Improv: Teaching Programming at Scale via Live Coding

Charles H. Chen
UC San Diego
La Jolla, CA, USA
hsc052@ucsd.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT
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.

INTRODUCTION
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 [18, 27, 30, 35].
- 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 [35].
- 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 [21, 31].

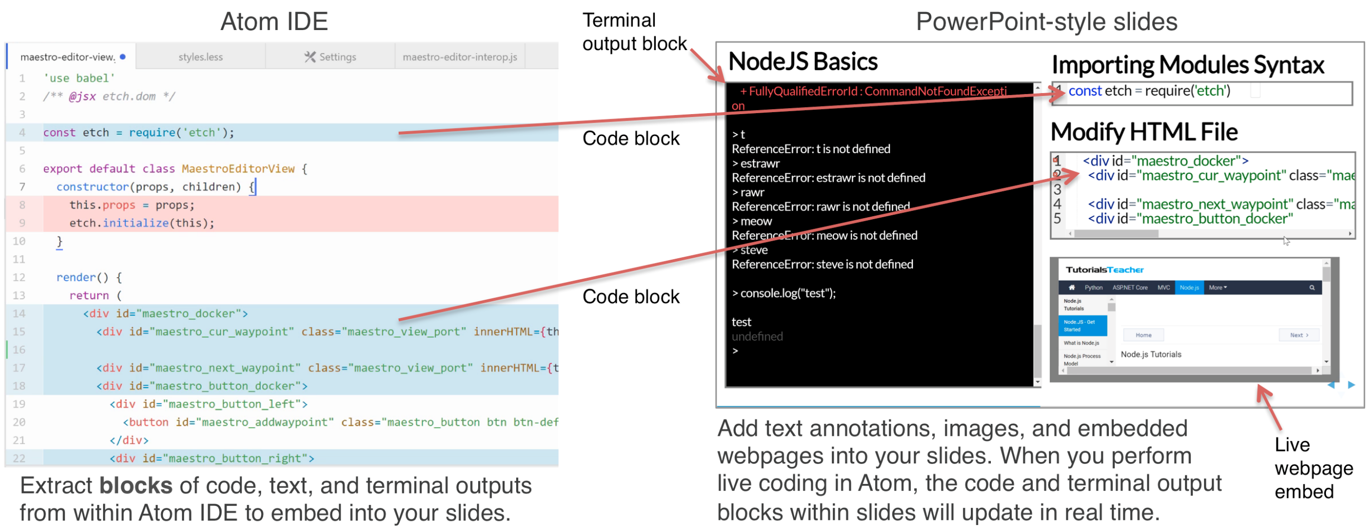
Despite the prevalence of live coding for education, current programming environments (IDEs) provide no support for this type of activity. Thus, presenters usually end up screen-sharing and recording their entire desktop displays. This setup is cumbersome since there are lots of extraneous on-screen 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 [25] from educational psychology. Figure 1 shows an example usage scenario for Improv:

1. The instructor writes and tests their code in any language normally within the Atom IDE [2]. 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.

L@S '19, June 24–25, 2019, Chicago, IL, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-6804-9/19/06...$15.00
DOI: https://doi.org/10.1145/3330430.3333627



```
'use babel'
/** @jsx etch.dom */

const etch = require('etch');

export default class MaestroEditorView {
  constructor(props, children) {
    this.props = props;
    etch.initialize(this);
  }

  render() {
    return (
      <div id="maestro_docker">
        <div id="maestro_cur_waypoint" class="maestro_view_port" innerHTML={th

        <div id="maestro_next_waypoint" class="maestro_view_port" innerHTML={t
        <div id="maestro_button_docker">
          <div id="maestro_button_left">
            <button id="maestro_addwaypoint" class="maestro_button btn btn-def
          </div>
          <div id="maestro_button_right">
```

Extract **blocks** of code, text, and terminal outputs from within Atom IDE to embed into your slides.

