UNIVERSITY OF CALIFORNIA SAN DIEGO

Scaling Computer Science Education Through Live Coding and Streaming Systems

A thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Charles Hsien-Che Chen

Committee in charge:

Philip Guo, Chair
Joe Politz
Leo Porter

2019

The thesis of Charles Hsien-Che Chen is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge my God whom I love; my strength, my rock, my fortress, and my deliverer. “My son, keep my words and treasure up my commandments with you; keep my commandments and live; keep my teaching as the apple of your eye;” Proverbs 7:1-2

I would like to acknowledge Christine H. Lee, my mother who has spent her entire life pouring, investing, and nurturing my soul to become the man that I am today. She has shown me what it means to love someone with one's heart, soul, mind, and strength.

I would like to acknowledge Philip J. Guo for his dedication and investment into my master's degree, for being willing to take me under his wings and teach me about CS education.

I would like to acknowledge Leonard E. Porter and Joseph G. Politz for their participation on the thesis committee as well as feedback to the paper.

Chapter 1 through chapter 7 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

ABSTRACT OF THE THESIS

Scaling Computer Science Education Through Live Coding and Streaming Systems

by

Charles Hsien-Che Chen

Master of Science in Computer Science

University of California San Diego, 2019

Professor Philip Guo, Chair

Livestreaming promotes dynamic learning between audiences and instructors by allowing live coding, live feedback, and impromptu reaction to the audience. Howevever, livestreaming tutorials are often hard to follow because of a variety of context switches, varying environmental setups, and hiccups in presentation. Other video streaming mediums such as video conference calls allow specific tutor to tutee interactions for communicating coding concepts but lacks scalability and the intuitiveness of an in-person contact received from real-time reaction to instructions. We seek to provide high fidelity interactions and communications that is scalable for computer science education through the livestreaming medium by looking at two designed systems: Improv and TutorX. Improv is a programming presentation platform that enables streamers to model live

coding settings informed by Mayer's principles of multimedia learning [May09]. Improv enables instructors to synthesize blocks of code, output slides, and create preset waypoints to guide their presentations. A case study on 30 educational videos containing 28 hours of live coding showed that Improv was versatile enough to replicate approximately 96% of the content within those videos. TutorX is a streaming application modeled after video conferences where a tutor teaches a tutee. In our needfinding, tutors in video conferences have expressed difficulties in gauging the tutee's grasp of coding syntax, semantics and concepts. TutorX keeps track of application specific inputs and aggregates that information for the tutor to provide targeted feedback to a specific user or group of users. Tutors can send messages and observe how users are doing during each tutoring session by tracking operating system events through informed "smart" messages in order to gauge how tutees comprehend the material. Tutees can inform tutors via quick question prompts much like clicker questions [PBC+16].

Introduction

A Tale of Two Systems

Livestreaming has been an up and rising platform over the past few decades. With the arrival of platforms like Twitch.tv and Livecoding.tv, users can watch a variety of videos that stream a range of subjects such as gaming, art, cooking, and much more. Twitch allows a platform of high-fidelity videos and low-fidelity interactions via its IRC chat functionality [HGK14]. Just on Twitch alone there are over 100,000 communities [Twi17] that come together as communities to engage in activities of common interest and social interactions [HBNSH18]. With a platform that supports livestreams, there are those who livestream their coding skills in order to teach and form communities together to learn about coding [Han14]. Twitch and similar streaming services provide a platform for live-coding tutorials. Live coding is where teachers would write a block of code live, run the code, and then describe the code as it happens. As of the time of writing for this thesis, little to no research has been done on how one can leverage live coding in livestreaming environments in order to teach computer science. The thesis will go in-depth on two particular systems: Improv and TutorX.

Improv is an extension to a programming IDE that allows coding examples to be beamed to a web-based powerpoint like presentation for multiple students. Improv aims to reduce cognitive load by reducing context switching within live coding tutorials. Improv provides a presenter that

contains a live terminal, live code blocks, and live web browser. We also perform a case study on 30 educational videos totaling about 28 hours of footage and found Improv to be able to model most live coding interactions with a 96% accuracy. Chapter 1 through 7 provide details about this work. Improv was published as a paper titled “Improv: Teaching Programming at Scale via Live Coding” in ACM conference on Learning at Scale 2019.

TutorX is a work-in-progress one-to-many or one-to-one conference streaming application seeking to provide high-fidelity environments for tutors to tutees. During design studies, tutors in tutoring settings often find themselves lacking feedback from the tutee because of the limits of communication in video conferences. TutorX aims to address this issue by tracking operating system events and providing filtered information to tutors in order to allow them to gauge the tutee’s understanding of material. Scalability is also supported, by allowing one tutor to group multiple tutees based off their actions. TutorX is a system that is still in development. Descriptions of the system and further potential work can be found in chapter 8.

Chapter 1

Improv Introduction

Computer programming instructors frequently perform live coding in settings ranging from MOOC lecture videos to online livestreams. However, there is little tool support for this mode of teaching, so presenters must now either screen-share or use generic slideshow software. To overcome the limitations of these formats, we propose that programming environments should directly facilitate live coding for education. We prototyped this idea by creating Improv, an IDE extension for preparing and delivering code-based presentations informed by Mayer's principles of multimedia learning. Improv lets instructors synchronize blocks of code and output with slides and create preset waypoints to guide their presentations. A case study on 30 educational videos containing 28 hours of live coding showed that Improv was versatile enough to replicate approximately 96% of the content within those videos. In addition, a preliminary user study on four teaching assistants showed that Improv was expressive enough to allow them to make their own custom presentations in a variety of styles and improvise by live coding in response to simulated audience questions. Users mentioned that Improv lowered cognitive load by minimizing context switching and made it easier to fix errors on-the-fly than using slide-based presentations.

Chapter 2

Improv Overview

A popular way to teach computer programming, both online and in-person, is for the instructor to write snippets of code, run it, and then explain what their code does. By livestreaming or recording these performances, instructors can easily share technical insights with thousands of viewers on learning at scale platforms such as MOOCs, YouTube, and webinars. This sort of live coding now takes place in diverse settings:

- Instructors write code live in front of their classrooms. Computing education researchers recommend this as a best practice since students can see their instructors' thought processes, watch how mistakes are made and corrected, and ask clarifying questions at each step [GL07, Pax02, RPHH18, Wil18].
- Similarly, instructors of online courses broadcast their live programming in webinars ("web seminars"). They also record these sessions as videos for MOOCs and YouTube.
- Programmers write code live on stage during industry conference presentations, which are recorded to share with the wider professional community. They like doing live demos to convey a greater sense of authenticity and realism [Wil18].
- Programmers in domains such as game development livestream their coding sessions on sites such as **`Twitch.tv`** and **`Livecoding.tv`** to educate their online fans [Hin17, Ros15].

Despite the prevalence of live coding for education, current programming environments (IDEs) provide no support for this type of activity. Thus, presenters usually end up screensharing and recording their entire desktop displays. This setup is cumbersome since there are lots of extraneous on-screen components that are not relevant to the code-related ideas that the presenter is trying to convey at each moment. Also, it can be awkward to switch contexts in the middle of a live demo by moving and flipping between windows on the desktop. Finally, it is hard to present higher-level concepts such as topic outlines or algorithm descriptions by sharing only the contents of one's code development environment.

An alternative format is for the presenter to copy-and-paste all of their relevant code and output snippets into pre-made PowerPoint slides. This format has the advantage of greater structure and predictability. However, slide presentations can appear stilted, inauthentic, and not in sync with real working code. Also, presenters cannot as easily improvise in response to audience questions.

To overcome the limitations of these existing presentation formats, we propose that *programming environments (IDEs) should directly facilitate teaching via live coding*. To prototype this idea, we developed a system called Improv that helps instructors prepare and deliver code-based presentations entirely from within their IDE. Its design was informed by our formative studies and by Mayer's principles of multimedia learning [May09] from educational psychology. Figure 2.1 shows an example usage scenario for Improv:

1. The instructor writes and tests their code in any language normally within the Atom IDE [ato18]. Improv extends Atom with shortcuts that allow them to select any piece of code or terminal output in order to embed a live synced view of that snippet into PowerPoint-style slides.
2. Improv also extends Atom with an embedded slide presentation editor to drag-and-drop components into each slide. Supported components include: live code and output selections from their IDE, text annotations, images, and iframe-embedded contents from any webpage.

