the source code as text. This full pipeline is necessary to be able to perform inference on new, unseen video programming tutorials. However, the literature frequently focuses only on a subset of these tasks. Ott et al. (2018a) trained a CNN classifier to identify whether a screenshot contains fully visible code, partially visible code, handwritten code or no code. Ott et al. (2018b) use a CNN to locate the code and to classify its programming language (Java or Python) purely by using image features. Alahmadi et al. (2020b) tackle code localization to remove the noise in OCR results which is introduced by additional IDE elements such as the menu by comparing five different backbone architectures. Malkadi et al. (2020) focus on comparing different OCR engines, thus, focusing on step (3). The literature analyzes between one and seven different programming languages (if these are specified). *Java* is most popular (Alahmadi et al., 2018, 2020b; Bao et al., 2019, 2020b, 2020a; Bergh et al., 2020; Ott et al., 2018a, 2018b; Ponzanelli et al., 2016b, 2016a, 2019; Yadid & Yahav, 2016; Zhao et al., 2019), followed by *Python* (Alahmadi et al., 2020b; Bergh et al., 2020; Malkadi et al., 2020; Ott et al., 2018b; Zhao et al., 2019), *C#* (Alahmadi et al., 2020b; Malkadi et al., 2020) and *XML* (Alahmadi, 2023). Also, different IDEs, such as Eclipse (Bao et al., 2015, 2017, 2019), Visual Studio (Malkadi et al., 2020) or a diverse mix of multiple IDEs (Alahmadi et al., 2018, 2020b, 2020a; Alahmadi, 2023; Malkadi et al., 2020; Ott et al., 2018a, 2018b; Zhao et al., 2019) are used. Other related tasks include recognizing user actions (e.g. enter or select text) in coding screencasts (Zhao et al., 2019) and identifying UI screens in videos (Alahmadi et al., 2020a).

2.2 Tools

Ponzanelli et al. (2016b) present CodeTube, a recommender system that leverages textual, visual and audio information from coding screencasts. The users inputs a query, and CodeTube intends to return a relevant, cohesive and self-contained video. Bao et al. (2020b) introduce ps2code, a tool which leverages code extraction to provide a coding screencast search engine and a screencast watching tool which enables interaction. PSFinder (Yang et al., 2022) classifies screenshots into containing or not containing an IDE to identify live-coding screencasts.

2.3 Datasets

To the best of our knowledge, only two of the previously mentioned works have their full annotated datasets publicly available. The dataset presented by Malkadi et al. (2020) comprises screenshots of only the visible code with the associated source code files in Java, Python and C#. The dataset was created manually, and no pixel-wise correspondence between source code and images is available. Furthermore, there is low diversity in the IDE color scheme and font. Alahmadi et al. (2020a) present a dataset where UI screenshots are annotated with bounding boxes. Since they do not target code extraction, no such annotations are provided.

2.4 Discussion

To empower learners to engage in hands-on experimentation seamlessly and to improve the search quality within and across videos, it is crucial to be able to extract the source code of a complete project. This requires, for example, to identify the file tree and the currently active tab. Moreover, it is important to reliably and incrementally edit existing files. In order to enable such sophisticated full-project source code extraction, it is essential to have a large, high-quality dataset with detailed annotations. Since currently only one relevant annotated dataset with insufficient annotation granularity is publicly available (Malkadi et al., 2020), we present the novel dataset CodeSCAN. CodeSCAN covers 24 different programming languages, over 90 Visual Studio Code themes and 25 different fonts. We provide a multiplicity of automatically annotated pixel-accurate annotations that reach far beyond code localization. Furthermore, we are the first to benchmark diverse IDE element localization and to analyze the influence of image binarization and image quality on OCR.

3 DATASET GENERATION

Our dataset was acquired by scraping data from https://github.dev. The details on the dataset acquisition will be presented in Sec. 3.1. Subsequently, we lay out our annotation generation approach in Sec. 3.2.

3.1 Data Acquisition

For the data acquisition, we exploit the fact that any Github repository can be opened with a browser version of Visual Studio Code by changing the URL from *https://github.com/USERNAME/REPONAME* to *https://github.**dev**/USERNAME/REPONAME*. Thus, we randomly select 100 repositories with a permissive license (MIT, BSD-3 or WTFPL) per programming language. Since we consider 24 programming languages, this results in a total of $2,400$ repositories. The considered programming languages are *C*, *C#*, *C++*, *CoffeeScript*, *CSS*, *Dart*, *Elixir*, *Go*, *Groovy*, *HTML*, *Java*, *JavaScript*, *Kotlin*, *Objective-C*, *Perl*, *PHP*, *PowerShell*, *Python*, *Ruby*, *Rust*, *Scala*, *Shell*, *Swift,* and *TypeScript*. We select five files with file endings corresponding uniquely to the underlying language (e.g. *.py* for *Python*) at random per repository, leading to a total of $12,000$ files. To achieve highly diverse IDE appearances, we use more than 90 different Visual Studio Code themes and 25 monospaced fonts. Moreover we apply the following fully automated, random layout changes