NodeJS Basics

```
  + FullyQualifiedErrorId : CommandNotFoundExcepti
on

> t
ReferenceError: t is not defined
> estrawr
ReferenceError: estrawr is not defined
> rawr
ReferenceError: rawr is not defined
> meow
ReferenceError: meow is not defined
> steve
ReferenceError: steve is not defined

> console.log("test");

test
undefined
>
```

Importing Modules Syntax

```
const etch = require('etch')
```

Modify HTML File

```
 <div id="maestro_docker">
  <div id="maestro_cur_waypoint" class="mae

  <div id="maestro_next_waypoint" class="ma
  <div id="maestro_button_docker"
```

Add text annotations, images, and embedded webpages into your slides. When you perform live coding in Atom, the code and terminal output blocks within slides will update in real time.

Figure 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 3 shows Improv's slide editor, and Figure 4 shows its code waypoints feature.

3. The instructor plans their live demo by creating optional *code waypoints* and *subgoal labels* [14] 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 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 paper's contributions to Learning at Scale 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 [25], 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.

RELATED WORK

Researchers have mostly studied live coding in educational settings [30]. 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 [18, 27], enabling instructors to respond to "what-if?" questions from students by editing their code on-the-fly [35], forcing instructors to incrementally build up code and narrate aloud rather than showing large blocks of code at once [35], revealing sources of common coding mistakes [18], making the instructor more relatable since students can see that they make mistakes as well [12], and holding students' attention better since live coding is more dynamic than static PowerPoint slides [32].

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 [22], Codestrates [29]). Live streamers sometimes use video mixing software such as OBS [6] to broadcast only selected parts of their monitors, manage multi-monitor setups, and display custom banners on their streams [21]. 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 [11]. Similar restylings can theoretically be done on Codestrates notebooks [29] as well. IDEs such as Cloud9 [3], Visual Studio [8], CodePilot [33], and Codechella [20] 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 [28] 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 [16]. HyperSlides [17] helps authors plan hierarchical and non-linear navigation paths using a markup language. In contrast to slide-based presentation systems, tools such as Pad++ [13], CounterPoint [19], and Fly [23] 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.

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 1). Although no small sample can be universally representative, we strove for diversity in venues, modalities, and programming languages; Figure 2 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 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 2a), they often switched between browser tabs. Some arranged their windows in split-screen views (Figure 2c and e) while others left their desktops messy with overlapping windows and visual noise in taskbar and dock icons (Figure 2f).

Presenters usually opened all relevant windows and application tabs before their talk began and flipped through them during their talk. For instance, Figure 2a shows 8 open web

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

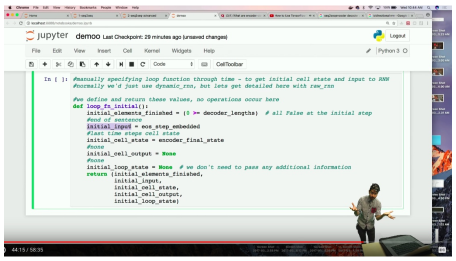
Table 1: Corpus of live coding videos for our formative study. URLs should be prepended with `https://goo.gl/`. Feature abbreviations: E=editor (e.g., Emacs, Vim), I=IDE, J=Jupyter notebook, Q=questions from audience, S=slide presentation, T=terminal, W=web browser

browser tabs, Figure 2e shows 29 browser tabs and 7 source code tabs in the IDE, and Figure 2f shows 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 2b 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.

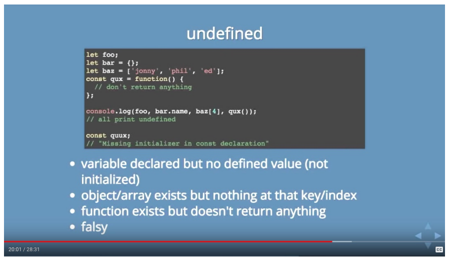
Highlighting code and outputs: While live coding, presenters often selected text ranges in the editor to highlight a piece of relevant code as they explained its purpose (Figure 2d). They also switched to terminal windows to show textual output for command-line-based programs, or a browser to show visual output for web applications and interactive data visualizations. Some presenters used Jupyter notebooks to show both code and output together (Figure 2a); 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-



```
In [ ]: #manually specifying loop function through time - to get initial cell state and input to RNN
        #normally we'd just use dynamic_rnn, but lets get detailed here with raw_rnn

        #we define and return these values, no operations occur here
        def loop_fn_initial():
            initial_elements_finished = (0 >= decoder_lengths)  # all False at the initial step
            #end of sentence
            initial_input = eos_step_embedded
            #last time steps cell state
            initial_cell_state = encoder_final_state
            #none
            initial_cell_output = None
            #none
            initial_loop_state = None  # we don't need to pass any additional information
            return (initial_elements_finished,
                    initial_input,
                    initial_cell_state,
                    initial_cell_output,
                    initial_loop_state)
```