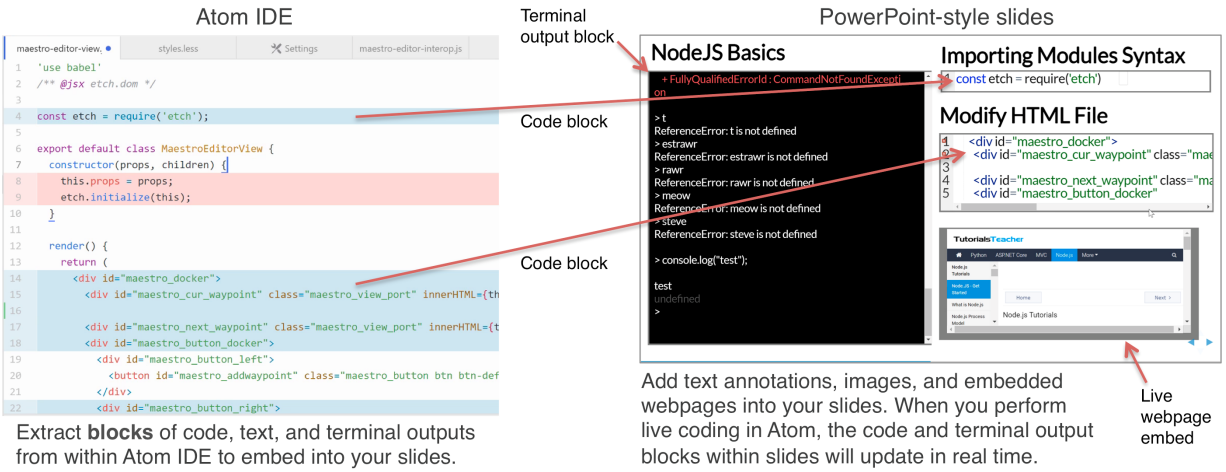


Figure 2.1: Improv augments the Atom IDE with UI affordances for preparing and delivering live coding presentations. The screenshots above show the Atom IDE (left) and the slide viewer that the audience sees (right). Figure 5.1 shows Improv's slide editor, and Figure 5.2 shows its code waypoints feature.

3. The instructor plans their live demo by creating optional *code waypoints* and *subgoal labels* [Cat98, MMG15] for what code they plan to write at each step. These serve as scaffolding during the presentation so that they can remain on course and so that the audience also knows what to expect at each step.

4. When the instructor presents live, the audience always sees a fullscreen view of the slides instead of the entire computer desktop (right part of Figure 2.1). As they write and run code within Atom, their audience sees the embedded code and output snippets update in real time on slides.

5. If they need to improvise in response to audience questions, they can edit code and slides on-the-fly in the middle of a talk, and the audience sees all updates in real time. They can also snapshot their code so that they can quickly restore it and get back on track after they are done improvising.

Instructors can use Improv either in a traditional in-person lecture setting or in a learning at scale setting by livestreaming or video-recording their hybrid code+slide presentations. Since it is web-based, remote viewers can connect to the Improv server to see the instructor's live

demonstration and copy-paste code snippets into their own IDEs to follow along.

To evaluate the versatility and expressiveness of Improv, we ran a pair of complementary studies. We first performed a case study on 30 videos containing 28 collective hours of live coding presentations in settings ranging from university lectures to online livestreams. We found that Improv was versatile enough to be used to present approximately 96% of the content within those videos. We then performed a preliminary user study by letting four first-time users prepare and deliver 10-minute coding tutorial presentations using Improv. We found that Improv was expressive enough to allow them to create their own custom presentations in a variety of styles and improvise by live coding in response to simulated audience questions. Users said that Improv lowered cognitive load by minimizing context switching and made it easier to fix errors and improvise than using slide-based presentations.

This thesis' contributions are:

- A formative study of 20 educational videos to characterize the diverse settings in which people perform live coding.
- The idea that existing IDEs should add integrated support for teaching programming via live coding.
- A prototype of this idea in the Improv system. Improv introduces new UI affordances, informed by Mayer's principles of multimedia learning [May09], that help instructors prepare and deliver code-based presentations within their IDE. We evaluated Improv with a case study on 30 videos and a preliminary user study on four teaching assistants.

Chapter 2 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

Chapter 3

Improv Related Works

Researchers have mostly studied live coding in educational settings [RPHH18]. As a pedagogical best practice they recommend having an instructor write and explain code live in the classroom or on video. Benefits include: making the instructor's step-by-step thought processes explicit [GL07, Pax02], enabling instructors to respond to "what-if?" questions from students by editing their code on-the-fly [Wil18], forcing instructors to incrementally build up code and narrate aloud rather than showing large blocks of code at once [Wil18], revealing sources of common coding mistakes [GL07], making the instructor more relatable since students can see that they make mistakes as well [BGDR05], and holding students' attention better since live coding is more dynamic than static PowerPoint slides [Rub13].

Although live coding is now common place in livestreaming tutorials, there could be contention that live coding is not fully shown to be effective due to a lack of clear quantitative evidence [RPHH18]. Much methodologies and course design has been centered around live coding [Pax02, Wil18] but there is still no assuring positive quantitative consensus. Improv does not attempt to argue that live coding is the magical bullet to effectively teach programming; however, Improv attempts to improve a popular coding tutorial technique by encouraging a system to promote reduced cognitive load for audiences. Improv encourages instructors that already work

off existing and prior coding to do so in such a way that can be as clear to the students as possible.

Live coding is currently done by sharing the presenter's screen with their audience (via a projector or online video stream) as they write and run code in text editors, terminals, IDEs, or, more recently, computational notebooks (e.g., Jupyter [KRKP+16], Codestrates [RNA+17]). Live streamers sometimes use video mixing software such as OBS [obs18] to broadcast only selected parts of their monitors, manage multi-monitor setups, and display custom banners on their streams [Hin17]. Since computational notebooks mix narrative text and code, some presenters manually scroll through them as a way of narrating their code-based live demos. Users have restyled the CSS of Jupyter notebooks to make them look more like PowerPoint slides [Avi17]. Similar restylings can theoretically be done on Codestrates notebooks [RNA+17] as well. IDEs such as Cloud9 [clo18], Visual Studio [vsl18], CodePilot [WG17], and Codechella [GWZ15] support real-time multi-user code editing akin to Google Docs; this feature can be used as a form of "IDE screensharing" when giving talks. However, none of these tools were designed with structured presentation planning and delivery as their use case. In contrast, Improv integrates a slide-based presentation system and live coding environment into a programmer's workflow within an IDE. The next section (Formative Study and Design Goals) will highlight limitations of current systems and how they inspired the design of Improv.

More broadly, Improv contributes to the long lineage of HCI research in presentation systems by being the first, to our knowledge, to be designed specifically for live coding presentations. One major class of work here extends Microsoft PowerPoint: TurningPoint [PYE14] implements six narrative templates derived from guides of best practices centered on storytelling techniques; users fill in the templates, and the system automatically generates starter slides. StyleSnap and FlashFormat help authors edit a large collection of PowerPoint slides to maintain consistent visual style across similar elements [EGMF+15]. HyperSlides [ESY13] helps authors plan hierarchical and non-linear navigation paths using a markup language. In contrast to slide-based presentation systems, tools such as Pad++ [BH94], CounterPoint [GB02], and Fly [LKB09] use a canvas

metaphor where presenters lay out elements in arbitrary locations on a zoomable plane. However, all of the above systems are meant for general-purpose presentations, whereas Improv is specialized for code-related demos that mix live programming and traditional slides. Improv improves upon general-purpose presentation systems by adding novel interactions for interfacing with a programmer's workflow within an IDE.

An increasing amount of related work has been done on subgoal labeling, which is supported by Improv through the waypoint system [Cat98]. Although the waypoint system is designed for instructors to first and foremost organize their thoughts, instructors can utilize waypoints to refer to a premade subgoal on their slides. Subgoal labeling for students in computer science have been shown to be effective in efficient learning as compared to students who have not received subgoal labels [MMG15]. General learning of computational problems appear to benefit less from subgoal labeling than from other academic fields because of factors such as variable name retention and textual language familiarity [MMG15]. Although Improv does not further contribute discussion on this growing subject in computer science education, Improv seeks to provide a platform for further discussion by enabling instructors to develop courses with subgoal labeling for both novice and adept students in various computer science domains.

Chapter 3 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

Chapter 4

Improv Formative Study and Design Goals

To understand how educators currently perform live coding and to inform the design of Improv, we ran a formative study by watching 20 programming videos (Table 4.1). Although no small sample can be universally representative, we strove for diversity in venues, modalities, and programming languages; Figure 4.1 shows selected screenshots from these videos. These code-based presentations were recorded in university lecture halls, at software industry conferences, from livestreams on **`Twitch.tv`**, and from screencast tutorials on YouTube. Here are our most salient observations from watching these videos:

Context switching and visual noise: In Table 4.1, the "Features" column shows that most presentations featured more than one medium (e.g., IDE, web browser, and slides). Presenters frequently switched between app windows; even when the web browser was the active window (e.g., Figure 4.1a), they often switched between browser tabs. Some arranged their windows in split-screen views (Figure 4.1c and e) while others left their desktops messy with overlapping windows and visual noise in taskbar and dock icons (Figure 4.1f).