- different window sizes (for desktop 1920×1200, 1920×1080, 1200×1920, 1440×900 and tablet 768×1024, 1280×800, 800×1280),
- layout type (classic or centered layout),
- coding window width,
- degree of zoom for the source code text,
- output panel visibility, size and type (e.g. Terminal, Debugger, etc.),
- sidebar visibility, location, size and type (toggling sidebar options),
- menubar visibility (classic, compact, hidden, visible),

- activity bar visibility,
- status bar visibility,
- breadcrumbs visibility,
- minimap visibility, and
- control characters and white spaces rendering visibility

and realistic user interactions

- highlighting multiple lines of code,
- right clicking at random positions in the code,
- adding or removing characters from the source code,
- search within the opened file,
- search within the project, and
- open a random number of other tabs.

The automation of all these appearance changes was implemented using the framework Selenium[1]. We refer to one such final IDE configuration (i.e. the full visual appearance of the IDE with the current active source code file and its appearance characteristics) as *scene* in the following. To persist the current scene, we export the scene webpage using SingleFile[2]. In addition, we save the full source code of the currently visible file as reference. Thus, at the end of the dataset acquisition phase, we have a *source.txt* and *page.html* file for each of the $12,000$ code files.

Note, that especially selecting random IDE themes is challenging to automate[3] and thus, not always the desired theme was applied correctly for each scene. Since each scene is represented in a single HTML file, however, we are able to leverage the CSS specifications (background and font color) to perform theme-wise clustering after the data acquisition for quality assurance.

3.2 Annotation Generation

We utilize the HTML and source code file to automatically generate all annotations. The bounding box of any IDE element can be retrieved by manually identifying the CSS selector for the relevant element inside the HTML and accessing its bounding box information. Since the naming is consistent across all HTML files, this process is necessary only once per element. We annotate numerous IDE elements (e.g. coding area, active tab, status bar etc.) and refer to Tab. 1 for a full list and to Fig. 2b for a visualization. Note, that arbitrary additional annotations for other IDE elements can be added any time to our dataset. By taking an automated screenshot of the scene using the original resolution, we get the underlying image to which the annotations can be applied directly without any offsets.

Moreover, we extract annotations for text detection and recognition for the source code of the active file visible in the coding area. This is done on a per-line and per-word basis. Additionally, we provide annotations for the *coding grid*, i.e. the tight bounding box around the visible code including the character height and width. These annotations of only six numbers (four for the bounding box and character width and height) enables the computation of the coding grid as visualized in Fig. 2c. The coding grid can be used to retrieve line and character-based text bounding boxes and brings the additional benefit, that it is much easier to accurately retrieve indentation, which is crucial for languages such as Python.

Since we additionally analyze the influence of binarization, i.e. conversion to a clean black-and-white version of the screenshot, we automatically generate annotations for this use case. This is done by creating a binarized HTML version of the scene by identifying the relevant CSS selectors for source code text. We enforce black color for all source code text and white for everything else by adjusting the CSS parameters. See Fig. 2d for an example.

The dataset is split into a training, validation and test set of sizes $7,561$, $1,921$, and $2,518$, respectively. There is no overlap of fonts across the splits, and only a minor overlap across themes, which was inevitable due to the previously mentioned issues in automating the theme selection process.

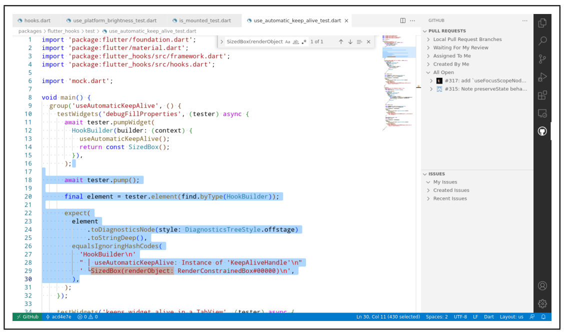
4 EVALUATION

Coding screen cast analysis comprises several parts. For our evaluations, we assume videos are processed frame-by-frame and thus, consider only single frames as input for the respective components. We evaluate IDE element detection in Sec. 4.1. Next, we evaluate image binarization in Sec. 4.2, which can help to improve text recognition performance as analyzed in Sec. 4.3. In addition to the influence of binarization, we also compare different OCR engines and investigate how they are affected by changes in the input image quality.

[1]See https://www.selenium.dev/documentation/.