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

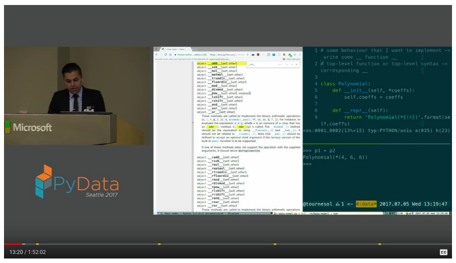


undefined

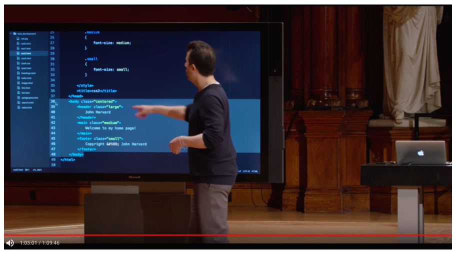
```
let foo;
let bar = {};
let baz = ['jonny', 'phil', 'ed'];
const qux = function() {
  // don't return anything
};

console.log(foo, bar.name, baz[4], qux());
// all print undefined

const quux;
// "Missing initializer in const declaration"
```

- variable declared but no defined value (not initialized)
- object/array exists but nothing at that key/index
- function exists but doesn't return anything
- falsy

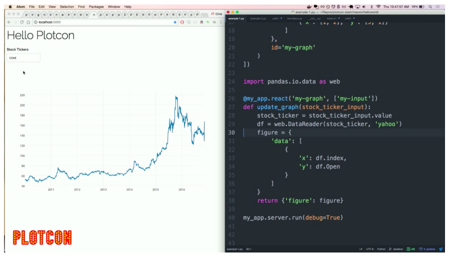
(b) Slides with non-runnable code (URL: 17h8Be)



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

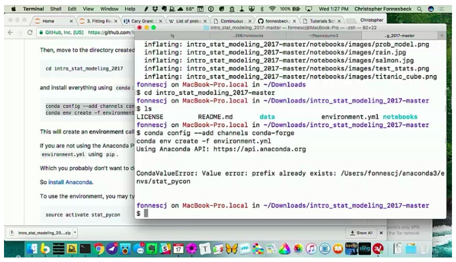


(d) Pointing to text editor in lecture (URL: xhgYsn)



```
            ]
        },
        id='my-graph'
    )
])

import pandas.io.data as web

@my_app.react('my-graph', ['my-input'])
def update_graph(stock_ticker_input):
    stock_ticker = stock_ticker_input.value
    df = web.DataReader(stock_ticker, 'yahoo')
    figure = {
        'data': [
            {
                'x': df.index,
                'y': df.Open
            }
        ]
    }
    return {'figure': figure}

my_app.server.run(debug=True)
```

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



intro_stat_modeling_2017-master — fonnescj@MacBook-Pro — -zsh — 80×22
Wed 1:27 PM Christopher Fonnesbeck

```
  inflating: intro_stat_modeling_2017-master/notebooks/images/prob_model.png
  inflating: intro_stat_modeling_2017-master/notebooks/images/rain.jpg
  inflating: intro_stat_modeling_2017-master/notebooks/images/salmon.jpg
  inflating: intro_stat_modeling_2017-master/notebooks/images/test_stats.png
  inflating: intro_stat_modeling_2017-master/notebooks/images/titanic_cube.png
fonnescj on MacBook-Pro.local in ~/Downloads
$ cd intro_stat_modeling_2017-master
fonnescj on MacBook-Pro.local in ~/Downloads/intro_stat_modeling_2017-master
$ ls
LICENSE         README.md       data            environment.yml notebooks
fonnescj on MacBook-Pro.local in ~/Downloads/intro_stat_modeling_2017-master
$ conda config --add channels conda-forge
conda env create -f environment.yml
Using Anaconda API: https://api.anaconda.org


CondaValueError: Value error: prefix already exists: /Users/fonnescj/anaconda3/e
nvs/stat_pycon


fonnescj on MacBook-Pro.local in ~/Downloads/intro_stat_modeling_2017-master
$
```

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

Figure 2: Screenshots from videos in Table 1 that show the diversity of modalities in live coding presentations. (Prepend `https://goo.gl/` to URLs.)