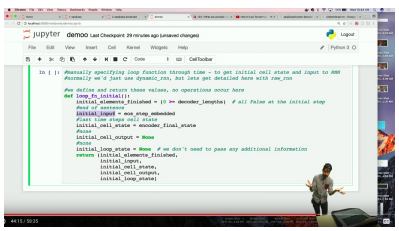
Presenters usually opened all relevant windows and application tabs before their talk began and flipped through them during their talk. For instance, Figure 4.1a shows 8 open web browser tabs, Figure 4.1e shows 29 browser tabs and 7 source code tabs in the IDE, and Figure 4.1f shows

Table 4.1: Corpus of live coding videos for our formative study. URLs should be prepended with `https://goo.gl/`. Feature abbreviations: E=editor (e.g., Emacs, Vim), I=IDE, J=Jupyter notebook, Q=questions fromaudience, S=slide presentation, T=terminal, W=web browser

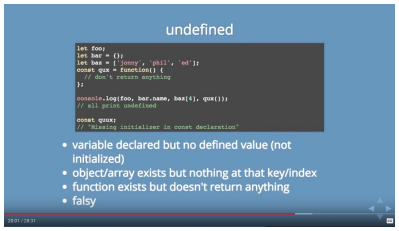
URL	Video Title (abbreviated)	Language	Features
	University Classroom Lectures		
xhgYsn	Harvard CS50: Web Development Tech	HTML/CSS	E,S,T,W
BjHrZA	Haverford College CMSC245: Pointers	C++	E,S,T
	Code-Based Conference Presentations		
17h8Be	Tricky JS Interview Questions	JS	S
XSx3eS	Creating Electronic Dance Music	JS, Node.js	E,S
itJTjb	Python And The Blockchain	Python	J,S
M7cL5W	Web programming from the beginning	Python	E,Q,T,W
KJAAaX	Introduction to Statistical Modeling	Python	J,Q,T,W
GbSTWy	Time Series Analysis	Python	J,Q,S
o9ySGS	PLOTCON 2016: Dash: Shiny for Python	Python	I,Q,W
F9Jwv9	What Does It Take To Be An Expert?	Python	E,Q,T,W
	Coding Livestreams on https://twitch.tv		
jEjRkp	Advice for Writing Small Programs in C	C	I,Q,S,T
F6sBGj	Private Data & Getters/Setters (Epic rant!)	C++	E,Q
E8RkKu	Building a Website - Live Coding w/ Jesse	HTML/CSS	E,W
	Coding Screencast Tutorial Videos		
5irDUF	Learn PHP in 15 minutes	PHP	E,S,W
b1pKf3	Ruby Essentials for Beginners	Ruby	E,T
fYtjsB	C# programming tutorial - Step by Step	C#	I,S
5sLQ5V	Frequency Analysis with FFT	JS, p5.js	E,Q,W
QJQiKM	Tensorflow for Seq2seq Models	Python	J
KGx9V6	MongoDB Quickstart with PyCharm	Python	I,Q,S,W
WpTnuY	Python Tutorial for Beginners #1 - Variables	Python	J,S

7 browser tabs and 4 terminal app tabs. This complexity made it hard for them to find specific windows on-demand, and they sometimes lost their place when navigating between windows. In contrast, those who projected full-screen slideshows did not worry about context switching. However, slideshows lack the dynamism of live coding performances.

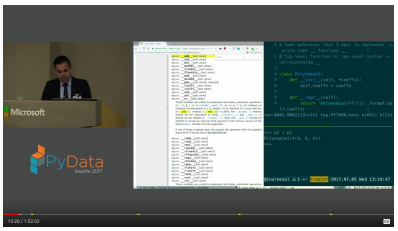
Slides + live coding: Presenters often used pre-made slides to convey higher-level concepts and then performed live coding to demonstrate those concepts in practice. As Figure 4.1b



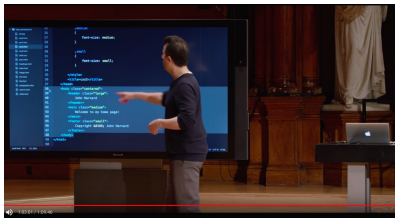
(a) Live coding in Jupyter notebook
(URL: QJQiKM)



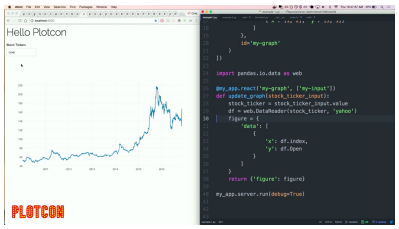
(b) Slides with non-runnable code
(URL: 17h8Be)



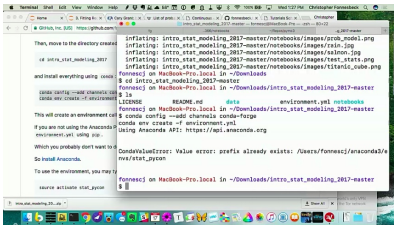
(c) Web browser & editor split-screen
(URL: F9Jwv9)



(d) Pointing to text editor in lecture
(URL: xhgYsn)



(e) Web app and IDE split-screen
(URL: o9ySGS)



(f) Many tabs & overlapped windows
(URL: KJAAaX)

Figure 4.1: Screenshots from videos in Table 4.1 that show the diversity of modalities in live coding presentations. (Prepend `https://goo.gl/` to URLs.)

shows, some slides interspersed (non-runnable) code with bullet-point descriptions of their properties. After explaining the code snippets on those slides, presenters then had to context-switch over to their text editor or IDE to edit a runnable version of that code in a live demo. If they want to update their presentations, they would need to keep both the within-slides and within-IDE versions of their code in sync.

Highlighting code and outputs: While live coding, presenters often selected text ranges in the editor to highlight a piece of relevant code as they explained its purpose (Figure 4.1d). They also switched to terminal windows to show textual output for command-line-based programs, or a browser to show visual output for web applications and interactive data visualizations. Some presenters used Jupyter notebooks to show both code and output together (Figure 4.1a); they scrolled through the notebooks to highlight relevant parts.

Typos, commented-out code blocks, and copy-and-paste: One big risk of live coding is that the presenter may make mistakes. To reduce this risk, some presenters placed pre-made commented-out blocks of code to use as references while they were live coding. Others copied-

and-pasted snippets from auxiliary files into their code to avoid manually typing everything from scratch. However, doing so detracts from the authenticity of a fully-live performance. Ideally a presentation system would let presenters write code live and provide a safety net to fall back on in case they made mistakes.

Improvising in response to live questions: One major benefit of live coding is the ability to improvise in response to questions (the "Q" feature in Table 4.1). The audience asked questions verbally during class lectures and conference talks and via text chat in Twitch.tv livestreams. The presenter would modify their code and re-run it to demonstrate their answers. Afterward they need to remember what they were working on before the question and restore their original code.

Based on both these firsthand observations and by consulting Mayer's principles of multimedia learning [May09], we developed the following design goals for our Improv system:

- **D1**: Minimize context switching and visual noise to help both the presenter and audience focus better
- **D2**: Integrate presentation slides with live runnable code
- **D3**: Support highlighting of code and outputs within slides
- **D4**: Minimize the risk of errors while live coding
- **D5**: Enable improvising and quickly restoring prior context

Chapter 4 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

Chapter 5

Improv System Design and Implementation

Improv integrates into a programmer's existing workflow within an IDE so they can minimize context switching (Design Goal D1). It is implemented with standard web technologies as an add-on for Atom, an extensible IDE [ato18]. Improv's slide viewer uses Reveal.js [rev18] to display web-based slides and Meteor.js [met18] to perform real-time syncing.

Improv contains a set of novel interaction techniques for extracting code, creating and presenting slides, and adding instructional scaffolding via code waypoints and subgoal labels.

5.1 Extracting Code Blocks and Terminal Outputs from Atom

A programmer starts using Improv by creating a code project in any language within the Atom IDE and testing to make sure their code works as intended. Then they select blocks of code from their source files to include in their presentation slides.

When the user selects a piece of text in either a code editor buffer or a terminal pane within Atom (which shows live-updated contents of a shell, REPL, or compiler output), they can

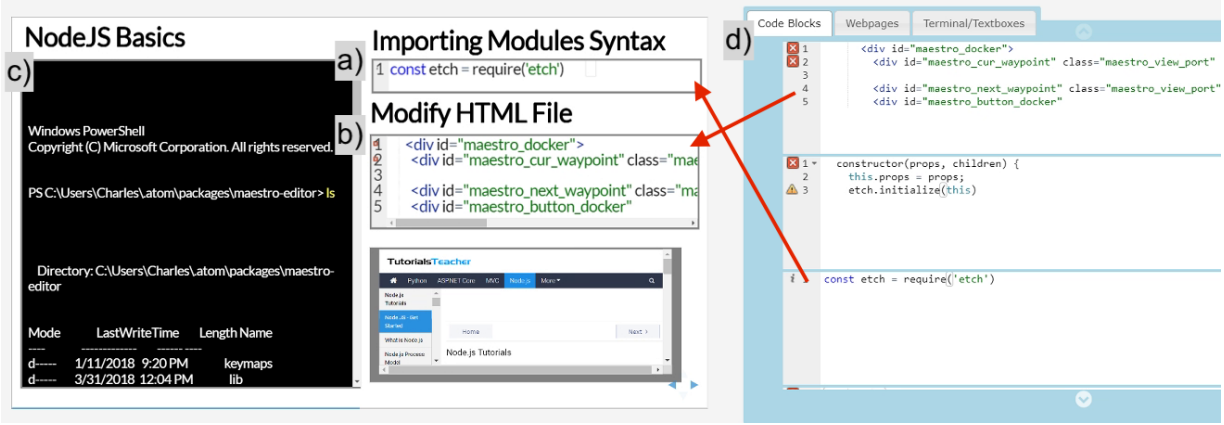


Figure 5.1: Improv's *slide editor* is a pane within the Atom IDE. Here it shows two code blocks (a & b) and a terminal output block (c) that were extracted from Atom in Figure 2.1. d) Extracted blocks are first put in a storage bin. The presenter can drag-and-drop these blocks, webpage embeds, and other elements onto slides; they can also add/remove/reorder slides. The *slide viewer* that the audience sees (right of Figure 2.1) is synchronized with this editor.

use a keyboard shortcut to extract that selection into a *code block*. They can also select the entire file to extract as a single block. (We call this a *code block* for simplicity, although the user can extract any part of any Atom text buffer.)