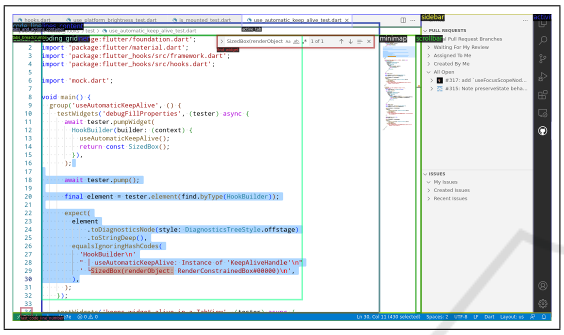
[2] https://github.com/gildas-lormeau/SingleFile.

[3]This is due to strongly varying loading times of the theme and since themes frequently have multiple available color schemes.

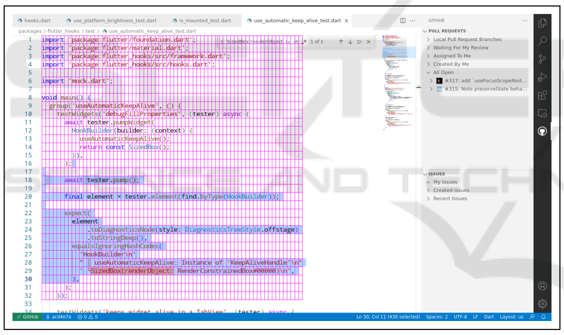


```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:flutter_hooks/src/framework.dart';
import 'package:flutter_hooks/src/hooks.dart';

import 'mock.dart';

void main() {
  group('useAutomaticKeepAlive', () {
    testWidgets('debugFillProperties', (tester) async {
      await tester.pumpWidget(
        HookBuilder(builder: (context) {
          useAutomaticKeepAlive();
          return const SizedBox();
        }),
      );

      await tester.pump();

      final element = tester.element(find.byType(HookBuilder));

      expect(
        element
            .toDiagnosticsNode(style: DiagnosticsTreeStyle.offstage)
            .toStringDeep(),
        equalsIgnoringHashCodes(
          'HookBuilder\n'
          " │ useAutomaticKeepAlive: Instance of 'KeepAliveHandle'\n"
          ' └SizedBox(renderObject: RenderConstrainedBox#00000)\n',
        ),
      );
    });

    testWidgets('keeps widget alive in a TabView', (tester) async {
```

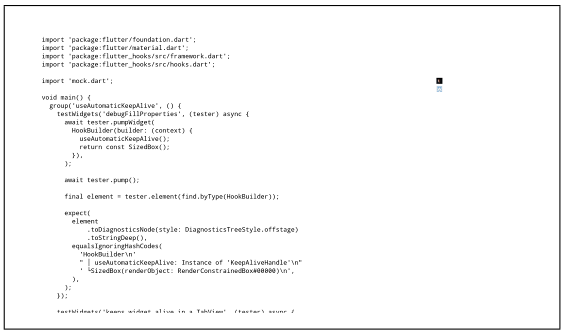
(a) Color Image of Scene



(b) Bounding Box Annotations



(c) Coding Grid Annotation



```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:flutter_hooks/src/framework.dart';
import 'package:flutter_hooks/src/hooks.dart';

import 'mock.dart';

void main() {
  group('useAutomaticKeepAlive', () {
    testWidgets('debugFillProperties', (tester) async {
      await tester.pumpWidget(
        HookBuilder(builder: (context) {
          useAutomaticKeepAlive();
          return const SizedBox();
        }),
      );

      await tester.pump();

      final element = tester.element(find.byType(HookBuilder));

      expect(
        element
            .toDiagnosticsNode(style: DiagnosticsTreeStyle.offstage)
            .toStringDeep(),
        equalsIgnoringHashCodes(
          'HookBuilder\n'
          " │ useAutomaticKeepAlive: Instance of 'KeepAliveHandle'\n"
          ' └SizedBox(renderObject: RenderConstrainedBox#00000)\n',
        ),
      );
    });

    testWidgets('keeps widget alive in a TabView', (tester) async {
```

(d) Black-and-white Image of Scene

Figure 2: Example of the different available annotation and data types showing the Github repository rrousselGit/flutter_hooks with file use_automatic_keep_alive_test.dart. Note that we omit line, word and character annotations in (b) for better readability.

4.1 Object Detection

Since the focus of this work is not to improve object detection algorithms, we benchmark our dataset on a well-established baseline. We use a Mask R-CNN (He et al., 2017) with a ResNet-50-FPN (He et al., 2016; Lin et al., 2017) backbone that was pre-trained on MS COCO (Lin et al., 2014) for our experiments. We freeze the weights at stage four and use stochastic gradient descent with momentum, a batch size of 16 and a cosine learning rate schedule (Loshchilov & Hutter, 2017). For the learning rate schedule, we set the initial learning rate to 0.001, the final learning rate to 0 after 10,000 and use a linear warm-up during the first 1,000 iterations. The results are summarized in Tab. 1. Some elements, such as *sidebar*, *tabs and actions container*, *minimap* and *output panel* achieve a Box AP above 80. Other elements, such as *title bar*, *notifications*, *last code line number*, *suggestion widget*, *tabs breadcrumbs* are harder to detect and the Box AP lies below 50. The bounding box of the *coding grid* and the ones for source code text *lines* are most important for text detection and recognition, and achieve a Box AP of 69.9 and 71.9, respectively.