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 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 [25], 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

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 [2]. Improv's slide viewer uses Reveal.js [7] to display web-based slides and Meteor.js [4] 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.

Extracting Code Blocks and Terminal Outputs from Atom

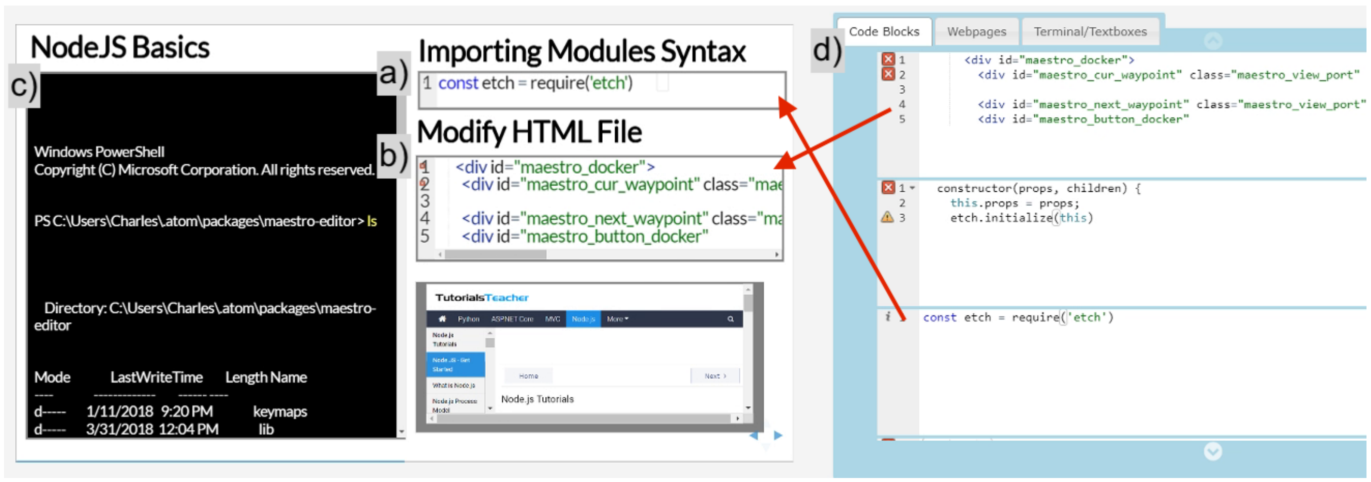
A programmer starts using Improv by creating a code project in any language within the Atom IDE and testing to make sure their code works as intended. Then they select blocks of code from their source files to include in their presentation slides.

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 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 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 3d), which can be dragged onto slides.

Slide Presentation Editor

Improv augments Atom with a simple slide presentation editor situated in a new tab within the IDE. Figure 3 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



```
const etch = require('etch')
```

```
<div id="maestro_docker">
  <div id="maestro_cur_waypoint" class="mae

  <div id="maestro_next_waypoint" class="ma
  <div id="maestro_button_docker"
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\Charles\.atom\packages\maestro-editor> ls

  Directory: C:\Users\Charles\.atom\packages\maestro-
editor

Mode        LastWriteTime     Length Name
----        -------------     ------ ----
d-----    1/11/2018  9:20 PM        keymaps
d-----    3/31/2018 12:04 PM         lib
```

```
<div id="maestro_docker">
  <div id="maestro_cur_waypoint" class="maestro_view_port"

  <div id="maestro_next_waypoint" class="maestro_view_port"
  <div id="maestro_button_docker"
```

```
constructor(props, children) {
  this.props = props;
  etch.initialize(this)
```

```
const etch = require('etch')
```

Figure 3: 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 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 1) is synchronized with this editor.