Each selection gets colored in light blue within Atom (left half of Figure 2.1). When the user makes later edits within its range, the highlighted area will grow or shrink accordingly. These ranges get properly preserved even if new text is inserted above or below the selections. If the user erases everything within the selection (or its enclosing file gets deleted), then its corresponding code block gets deleted too. Users can select and extract any number of code blocks across any files in Atom. Each block is put into a storage bin in the presentation editor (Figure 5.1d), which can be dragged onto slides.

5.2 Slide Presentation Editor

Improv augments Atom with a simple slide presentation editor situated in a new tab within the IDE. Figure 5.1 shows the UI of the slide presentation editor, which mimics a simplified

version of PowerPoint or Keynote. The user can create, reorder, and delete slides. Within each slide, they can add text and images with direct manipulation. We implemented only basic slide editor functionality and did not replicate more advanced features such as animated transitions or style guides. Improv's editor supports two novel types of slide elements specialized for our use case of live coding performances:

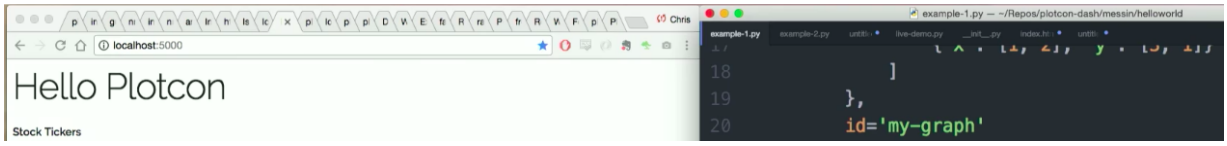
Code block elements: All code blocks extracted from Atom appear in a storage bin area at the right of the slide editor (Figure 5.1d). The user can drag and drop these blocks onto any slide just like how they can insert text and images. This way, a single code block can appear on multiple slides. Code blocks in slides are synced in real time with their corresponding selected regions in the IDE. Thus, when the user edits that code, it will also update on the slide(s). When the user's code runs and produces output, any embedded terminal blocks also update live (Figure 5.1c). This feature allows presenters to create slides that mix text and runnable code (Design Goal D2).

Even if the presenter does *not* want to perform live coding and simply wants to deliver a standard slideshow presentation, this live code block feature is still beneficial for two reasons: 1) It avoids having to keep two copies of code in sync between the IDE and slides, 2) It helps ensure that code which appears on slides actually compiles and runs, since *it is real working code that can be executed and tested within the IDE*. Otherwise it is easy for subtle typos and bugs to creep into code-based slides, which causes learner frustration [MG17].

Webpage iframe elements: The editor also allows the user to live-embed any webpage as an iframe into their slides. This is important because many of the live coding videos we watched featured presenters navigating through webpages such as API documentation, data visualizations, and Jupyter notebooks. The user can scroll to any portion of the webpage to start showing that part during the presentation, which is convenient for long webpages such as API docs or Jupyter notebooks.

These features give presenters both the organizational benefits of pre-made slides and the dynamism of live code and webpages. To demonstrate the utility of embedded code and webpage

elements, here is a zoomed-in view of the top part of the video in Figure 4.1e from our formative study corpus:



This presenter has 29 web browser tabs open (left half) and 7 source code tabs open in their IDE (right half), which they had to juggle throughout their talk. They had so many open browser tabs that they could not even see the tab titles. Throughout the talk, they also had to constantly scroll to different parts of webpages and source code files to find the right parts to discuss. If they had used Improv, they would have been able to selectively embed the desired portions of webpages and source code files into a series of slides with a logical ordering and accompanying slide titles for exposition.

5.3 Slide Viewer: Presentation Delivery and Live Coding

After the user creates a code-based presentation within Atom, they can also deliver their presentation entirely within Atom.

To do so, they first open a new web browser window and point it to a localhost URL for the slide viewer app. This viewer is a webpage that synchronizes its contents in real time with the currently-active slide displayed in Atom's slide editor. Then they connect their laptop to a projector and move the slide viewer window to the projected screen in full-screen mode. This way, the presenter still sees their own Atom IDE (and everything else on their desktop) while the audience sees only the viewer app on the projected screen. This minimizes visual noise (Design Goal D1) since the audience no longer sees the entire contents of the presenter's desktop. It also conforms to Mayer's *coherence principle* of multimedia learning [May09], which posits that people learn better when extraneous visual elements are excluded from view to minimize

distractions.

Alternatively, the presenter can host the slide viewer web app on a public IP address. This way, audience members (either in the room or remotely on the internet) can connect to that IP to watch the presentation live from their own web browsers. They can also copy-and-paste the code shown in the presentation to experiment with it locally in their own IDEs.

Since the contents of Improv's slide editor are always in sync with the slide viewer, *there is no distinction between presentation editing and delivery modes*. The audience always sees the currently-active slide in the editor. To deliver a presentation, the presenter simply flips through their slides in sequence in the editor. If they want to create new slides or modify the contents of existing slides on-the-fly, the audience will see those changes immediately. In addition, besides showing traditional slideshows, Improv's slide viewer also has support for webpage iframes, code blocks, and live coding:

Presenting webpage iframe elements: Since webpages are embedded as iframes, when the presenter scrolls through each iframe in the slide editor, Improv synchronizes its current scroll position with the viewer app so that the audience sees that same scrolling happening. This way, the presenter can walk through each page's contents by scrolling and narrating. The viewport sizes of the editor and viewer iframes are identical, so webpages render identically in both when scrolling.

Presenting code block elements: Likewise, the presenter can scroll through code blocks in the slide editor, and the audience again sees a synchronized view. This is effective for explaining pre-written static blocks of code, but what happens when the presenter wants to write code live?

Live coding: To perform live coding anytime during a presentation, the presenter clicks a button atop an embedded code block in the slide editor to jump to the portion of the original source file in the Atom IDE where that code was extracted from. They should now see that selection of code highlighted in yellow, which indicates that it is being projected on the active slide for the audience to view (Figure 5.2a). They can edit that code normally within Atom,

and all updates will propagate live to the slide viewer app. In addition, in the slide viewer that code block auto-scrolls so that its current cursor position is vertically centered. This ensures that currently-edited code is always visible to the audience.

In our example, the contents of the code blocks labeled ‘a’ and ‘b’ in Figure 5.1 always remain in sync across the Atom code editor, slide presentation editor, and slide viewer. Similarly, the terminal output block (Figure 5.1c) is also synced.

While the audience sees only what is on the slides, the presenter can perform live coding within their IDE with full access to all of the programming assistance tools they are accustomed to. The presenter can also access other desktop apps, API documentation, or speaker notes “behind the curtains” without the audience seeing or getting distracted by them.

If the presenter highlights a selection of text as they are live coding, that visual highlight will also appear on the slides for the audience to view (Design Goal D3). This allows the presenter to point out specific pieces of code or terminal output to explain it in detail.

Improv gives instructors a great deal of flexibility in terms of how to structure their live coding sessions. For instance:

- To demonstrate how a terminal-based program works (e.g., a Python or C program), the presenter can create a slide that contains a code block alongside a terminal output block.
- To show how to build a web application or interactive visualization, the presenter can create a slide with a code block alongside a webpage iframe pointed to the web application that is currently under development.
- To teach a classroom lesson on algorithms, the presenter can embed text and images of the relevant concepts alongside a live code block that shows its implementation.
- To demonstrate how to use a particular API, the presenter can embed a webpage of the API docs next to a code block.

One recommended way to organize slides is to have each slide hold one part of the overall

live demo. This way, presenters can advance through all slides in sequence and perform live coding on the appropriate parts of their codebase without fumbling to look for the relevant source files or browser tabs.

5.4 Code Waypoints and Subgoal Labels

It can be hard to write code live during a presentation without making mistakes, so the presenters in the formative study videos we watched often resorted to copying and pasting pre-written blocks of code from other text buffers into their target source files. Unfortunately, doing so detracts from the authenticity and natural flow of truly doing live coding. To preserve such authenticity while also guarding against errors (Design Goal D4), Improv lets users define a set of *code waypoints* for each code block (inspired by navigational waypoints [way18]).

After the user extracts a code block in the IDE by making a text selection, they can optionally open an inline pane to add waypoints for that given block (Figure 5.2). Each waypoint is a manually-defined version of the code in that block. It is up to the user to fill out the contents of each version, but one heuristic is to have each version represent a new concept or step that is being incrementally built up within that block.

The user can also add a *subgoal label* to each waypoint, which is a textual annotation that describes the purpose (goal) of what that waypoint aims to achieve. Subgoal labeling is a pedagogical best practice whereby the instructor adds a higher-level conceptual description for each group of steps in a tutorial; it has been shown to help improve learner comprehension in a variety of STEM domains [Cat98, MGC12, WKGM15].