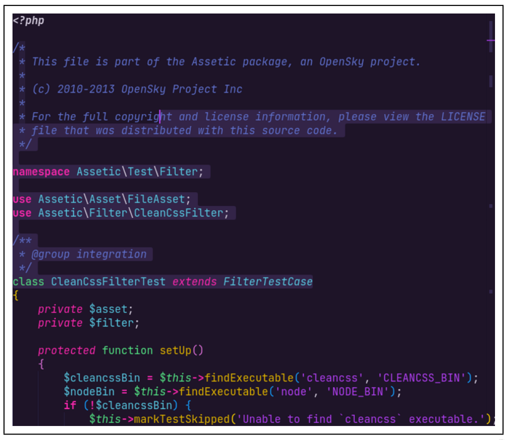
Table 1: Quantitative object detection performance per class using Box AP.

	Bounding Box		
Class name	AP	AP50	AP75
sidebar	92.0	99.8	98.7
minimap	84.6	99.0	97.6
tabs_and_actions_container	82.9	98.2	96.4
output_panel	82.4	97.6	93.9
status_bar	79.5	99.0	95.4
activity_bar	76.8	95.6	82.0
editor_container	72.5	92.4	84.7
line	71.9	93.3	87.4
code_container	71.9	90.9	82.9
coding_grid	69.9	96.7	78.6
code_line_number_bar	62.1	88.3	66.9
scrollbar	56.3	80.0	70.1
active_tab	53.1	70.0	60.0
find_widget	53.1	69.6	63.8
last_code_line_number	48.7	80.7	56.2
notifications	48.4	89.2	40.6
suggestion_widget	48.2	68.9	57.4
tabs_breadcrumbs	45.4	71.7	53.0
title_bar	31.6	44.5	40.2

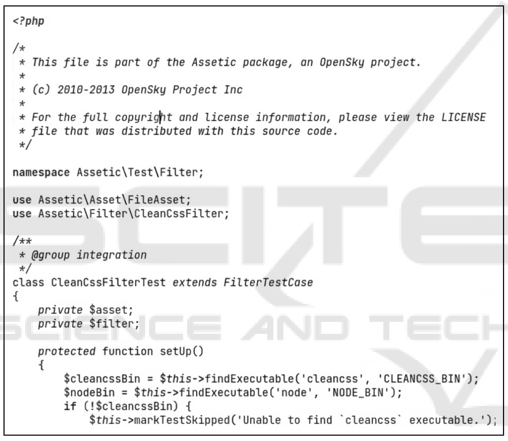
4.2 Binarization

We investigate the performance of image binarization approaches for converting a screenshot from a coding screencast to a binarized black-and-white version. For our experiments, we train Pix2PixHD (Wang et al., 2018), the successor of the very popular Pix2Pix (Isola et al., 2017) architecture for image-to-image translation. We train for 40 epochs and leave the remaining original training configuration unchanged. The results are summarized in Tab. 2 and we provide qualitative samples in Figs. 3 and 4. Overall, binarization works well as indicated by the mean Peak

Signal-to-Noise Ratio (PSNR) of 5.49 in Tab. 2 and the qualitative example in Fig. 3. The model seems to have difficulties especially with rare strong contrasts and very low contrast as indicated by the examples in Fig. 4.



```
<?php

/*
 * This file is part of the Assetic package, an OpenSky project.
 *
 * (c) 2010-2013 OpenSky Project Inc
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Assetic\Test\Filter;

use Assetic\Asset\FileAsset;
use Assetic\Filter\CleanCssFilter;

/**
 * @group integration
 */
class CleanCssFilterTest extends FilterTestCase
{
    private $asset;
    private $filter;

    protected function setUp()
    {
        $cleancssBin = $this->findExecutable('cleancss', 'CLEANCSS_BIN');
        $nodeBin = $this->findExecutable('node', 'NODE_BIN');
        if (!$cleancssBin) {
            $this->markTestSkipped('Unable to find `cleancss` executable.');
```



```
<?php

/*
 * This file is part of the Assetic package, an OpenSky project.
 *
 * (c) 2010-2013 OpenSky Project Inc
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Assetic\Test\Filter;

use Assetic\Asset\FileAsset;
use Assetic\Filter\CleanCssFilter;

/**
 * @group integration
 */
class CleanCssFilterTest extends FilterTestCase
{
    private $asset;
    private $filter;

    protected function setUp()
    {
        $cleancssBin = $this->findExecutable('cleancss', 'CLEANCSS_BIN');
        $nodeBin = $this->findExecutable('node', 'NODE_BIN');
        if (!$cleancssBin) {
            $this->markTestSkipped('Unable to find `cleancss` executable.');
```

Figure 3: Qualitative examples of the binarization showing the input image on the top and the predicted binarization on the bottom.