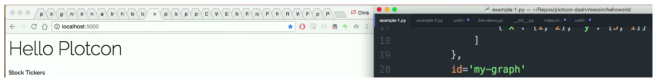
basic slide editor functionality and did not replicate more advanced features such as animated transitions or style guides. Improv's editor supports two novel types of slide elements specialized for our use case of live coding performances:

Code block elements: All code blocks extracted from Atom appear in a storage bin area at the right of the slide editor (Figure 3d). 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 3c). 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 [26].

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 2e from our formative study corpus:



Hello Plotcon

Stock Tickers

```
            ]
        },
        id='my-graph'
```

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.

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 [25], 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 4a). 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.

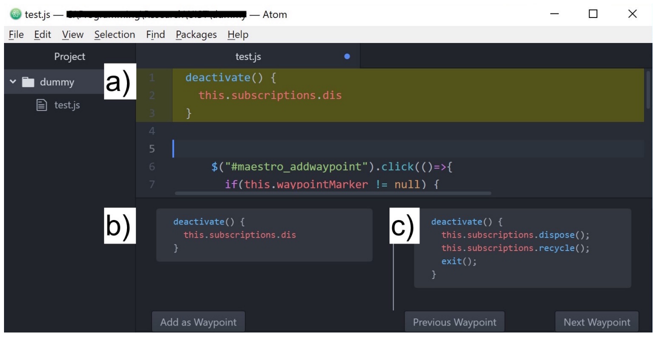
In our example, the contents of the code blocks labeled 'a' and 'b' in Figure 3 always remain in sync across the Atom code editor, slide presentation editor, and slide viewer. Similarly, the terminal output block (Figure 3c) is also synced.

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.



```
deactivate() {
  this.subscriptions.dis
}


    $("#maestro_addwaypoint").click(()=>{
      if(this.waypointMarker != null) {
```

```
deactivate() {
  this.subscriptions.dis
}
```

```
deactivate() {
  this.subscriptions.dispose();
  this.subscriptions.recycle();
  exit();
}
```

Figure 4: 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.

- 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.

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 [9]).

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 4). 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 [14, 24, 34].

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 4c). 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 similar to how DemoWiz [15] 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 [25], 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 4 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 4c).

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).

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?

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 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 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 5 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 [1] music production software, the Wireshark [10] network analyzer, or the Windows process manager for showing memory usage