Live coding with waypoints: If waypoints are set on a given code block, then when the presenter is live coding in there, they see not only their current code but also the waypoint's code in a separate pane (Figure 5.2c). This pane serves as a "teleprompter" that gives them a visual indicator of what code they ought to be writing at the moment to reach that waypoint. It is also

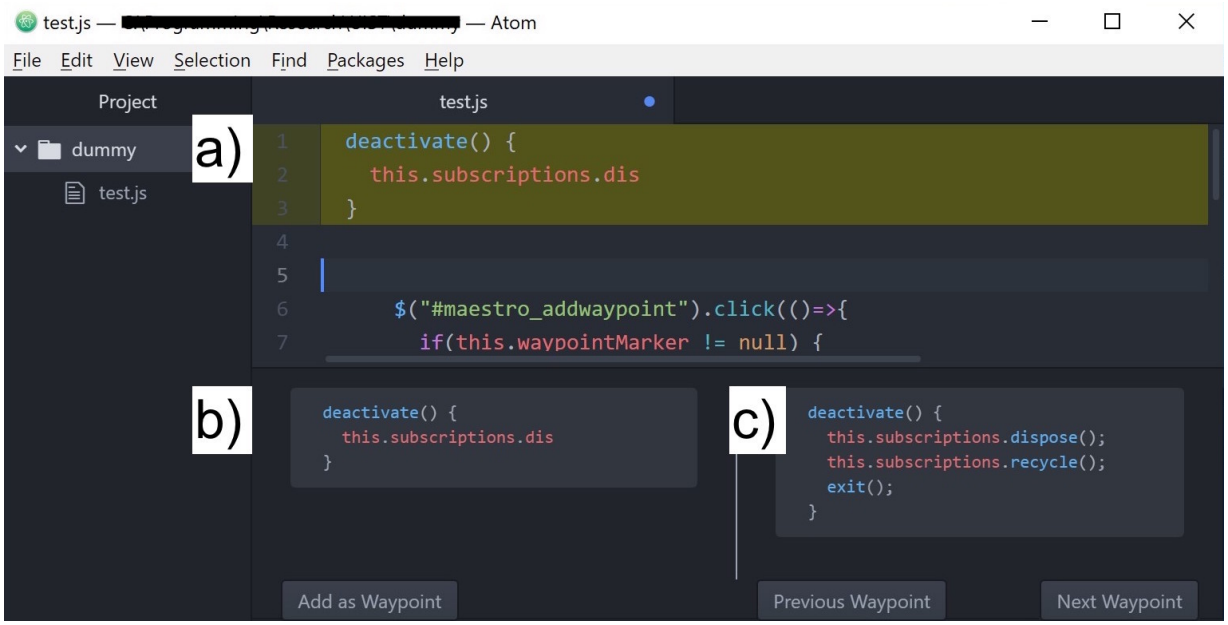


Figure 5.2: Improv lets users define waypoints to serve as scaffolding for live coding sessions. a) The code block that is projected on the current slide is highlighted in yellow. b) Click to add its state as a new waypoint. c) See a preview of the next waypoint, and navigate to other waypoints.

similar to how DemoWiz [CLD14] cues the presenter by showing previews of upcoming events on screencast videos.

In addition, the waypoint's subgoal label is always displayed above the code block in the slide viewer app so that the audience can see the purpose of the current step that is being demonstrated by the presenter. This feature abides by Mayer's *segmenting and signaling principles* of multimedia learning [May09], which posits that people learn better when presentations are divided into logical segments with cues that signal what to expect from each segment.

Once the presenter has written enough code in the current block so that it exactly matches the expected contents of the current waypoint, Improv automatically moves onto displaying the next waypoint. The presenter can always deviate from the waypoint if they want to improvise or format their code in a different way. In those cases, their end state will not be an exact string match, so they can manually move onto the next (or previous) waypoint with navigation controls.

If the presenter gets stuck or lost while live coding, they can click the "next waypoint"

button in the waypoint pane to copy the contents of the current waypoint into their code block and move onto the next waypoint. Although this action breaks the authenticity of live coding, it is convenient for getting the presenter out of a jam and ensuring that they still have working code (assuming that their pre-written waypoint code works and that they have not broken any code outside of that block).

For example, Figure 5.2 shows the waypoint pane in Atom. As the presenter is in the middle of writing the body of the `deactivate()` method in the yellow code block, they can see the next waypoint they are supposed to reach in order to complete the current segment of their demo (Figure 5.2c).

Impromptu waypoints: Finally, when the presenter gets a question from the audience or otherwise wants to improvise, they can click the "Add as Waypoint" button on a code block to save a snapshot of its current contents as a new waypoint. This way, they can freely modify their code to address the given question and quickly restore their original code afterward to return to their main presentation (Design Goal D5).

Chapter 5 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

Chapter 6

Improv Evaluation

To assess Improv's versatility and expressiveness, we ran a pair of studies to investigate the following questions:

- Is Improv *versatile* enough to be used to prepare and deliver a diverse variety of realistic code-based presentations?
- Is Improv *expressive* enough to let first-time users create presentations of their own original design?

6.1 Case Study of Coding Presentation Videos

To assess the versatility of Improv, we performed a case study on 30 programming tutorial videos from YouTube.

Procedure: We used the 20 videos in our formative study corpus (Table 4.1). To guard against "overfitting" on this initial corpus, we found 10 additional videos using a similar methodology but *after* finishing the implementation of Improv. These represent a diverse selection of code-based presentations that people have delivered to a range of audiences. We watched each video in detail and hand-classified the time durations within each one based on what is being

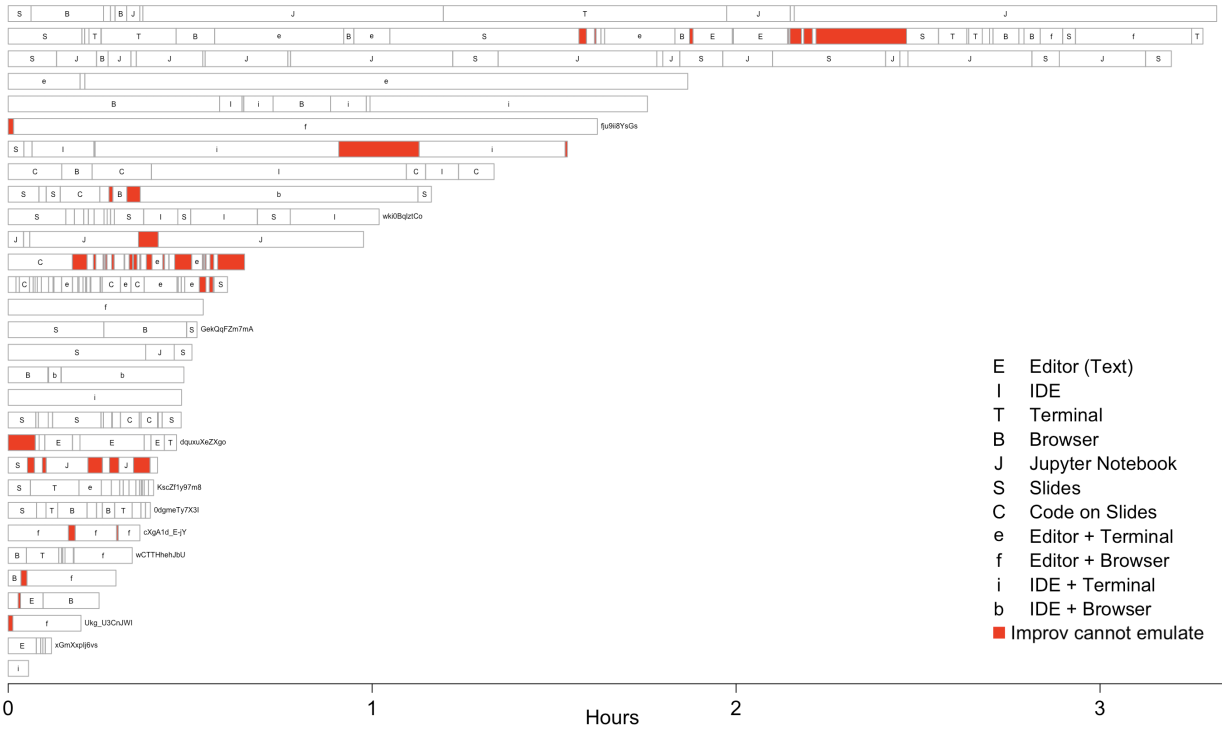


Figure 6.1: Presentation formats in 30 programming videos: 20 were from Table 4.1; YouTube IDs are next to 10 new videos. Red = Improv cannot emulate.

displayed on the screen at those times. Then we estimated the proportion of time that Improv could plausibly be used as a replacement for the presentation media that the speaker actually used.

Findings: Figure 6.1 summarizes our findings. The entire duration of each video is a horizontal bar, and the labeled portions represent time ranges when the presenter was working within a particular app (e.g., terminal, IDE, slides). The red portions represent time ranges where Improv *cannot* be used to emulate what the presenter was doing at those times (see details below). The red portions account for 4% of the total time across all 30 videos, which means that Improv can serve as a plausible replacement for presenting approximately 96% of the content in our 28-hour corpus of 30 live coding videos.