Table 2: Quantitative binarization performance.

	Accuracy ↑	PSNR ↑	DRDM ↓
Median	72.07	5.54	67.52
25% Quant.	67.09	4.82	62.74
75% Quant.	76.01	6.20	73.46
Mean	70.81	5.49	165.70
Std.	7.77	1.08	1699.39

4.3 Optical Character Recognition

We benchmark five different approaches for text recognition in the following. For the evaluation, we create a subset of our test split by randomly sampling 10 words from each of the 2,518 test images to reduce the computational workload while maintaining statistical significance. To eliminate the influence of text detection performance, we use the ground truth bounding boxes. We investigate the influence of binarization (color image vs. ground truth binarization vs. predicted binarization) and image quality (20%-100% of original image size as input) on text recognition performance. The state-of-the-art text recognizers we benchmark are:

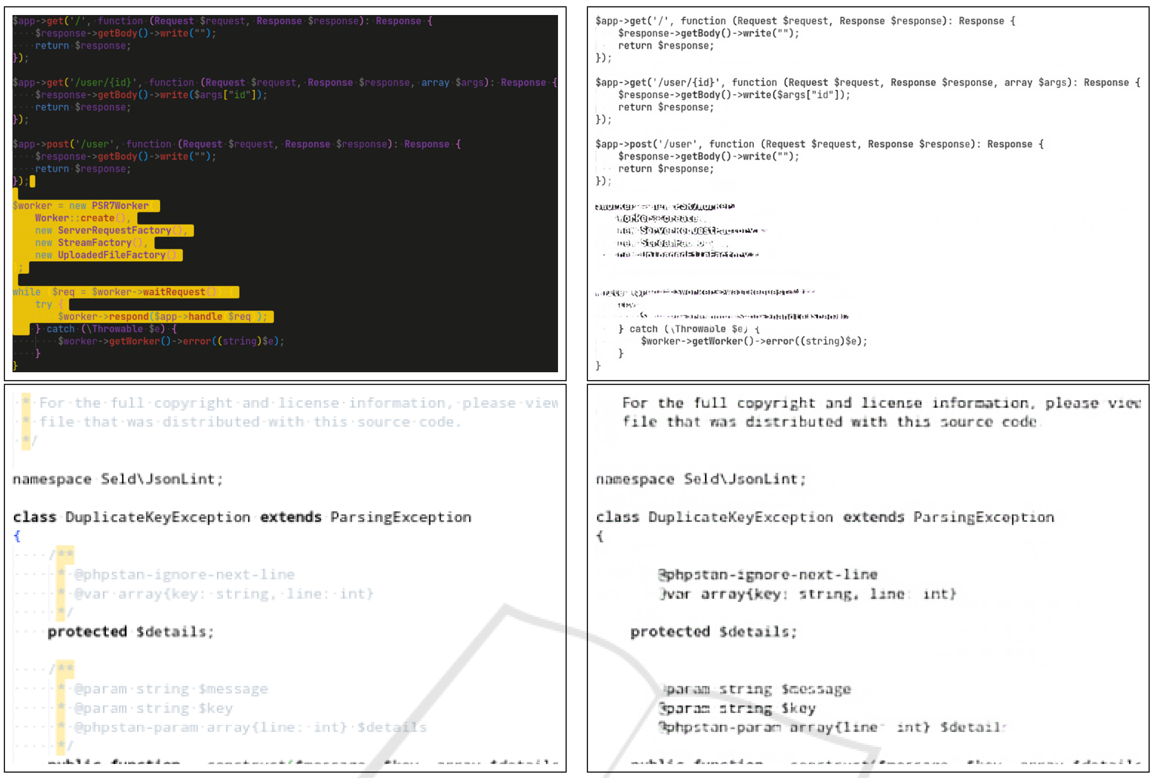
- Tesseract (Smith, 2007) (common baseline)
- SAR (Li et al., 2019) pre-trained mainly on MJSynth (Jaderberg et al., 2014, 2016), SynthText (Gupta et al., 2016) and SynthAdd (Li et al., 2019)
- MASTER (Lu et al., 2021) pre-trained on MJSynth (Jaderberg et al., 2014, 2016), SynthText (Gupta et al., 2016) and SynthAdd (Li et al., 2019)
- ABINet (Fang et al., 2021) pre-trained on MJSynth (Jaderberg et al., 2014, 2016) and SynthText (Gupta et al., 2016)
- TrOCR (Li et al., 2023) fine-tuned on the SROIE dataset (Huang et al., 2019)[4]

Note, that TrOCR does not distinguish lowercase and capital letters. Thus, we convert the ground truth to lowercase for computing the mean Character Error Rate (CER) in only this case. This obviously simplifies the task, which is why these results cannot be fairly compared with the other approaches.

On the full image resolution, we can observe from Tab. 3 that TrOCR and Tesseract benefit most from having access to the ground truth or predicted binarized version of the input image. Overall best performance is 0.08 mean CER for TrOCR and the ground truth binarized input images. The performance of SAR, MASTER and ABINet is more robust on color images which means those approaches do not benefit from a prior conversion to a black-and-white image using Pix2PixHD. We argue that these differences stem from the differences in training data. In scenarios where mostly binarized training data is available, using image-to-image translation approaches is a valid and helpful way to boost performance on the final task.

All approaches show robust performance when the image quality is degraded slightly, i.e. when it is reduced to 60%-70% of the original image size. This effect is constant across image types as seen in Fig. 5.

[4]See https://rrc.cvc.uab.es/?ch=13



```
$app->get('/', function (Request $request, Response $response): Response {
    $response->getBody()->write("");
    return $response;
});

$app->get('/user/{id}', function (Request $request, Response $response, array $args): Response {
    $response->getBody()->write($args["id"]);
    return $response;
});

$app->post('/user', function (Request $request, Response $response): Response {
    $response->getBody()->write("");
    return $response;
});

$worker = new PSR7Worker(
    Worker::create(),
    new ServerRequestFactory(),
    new StreamFactory(),
    new UploadedFileFactory()
);

while ($req = $worker->waitRequest()) {
    try {
        $worker->respond($app->handle($req));
    } catch (\Throwable $e) {
        $worker->getWorker()->error((string)$e);
    }
}
```

```
 * For the full copyright and license information, please view
 * file that was distributed with this source code.
 */

namespace Seld\JsonLint;

class DuplicateKeyException extends ParsingException
{
    /**
     * @phpstan-ignore-next-line
     * @var array{key: string, line: int}
     */
    protected $details;

    /**
     * @param string $message
     * @param string $key
     * @phpstan-param array{line: int} $details
     */
```

```
$app->get('/', function (Request $request, Response $response): Response {
    $response->getBody()->write("");
    return $response;
});

$app->get('/user/{id}', function (Request $request, Response $response, array $args): Response {
    $response->getBody()->write($args["id"]);
    return $response;
});

$app->post('/user', function (Request $request, Response $response): Response {
    $response->getBody()->write("");
    return $response;
});

[illegible]

    } catch (\Throwable $e) {
        $worker->getWorker()->error((string)$e);
    }
}
```

```
 For the full copyright and license information, please view
 file that was distributed with this source code

namespace Seld\JsonLint;

class DuplicateKeyException extends ParsingException
{
        @phpstan-ignore-next-line
        @var array{key: string, line: int}
    protected $details;

        @param string $message
        @param string $key
        @phpstan-param array{line: int} $detail
```

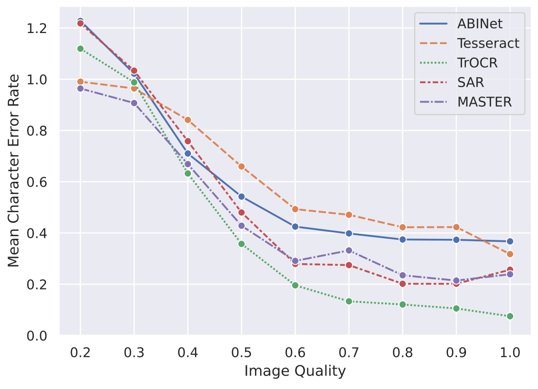
Figure 4: Challenging qualitative examples of the binarization showing the input image on the left and the predicted binarization on the right.

Table 3: Quantitative results at full image resolution using the binarized, converted (from Sec. 4.2) and color image. We report the *mean (standard deviation)* for the Character Error Rate (CER). (* CER is computed ignoring capitalization).

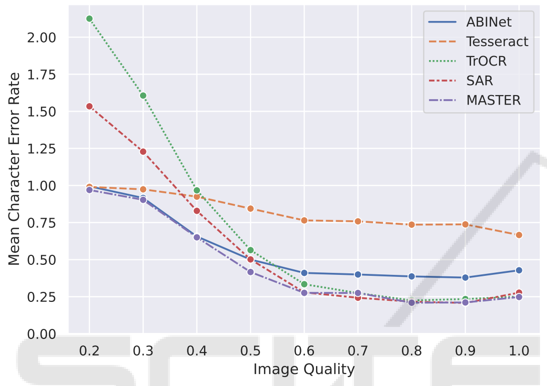
CER ↓	Binarized (GT)	Binarized (Pred.)	Color
ABINet (Fang et al., 2021)	0.37 (0.59)	0.43 (0.78)	0.43 (0.94)
Tesseract (Smith, 2007)	0.32 (0.45)	0.41 (0.46)	0.67 (0.51)
SAR (Li et al., 2019)	0.26 (0.73)	0.35 (0.97)	0.28 (0.86)
MASTER (Lu et al., 2021)	**0.24 (0.95)**	**0.35 (0.97)**	**0.25 (0.99)**
TrOCR* (Li et al., 2023)	*0.08 (0.27)*	*0.14 (0.33)*	**0.25 (0.93)**