Presenters could have used Improv to create nearly all parts of these presentations, based on the apps being used in them:

- *Text Editor* (3% of total running time across all 30 videos): With Improv, they could have extracted a code block to place on slides and performed live coding within Atom.

- *IDE* (6% of total video time): Same as text editor. Note that presenters did not use advanced IDE features; they used IDEs only as elaborate text editors for live coding.
- *Terminal* (6%): Extract a terminal output block to slides.
- *Web Browser* (9%): Embed a webpage iframe into slides.
- *Jupyter Notebooks* (20%): Same as web browser. (Note we put Jupyter into its own category since many presenters used it as a web-based live programming environment.)
- *Slide Presentation* (13%): Use Improv's slide editor.
- *Code on Slides* (3%): These are slides that primarily showcase snippets of code. With Improv, they can extract code blocks so that those snippets update live on their slides.

Presenters also concurrently used two apps in either a split-screen view or by rapidly flipping back and forth: text editor + terminal (11% of total time in all videos), text editor + web browser (12%), IDE + terminal (9%), IDE + web browser (4%). Improv can handle all of these cases by placing multiple components on a single slide: either a code block and a terminal output block, or a code block and a webpage iframe.

Improv could not plausibly emulate what presenters featured during 4% of the total time in these 30 videos (4.7% of total time in the original 20 formative study videos and 2.3% in the additional 10 we picked). During those times, presenters used GUI applications, sketching, or physical demonstrations. The most common modality was the presenter demonstrating specific features of GUI applications such as the Ableton [abl18] music production software, the Wireshark [wir18] network analyzer, or the Windows process manager for showing memory usage patterns. Three presenters made digital sketches over their screens. Finally, one presenter projected an Oculus Rift headset [rif18] display for a virtual reality live coding demonstration. In the future, we could extend Improv to embed a live view of the presenter's computer desktop or external video feeds in order to support these modalities.

Study Limitations: This is an informal case study on a hand-picked video corpus. Even though we strove for diversity in presentation formats and selected 10 additional videos after finishing Improv's implementation, we cannot be sure that these are representative of all code-based presentations. Also, there is no guarantee that the presenters would actually prefer replacing their current setups with Improv, or how the audience would react to seeing Improv versions of these talks.

6.2 Preliminary User Study with Teaching Assistants

Our case study demonstrated the potential versatility of Improv across a variety of presentation formats, but we also wanted to see how expressive it is when put into the hands of first-time users. To do so, we performed a preliminary study with 4 computer science teaching assistants at our university.

Procedure: Each participant came to our lab to use Improv individually for 1 hour. We first gave them a 10-minute tutorial of Improv's basic features and then instructed them spend most of the hour using it to create a five to ten-minute code-related presentation of the sort that they would normally make for their courses. When they finished creating the presentation, we had them use Improv to deliver it to the facilitator, who interrupted with questions to simulate an audience. Finally, we concluded the session by asking them to reflect on the advantages and disadvantages of using Improv when compared to presentation tools that they had previously used.

Findings: All four participants were able to use Improv to successfully prepare and deliver a presentation of their own original design. These varied widely in subject matter presented, Improv features used, and presentation styles:

- P1 taught basic JavaScript by demonstrating a simple command-line calculator with Node.js. When preparing their talk, they wrote pieces of code in Atom and included them onto

three slides along with explanatory text. They did not originally intend to perform live coding. However, as they started presenting their slides and the facilitator interrupted with questions, they used the code waypoints feature to create impromptu waypoints so that they could perform live coding to address questions. Then they returned to prior waypoints to resume their talk. They also improvised by embedding a terminal onto a slide and live coding from the Node.js REPL to show additional concepts.

- P2 taught the concept of function calls using pseudocode. Their two slides consisted of images and pseudocode. Despite not performing any live coding, they still found it useful to organize pseudocode in text files within Atom and to selectively extract them onto slides. When the facilitator pointed out a possible bug in the pseudocode, they were able to fix it right away by editing that section in the text file and seeing the update instantly appear on the slides.
- P3 taught array operations in Python. They created three Python source code files in Atom, one for each concept: indexing, slicing, and element skipping. They tested their code separately in each file and then extracted each one to place on its own slide with a corresponding title. They also placed a terminal output block at the bottom of all slides. During their talk, they moved between the three slides and performed live coding in Atom; when their Python code executed, it showed up in the terminal block on each slide.
- P4 taught Python variables and print statements. They took the most dynamic approach out of all four by using only one slide and having it serve as a fullscreen canvas. They put a code block, terminal output, and webpage iframe into that single slide. While preparing the talk, they used the iframe to look up Python reference documentation rather than switching to an external web browser since they could see the reference in the same context as their code. While delivering the talk, they live coded from within Atom.

Improv supported both the more static slides-based presentations (P1 and P2) and the

more dynamic live coding sessions (P3 and P4). P1 was even able to switch from slides to live coding on-demand when the facilitator asked questions.

During post-study debriefing interviews, participants mentioned the following advantages of Improv as compared to existing presentation tools that they had previously used:

- Lower cognitive load when live coding, since they did not need to repeatedly switch back and forth between different apps such as IDEs, terminals, browsers, and PowerPoint.
- Relatively easy to fix errors on the fly by adjusting code or slide content directly from within Atom and having the audience see those changes immediately.
- P1, P3, and P4 felt that the clearest benefit of Improv was the ability to write and run code in a real IDE but to have the audience see that code on an organized set of slides.
- P1 found waypoints useful for improvising. P3 manually emulated waypoints by having an auxiliary notes file where they stored code snippets that they copied into their presentation code. When we reminded them about the waypoints feature at the end, they agreed it would have been useful but was not sufficiently familiar with it as a first-time user.
- P4 appreciated the flexibility of Improv's slide format compared to traditional presentation tools: *"I think of these slides more as workspaces. [...] You normally see a slide as very specific and like a state of mind that is progressing. But here the slide itself is dynamic. More like a workspace where you can drop code and dynamic stuff is happening."*

Participants also pointed out their perceived limitations of Improv. Most notably, they mentioned higher cognitive load during presentation planning, since they had to develop a mental model of how three separate components synced up with one another: their own code within Atom, the slide editor, and the slide viewer. Also, they were surprised that they could not edit code on the slides to fix minor issues but instead had to edit within the corresponding source code file in Atom. To support this, we can implement bidirectional syncing between Atom's code editor and Improv's slide editor.

In sum, first-time users found Improv expressive enough to use for preparing simple teaching presentations that mixed both code and expository content. All four participants reported being interested in using Improv in their own teaching.

Study Limitations: We conducted an informal first-use study without a control group. Although participants reflected on the experience of using Improv versus tools they previously used, we did not perform a formal comparison against existing tools. Also, due to the short study duration, participants used Improv to deliver at most a 10-minute presentation, so we were not able to assess its scalability for preparing, say, hour-long lectures with dozens of slides. Finally, the facilitator served as a simulated audience, but ideally Improv would be evaluated in a class setting to gauge real student reactions.

Chapter 6 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

Chapter 7

Improv Discussion and Conclusion

Improv explores the design space of educational presentation tools in between slide-based software (organized but static) and desktop screensharing (authentic but messy). It melds the organizational benefits of slide-based presentations with the authenticity and improvisational flexibility of live coding. Improv's code-based slide format gives presenters the ability to organize their content in accordance with pedagogical best practices such as Mayer's principles of multimedia learning [May09]: They can display code in large fonts, eliminate extraneous visual noise from desktop apps, and supplement code with textual annotations, images, and webpage embeds.

One main limitation, though, is that there are times when presenters want to project their entire computer desktop for the audience to view instead of presenting slides. This could arise because they want to demonstrate how to interact with a set of complex GUI applications. Improv is not well-suited for those use cases since, by design, it shows only selected code blocks within a traditional slide presentation. In the future, we could extend it with a screensharing plug-in that allows it to embed portions of the presenter's desktop into slides.

Currently instructors who want to teach programming to a large audience must rely on ad-hoc setups involving screen sharing, PowerPoint slides, and flipping between disparate desktop

applications. Our Improv system takes steps toward making this form of technical pedagogy both more organized and more fluid. From an educational technology perspective, *Improv's main contribution to learning at scale is to bring Mayer's principles of multimedia learning [May09] into the popular domain of live coding presentations.* We hope that by deploying this system in MOOCs and other online learning platforms in the future, students will be able to learn better by watching instructors more fluently combine live coding with slide-based presentations. They can either connect to the Improv slide viewer web app for a live broadcast or watch prerecorded videos of these hybrid code and slide presentations.

Chapter 7 of this thesis, in part, contains material as it appears in Improv: Teaching Programming at Scale via Live Coding. Chen, Charles, and Philip J. Guo. ACM Conference on Learning at Scale, 2019.

Chapter 8

TutorX Introduction

One of the biggest difficulties in teaching computer science during livestreaming is the high-fidelity visuals with low-fidelity interactions [HGK14]. Platforms like Twitch, provide viewers the ability to message the streamer and vice versa to communicate questions or ideas while the stream is ongoing. In coding tutorials, viewers can ask questions, respond to questions, and share their excitement about the streamed content. Streamers can also engage in the content by pinging questions to the audience and provide voting incentives for audience participation via chatbots [Moo08]. However two major problems exist in the communication of live coding tutorials: communication is limited to a web-based chat that cannot convey traditional coding environments effectively from the streamer especially in various OS setups and communication is hard to manage as participants increase.