5 CONCLUSION

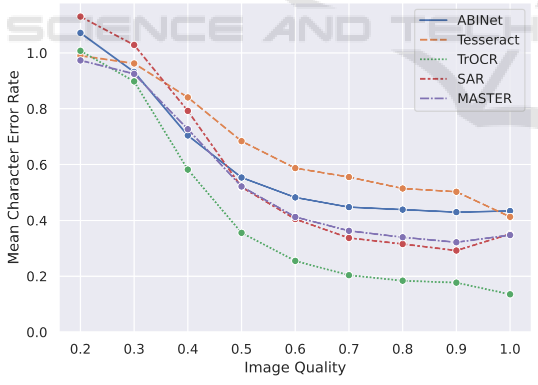
Coding screencasts have emerged as powerful tools for programming education of novices as well as experienced developers. In addition to the video content, having access to the full source code of the project at any given time of the programming tutorial brings two major benefits: (1) platform-wide and tutorial-based search can leverage the full source code to enhance the precision and relevance of search results and (2) it empowers learners to engage in hands-on experimentation with the source code seamlessly. In this work, we work towards enabling detailed information retrieval from such coding screencasts. We present a novel high-quality dataset called CodeSCAN, comprising 12,000 fully annotated IDE screenshots. CodeSCAN is highly diverse and contains 24 programming languages, over 90 different themes of Visual Studio Code, and 25 fonts while at the same time varying the IDE appearance through changing the visibility, position and size of different IDE elements (e.g. sidebar, output panel). Our evaluations show that baseline object detectors are suitable for text recognition and achieve a Box AP for source code line detection of 71.9. Moreover, we showed that baseline image-to-image translation architectures are well suited for coding screencast image binarization. Since we used baseline architectures for object



(a) Binarized (GT) Images



(b) Binarized (Pred.) Images



(c) Color Images

Figure 5: Quantitative evaluation of OCR performance for different image qualities.

detection and image binarization, we expect that performance can be increased significantly by resorting to state-of-the-art models. Finally, we compared different OCR engines and analyzed their dependence on image quality and image binarization. We found that binarization can boost performance for some OCR engines, while slight quality degradations in terms of image resolution do not significantly affect the text recognition quality. In future work, we plan to tackle the full source code retrieval from programming videos pipeline by leveraging the CodeSCAN dataset.

Several future directions are very interesting. Since we limit ourselves to screenshot analysis, the tracking, composing and synchronizing files during a video tutorial remains an open task. In addition, evaluation metrics could move from classical text recognition towards evaluating code executability and the tree distance to the original abstract syntax tree. Finally, the coding grid parameters could be estimated using an additional *coding grid head*. Its availability is expected to significantly simplify the identification of the correct indentation.

REFERENCES

Alahmadi, M., Hassel, J., Parajuli, B., Haiduc, S., & Kumar, P. (2018). Accurately Predicting the Location of Code Fragments in Programming Video Tutorials Using Deep Learning. *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*.

Alahmadi, M., Khormi, A., & Haiduc, S. (2020a). UI Screens Identification and Extraction from Mobile Programming Screencasts. *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*.

Alahmadi, M., Khormi, A., Parajuli, B., Hassel, J., Haiduc, S., & Kumar, P. (2020b). Code localization in programming screencasts. *Empirical Software Engineering*, 25 (2).

Alahmadi, M. D. (2023). VID2XML: Automatic Extraction of a Complete XML Data From Mobile Programming Screencasts. *IEEE Transactions on Software Engineering*, 49 (4).

Bao, L., Li, J., Xing, Z., Wang, X., Xia, X., & Zhou, B. (2017). Extracting and analyzing time-series HCI data from screencaptured task videos. *Empirical Software Engineering*, 22 (1).

Bao, L., Li, J., Xing, Z., Wang, X., & Zhou, B. (2015). Reverse engineering time-series interaction data from screen-captured videos. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

Bao, L., Pan, S., Xing, Z., Xia, X., Lo, D., & Yang, X. (2020a). Enhancing developer interactions with programming screencasts through accurate code extraction. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Bao, L., Xing, Z., Xia, X., & Lo, D. (2019). VT-Revolution: Interactive Programming Video Tutorial Authoring and Watching System. *IEEE Transactions on Software Engineering*, 45 (8).