Traditional video conferences provide much of the same benefits as livestreaming in programming education but trades scalability for student specific high-fidelity interactions via voice chat. Video conferences also provides facial feedback for intuitive in-person feedback. TutorX attempts to be the best of both worlds in livestreaming platforms and video conferencing platforms. Much like a livestream, TutorX streams the tutor's views to all users and does not allow voice communication from the tutee to tutor. Much like a video conference, TutorX allows tutors

to view each specific tutee's desktop on command and can see what the user is doing on their desktop via tutee specific action-drive “smart” messages. TutorX solve issues of communication in live coding tutorials by first introducing a more complex, event based, tutee action driven messaging system and introduce crowd sourcing tools to the tutor in order to promote coding specific learning. TutorX is a work in progress project, as a result, there are still a lot of potential implementations and tests that can still be done.

Chapter 9

TutorX System Design and Implementation

9.1 TutorX Overview

TutorX is an Electron based front-end application with NodeJS backend that creates a streaming environment for a tutor to teach single or multiple tutees like a group video conference. TutorX is split up into two main application views: tutor and tutee. Both views have similar layouts: a live video stream of the tutor on top of the sidebar and an events feed on the bottom. By default, both views start in a minimized side-bar form. Views can be toggled to an enlarged form where the live video stream is front and center for easier viewing. Much like a traditional streaming platform, TutorX provides a livestream view of the tutor's desktop for every user connected. From the tutor's perspective, the tutor can see various stats that their tutee sees. From the tutee's perspective, they see all the actions taken from the tutor in the form of message boxes.

9.2 Design Goals

Since TutorX is based off a video conferencing platform, we ran a series of needfinding in order to identify various pain points that users have experienced during a live tutoring session:

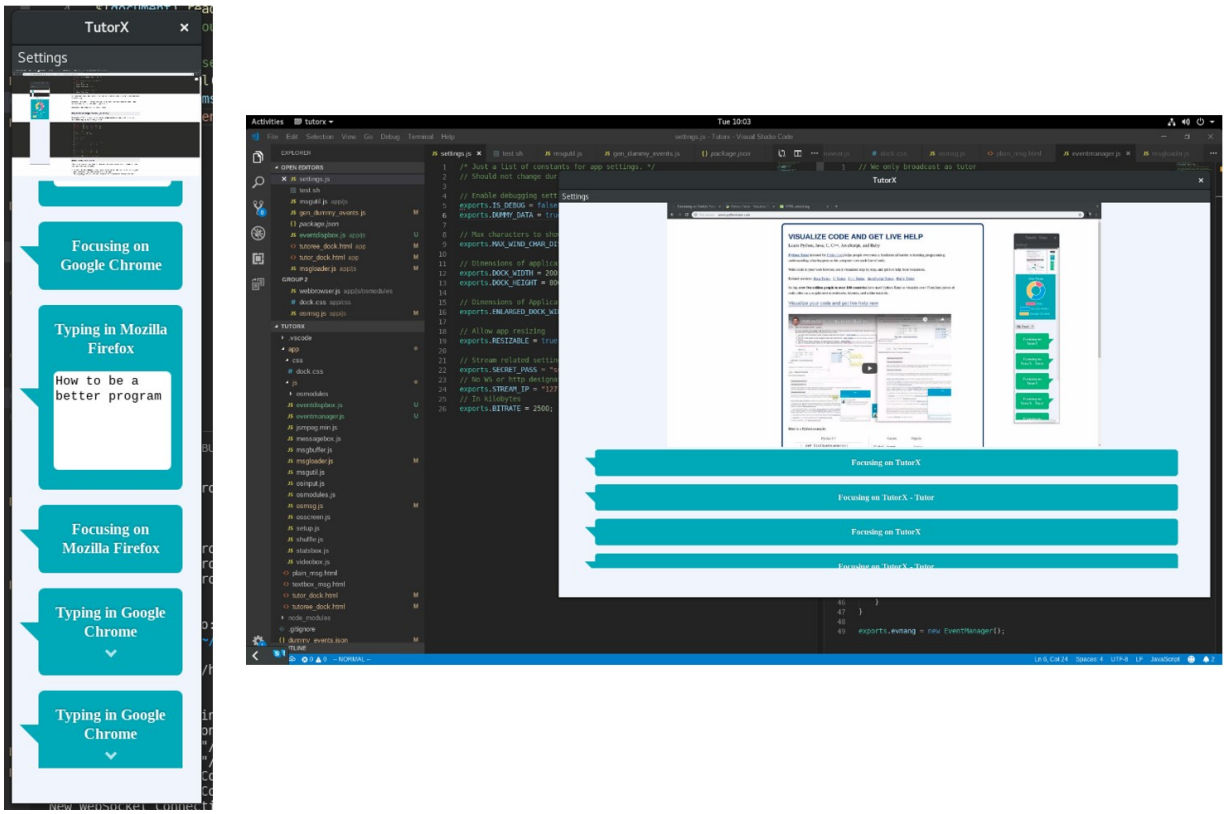


Figure 9.1: Tutee sidebar view (left) and tutee expanded view (right). Tutee can view the livestream displayed above from the tutor and get high-fidelity messages based off tutor's actions.

Over several video conference call on basic algorithms, a familiar pattern started to appear The tutee would take long extended periods of time to respond to questions or instructions that tutors would take long times to make questions. Tutors would then either be confused about the tutee's state or misunderstand the tutee's understanding of the material. Sometimes the tutor will think the tutee has understood the material and choose not to ask how the tutee is doing until much later. By then, the tutor will have to backtrack and try to figure out when the misunderstanding occurred. As a result, eventually more questions are asked by the tutor in order to clarify whether or not the tutee understood the material much more than normal in-person interactions. Occasionally tutees would make keyboard noises and other paper noises that causes responses of the tutor to clarify the situation. During live coding, many pauses in writing causes the tutor to ask how the tutee is doing. Occasionally tutors would interject to point out errors or inquire about what the tutee is doing without knowing whether or not the tutee is simply making a simple mistake or a deeper conceptual problem was at play. Communication for setting up environments became difficult as languages across various operating system (OS) platforms meant that often tutors have to switch their words for the feedback given for tutee. Occasionally the tutor would not know what to do because despite the desktop stream of the tutee, the tutor cannot manipulate the streams directly. This causes a lot of instructions that require backtracking on the part of the tutor for not knowing the environment well enough for precise instructions.

In accumulating needfindings, we've discovered several needs from various sessions. TutorX attempts to address each one through its design:

- **N1** Tutors and tutees have trouble trying to communicate what they are doing on their coding environments.
- **N2** Tutors have a very hard time trying to gauge tutee's understanding of coding concepts.
- **N3** As tutees increased, feedback became short and less specific to the user as an attempt to address the crowd; tutees have a hard time understanding feedback as tutees increased.

9.3 Addressing Communication Issues

TutorX seeks to send more precise information about what the streamer does by sending application specific operating system actions sorted and simplified via event-based messaging boxes. In traditional livestreaming platforms, all communication from the user to the streamer and vice versa occur through a text-based web messaging system. The limitations of a messaging system become very apparent as learners try to communicate what they are trying to do on a different OS environment.

TutorX solves this issue by recording the actions of the tutor or tutee and broadcasting each action in a neatly organized event box. Each action is grouped by application and minimized. These message boxes can be expanded to show the specific actions that the user took in the application. Every possible action that can be done on an OS can be captured such as keyboard presses, mouse clicks, and camera interfaces. As they input and interact with the application, these messages update to reflect the interaction. Since each message box is application based, tutors can see syntax and semantical issues based of the input tutees give to their coding editors. "Smart" messages are operating system and application invariant, that is, regardless of operating systems or the setup of the application per user, the events will display the same messages across all setups. This type of interaction provides uniformity across various platforms. Tutors are able to give feedback by using the same prescription across various OS. Event message boxes provide a high-fidelity view of what the tutor or tutee is doing during the live tutorial that is continuously updated, easily accessible, and invariant to every user's setup.

By default, the tutee sees every "smart" message by the tutor alongside their video stream (see figure 8.1). This allows the tutee to the specific actions that the tutors are performing on their screens that are fitted in messages that are OS independent. Tutors, on the other hand, can either choose to a specific tutee's actions to view in order to give better feed back or they can gauge the audience in general by accumulating actions per application.

In order to provide scalability, when connecting one user to another user, a live feed occurs for that specific user. That way, users can communicate to a specific user. This design implementation attempts to solves N1 and N2.

TutorX attempts to provide a solution to a large tutee participation in livestream settings by providing viewer activity data and clicker questions much like a traditional classroom setting [PBC+16]. TutorX can group tutees into various groups based off their actions. TutorX provides various actions for the tutor to gauge and interact with these groups. In the tutor view, TutorX provides a console to display user focus to gauge where the attention of the user is placed and clicker questions help assess the understanding of the user. Tutors can also choose to interact with the tutee by providing question in a form of prompts. In our system tests, we emulate a 300 tutee to 1 tutor session as seen in figure 8.2. With the aggregate info from the tutor's view, they can also message each specific group focusing on a specific window. They can also message specific people with specific inputs per application. With further tutor specified modifying specific actions and application hooks, tutors can have fine grain control to certain categories of users based off their actions.