Bao, L., Xing, Z., Xia, X., Lo, D., Wu, M., & Yang, X. (2020b). Psc2code: Denoising Code Extraction from Programming Screencasts. *ACM Transactions on Software Engineering and Methodology*, 29 (3).

Bergh, A., Harnack, P., Atchison, A., Ott, J., Eiroa-Lledo, E., & Linstead, E. (2020). A Curated Set of Labeled Code Tutorial Images for Deep Learning. *2020 19th IEEE International Conference on Machine Learning and Applications(ICMLA).*

Fang, S., Xie, H., Wang, Y., Mao, Z., & Zhang, Y. (2021). Read Like Humans: Autonomous, Bidirectional and Iterative Language Modeling for Scene Text Recognition. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition.*

Gupta, A., Vedaldi, A., & Zisserman, A. (2016). Synthetic Data for Text Localisation in Natural Images. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

He, K., Gkioxari, G., Dollar, P., & Girshick, R. (2017). Mask R-CNN. *IEEE International Conference on Computer Vision (ICCV).*

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Huang, Z., Chen, K., He, J., Bai, X., Karatzas, D., Lu, S., & Jawahar, C. V. (2019). ICDAR2019 Competition on Scanned Receipt OCR and Information Extraction. *2019 International Conference on Document Analysis and Recognition (ICDAR).*

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2017). Image-to-Image Translation with Conditional Adversarial Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Jaderberg, M., Simonyan, K., Vedaldi, A., & Zisserman, A. (2016). Reading Text in the Wild with Convolutional Neural Networks. *International Journal of Computer Vision*, 116 (1).

Jaderberg et al., 2014). Deep Features for Text Spotting. *Computer Vision – ECCV 2014.*

Li, H., Wang, P., Shen, C., & Zhang, G. (2019). Show, Attend and Read: A Simple and Strong Baseline for Irregular Text Recognition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33 (01).

Li, M., Lv, T., Chen, J., Cui, L., Lu, Y., Florencio, D., Zhang, C., Li, Z., & Wei, F. (2023). TrOCR: Transformer-Based Optical Character Recognition with Pre-trained Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37 (11).

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature Pyramid Networks for Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft COCO: Common Objects in Context. *Computer Vision – ECCV 2014.*

Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. *5th International Conference on Learning Representations (ICLR).*

Lu, N., Yu, W., Qi, X., Chen, Y., Gong, P., Xiao, R., & Bai, X. (2021). MASTER: Multiaspect non-local network for scene text recognition. *Pattern Recognition*, 117.

Malkadi, A., Alahmadi, M., & Haiduc, S. (2020). A Study on the Accuracy of OCR Engines for Source Code Transcription from Programming Screencasts. *Proceedings of the 17th International Conference on Mining Software Repositories.*

Ott, J., Atchison, A., Harnack, P., Bergh, A., & Linstead, E. (2018a). A deep learning approach to identifying source code in images and video. *Proceedings of the 15th International Conference on Mining Software Repositories.*

Ott, J., Atchison, A., Harnack, P., Best, N., Anderson, H., Firmani, C., & Linstead, E. (2018b). Learning lexical features of programming languages from imagery using convolutional neural networks. *Proceedings of the 26th Conference on Program Comprehension.*

Ponzanelli, L., Bavota, G., Mocci, A., Di Penta, M., Oliveto, R., Hasan, M., Russo, B., Haiduc, S., & Lanza, M. (2016a). Too Long; Didn't Watch! Extracting Relevant Fragments from Software Development Video Tutorials. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).*

Ponzanelli, L., Bavota, G., Mocci, A., Di Penta, M., Oliveto, R., Russo, B., Haiduc, S., & Lanza, M. (2016b). CodeTube: Extracting relevant fragments from software development video tutorials. *Proceedings of the 38th International Conference on Software Engineering Companion.*

Ponzanelli, L., Bavota, G., Mocci, A., Oliveto, R., Penta, M. D., Haiduc, S., Russo, B., & Lanza, M. (2019). Automatic Identification and Classification of Software Development Video Tutorial Fragments. *IEEE Transactions on Software Engineering*, 45 (5).

Smith, R. (2007). An Overview of the Tesseract OCR Engine. *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, 2.

Wang, T.-C., Liu, M.-Y., Zhu, J.-Y., Tao, A., Kautz, J., & Catanzaro, B. (2018). High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

Yadid, S., & Yahav, E. (2016). Extracting code from programming tutorial videos. *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.*

Yang, C., Thung, F., & Lo, D. (2022). Efficient Search of Live-Coding Screencasts from Online Videos. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering.*

Zhao, D., Xing, Z., Chen, C., Xia, X., & Li, G. (2019). ActionNet: Vision-Based Workflow Action Recognition From Programming Screencasts. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).*