With each tutee group, tutors can either send specifically catered clicker questions or one way messages in order to better help these groups. In the next iteration of TutorX, tutors can group by operating systems in order to promote environmental setup tutorials. The design choice attempts to solve N2 and N3.

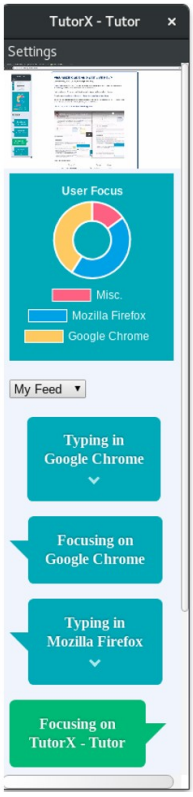


Figure 9.2: Tutor view provides a control panel of user focus to see what students are interacting with in the tutorial. Tutors can choose to ping each user based off group to provide user-specific like feedback [Blo84]. Here there are 300 simulated tutees connected. Their focus is displayed on the Tutor's window as a percentage.

Chapter 10

TutorX Future Work

Since TutorX is still in development, there is still a lot of exploration that can still be done with the system to further take advantage of the livestreaming medium. More can be explored in the realm of peer to peer interactions before tutor to tutee interactions. Groupings of various users into groups based off user-actions mean that we can allow peers with similar problems to interact with one another. One possible way to promote this interaction is the ability for users to send quick questions much like tutors do to tutees but with one another. Or being able to ping or message students. We can promote student to student interactions by prompting in the application.

More advanced prompts from tutors to tutee can be delivered through the visual feed of the stream. Presently, there is only a simple question and answer exchange between tutees represented by a simple message ping. More interactions can be done such as the highlights and overlays of the display in order to point tutees to regions of their screen. Tutors can try to view the stream from the tutee's perspective and point out problems in their code by simply drawing on their screen.

Promoting one to one higher-fidelity communication via in person cameras and bi-directional voice communication. Even though TutorX was inspired by video conferences,

one of the major components of video conferences, the bi-directional voice communication, was removed. The design was first removed because of a concern over scalability over practicability. Should the tutor be allowed to engage with voice chat in only select groups or specific persons? During voice chats, should the tutor be allowed to broadcast this communication to the rest of the tutees? More needfinding among group voice conferences and user testing can better propel this design choice.

Chapter 11

TutorX Conclusion

Overall, TutorX tries to balance between highly targeted feedback and large groups in livestreaming. Various tutees could have various different struggles, however, by observing their actions on their computers, tutors can try to split the audience into various group and address them individually based off coding needs. By providing the notion of groups and “smart” messages, tutees are able to better follow the tutor and provide tutor the ability to engage tutees in a manner that is based off the user's environmental setup, programming patterns, or syntax errors. Since TutorX is still in development there is still a lot of various interactions between groups and peer-to-peer that can be implemented or polished. Through the various design goals outlined above and in future work, TutorX attempts to address the common miscommunications errors by teaching computer science in a high-fidelity voice-chat streaming service caused by various environmental setups for coding and lack of coding specific feedback.

Bibliography

[abl18] Ableton: Music production with live and push. **https://www.ableton.com/en/**, 2018.

[ato18] Atom ide. **https://ide.atom.io/**, 2018.

[Avi17] Damian Avila. Rise: Reveal.js - jupyter/ipython slideshow extension. **https://damianavila.github.io/RISE/index.html**, 2017.

[BGDR05] Lecia J. Barker, Kathy Garvin-Doxas, and Eric Roberts. What can computer science learn from a fine arts approach to teaching? In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 421–425, New York, NY, USA, 2005. ACM.

[BH94] Benjamin B. Bederson and James D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, pages 17–26, New York, NY, USA, 1994. ACM.

[Blo84] Benjamin S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4–16, 1984.

[Cat98] Richard Catrambone. The subgoal learning model: Creating better examples so that students can solve novel problems. 127:355–376, 12 1998.

[CLD14] Pei-Yu Chi, Bongshin Lee, and Steven M. Drucker. Demowiz: Re-performing software demonstrations for a live presentation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 1581–1590, New York, NY, USA, 2014. ACM.

[clo18] Aws cloud9: A cloud ide for writing, running, and debugging code. **https://aws.amazon.com/cloud9/?origin=c9io**, 2018.

[EGMF+15] Darren Edge, Sumit Gulwani, Natasa Milic-Frayling, Mohammad Raza, Reza Adhitya Saputra, Chao Wang, and Koji Yatani. Mixed-initiative approaches to

global editing in slideware. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3503–3512, New York, NY, USA, 2015. ACM.

[ESY13] Darren Edge, Joan Savage, and Koji Yatani. Hyperslides: Dynamic presentation prototyping. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 671–680, New York, NY, USA, 2013. ACM.

[GB02] Lance Good and Benjamin B. Bederson. Zoomable user interfaces as a medium for slide show presentations. *Information Visualization*, 1(1):35–49, March 2002.

[GL07] Alessio Gaspar and Sarah Langevin. Restoring "coding with intention" in introductory programming courses. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education*, SIGITE '07, pages 91–98, New York, NY, USA, 2007. ACM.

[GWZ15] Philip J. Guo, Jeffery White, and Renan Zanelatto. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VL/HCC '15, pages 79–87, Oct 2015.

[Han14] Handmade hero. **https://handmadehero.org/**, 2014.

[HBNSH18] Zorah Hilvert-Bruce, James T. Neill, Max Sjöblom, and Juho Hamari. Social motivations of live-streaming viewer engagement on twitch. *Computers in Human Behavior*, 84:58 – 67, 2018.

[HGK14] William A. Hamilton, Oliver Garretson, and Andruid Kerne. Streaming on twitch: Fostering paticipatory communities of play within live mixed media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 1315–1324, New York, NY, USA, 2014. ACM.

[Hin17] Suz Hinton. Lessons from my first year of live coding on twitch. **https://medium.freecodecamp.org/lessons-from-my-first-year-of-live-coding-on-twitch-41a32e2f41c1**, 2017.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[LKB09] Leonhard Lichtschlag, Thorsten Karrer, and Jan Borchers. Fly: A tool to author planar presentations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 547–556, New York, NY, USA, 2009. ACM.

[May09] Richard E. Mayer. *Multimedia Learning*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[met18] Meteor: Build apps with javascript. **https://www.meteor.com/**, 2018.

[MG17] Alok Mysore and Philip J. Guo. Torta: Generating mixed-media gui and command-line app tutorials using operating-system-wide activity tracing. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 703–714, New York, NY, USA, 2017. ACM.

[MGC12] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 71–78, New York, NY, USA, 2012. ACM.

[MMG15] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 21–29, New York, NY, USA, 2015. ACM.

[Moo08] Moobot. **https://moo.bot/**, 2008.

[obs18] Open broadcaster software. **https://obsproject.com/**, 2018.

[Pax02] John Paxton. Live programming as a lecture technique. *J. Comput. Sci. Coll.*, 18(2):51–56, December 2002.

[PBC+16] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. A multi-institutional study of peer instruction in introductory computing. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 358–363, New York, NY, USA, 2016. ACM.

[PYE14] Larissa Pschetz, Koji Yatani, and Darren Edge. Turningpoint: Narrative-driven presentation planning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 1591–1594, New York, NY, USA, 2014. ACM.

[rev18] reveal.js - the html presentation framework. **https://revealjs.com/**, 2018.

[rif18] Oculus rift - oculus. **https://www.oculus.com/rift/**, 2018.

[RNA+17] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 715–725, New York, NY, USA, 2017. ACM.

[Ros15] Scott Rosenberg. The strange appeal of watching coders code. Backchannel: WIRED **https://www.wired.com/2015/08/the-strange-appeal-of-watching-coders-code/**, 2015.

[RPHH18] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. Role of live-coding in learning introductory programming. In *Rroceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, pages 13:1–13:8, new york, ny, usa, 2018. acm.

[Rub13] Marc J. Rubin. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 651–656, New York, NY, USA, 2013. ACM.

[Twi17] Twitch.tv. **twitch.tv**, 2017.

[vsl18] Visual studio live share: Real-time collaborative development. **https://code.visualstudio.com/visual-studio-live-share**, 2018.

[way18] Waypoint (wikipedia). **https://en.wikipedia.org/wiki/Waypoint**, 2018.

[WG17] Jeremy Warner and Philip J. Guo. Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 1136–1141, New York, NY, USA, 2017. ACM.

[Wil18] Greg Wilson. How to teach programming (and other things): Live coding. **http://third-bit.com/teaching/live.html**, 2018.

[wir18] Wireshark - go deep. **https://www.wireshark.org/**, 2018.

[WKGM15] Sarah Weir, Juho Kim, Krzysztof Z. Gajos, and Robert C. Miller. Learnersourcing subgoal labels for how-to videos. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 405–416, New York, NY, USA, 2015. ACM.