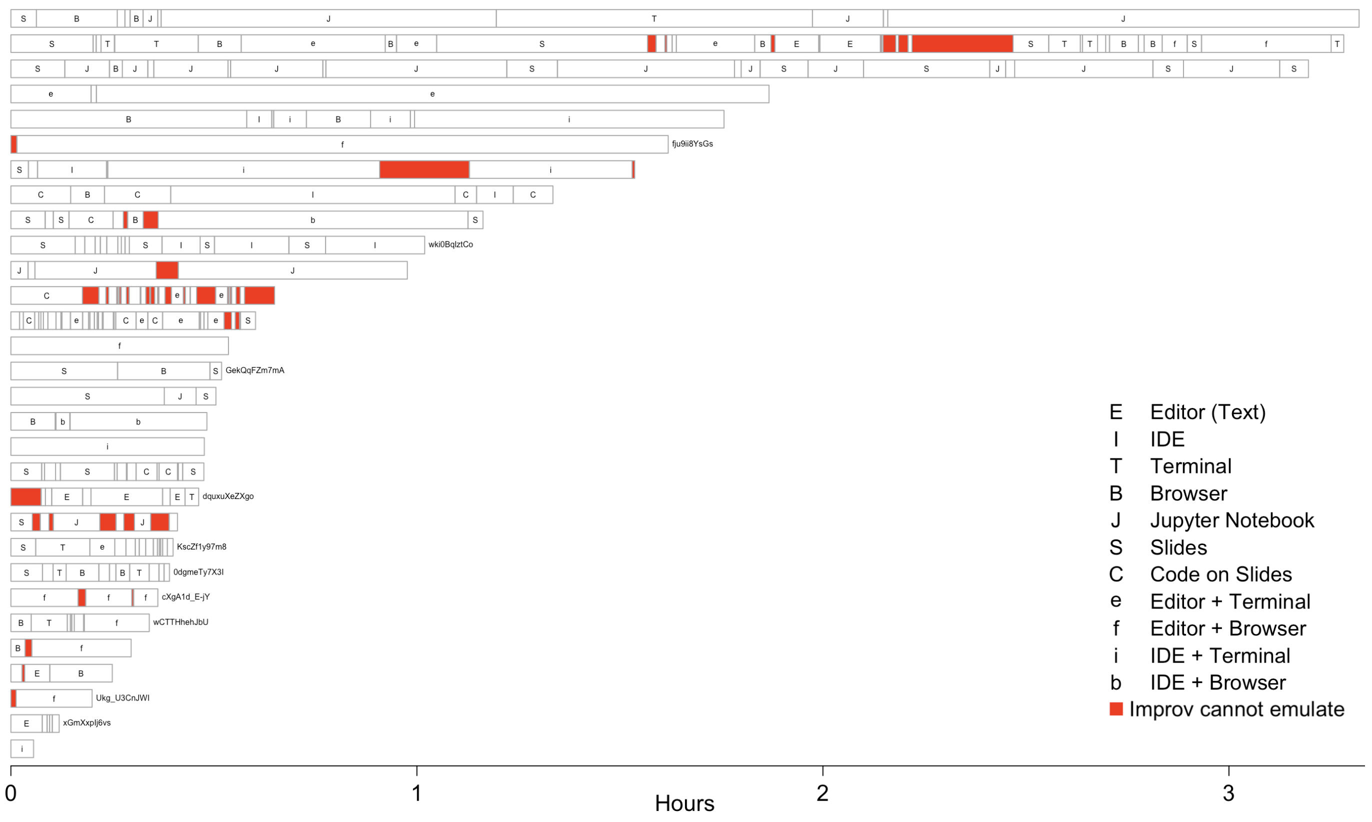


Figure 5: Presentation formats in 30 programming videos: 20 were from Table 1; YouTube IDs are next to 10 new videos. Red = Improv cannot emulate.

patterns. Three presenters made digital sketches over their screens. Finally, one presenter projected an Oculus Rift headset [5] 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.

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.

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 [25]: 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 [25] 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.

ACKNOWLEDGMENTS

Thanks to Darren Chan, Parmit Chilana, Jonathan Edwards, Ailie Fraser, Andrew Hild, Kandarp Khandwala, Sean Kross (for Figure 5!), Jeremy Lai, Christine Lee, and Alok Mysore for their feedback on the research that led to this paper.

REFERENCES

1. 2018. Ableton: Music production with Live and Push. **https://www.ableton.com/en/**. (2018).
2. 2018. Atom IDE. **https://ide.atom.io/**. (2018).
3. 2018. AWS Cloud9: A cloud IDE for writing, running, and debugging code. **https://aws.amazon.com/cloud9/?origin=c9io**. (2018).
4. 2018. Meteor: Build Apps with JavaScript. **https://www.meteor.com/**. (2018).
5. 2018. Oculus Rift - Oculus. **https://www.oculus.com/rift/**. (2018).
6. 2018. Open Broadcaster Software. **https://obsproject.com/**. (2018).
7. 2018. reveal.js - The HTML Presentation Framework. **https://revealjs.com/**. (2018).
8. 2018. Visual Studio Live Share: Real-time collaborative development. **https://code.visualstudio.com/visual-studio-live-share**. (2018).
9. 2018. Waypoint (Wikipedia). **https://en.wikipedia.org/wiki/Waypoint**. (2018).
10. 2018. Wireshark - Go Deep. **https://www.wireshark.org/**. (2018).
11. Damian Avila. 2017. RISE: Reveal.js - Jupyter/IPython Slideshow Extension. **https://damianavila.github.io/RISE/index.html**. (2017).
12. Lecia J. Barker, Kathy Garvin-Doxas, and Eric Roberts. 2005. 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)*. ACM, New York, NY, USA, 421–425.
13. Benjamin B. Bederson and James D. Hollan. 1994. 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)*. ACM, New York, NY, USA, 17–26.
14. Richard Catrambone. 1998. The Subgoal Learning Model: Creating Better Examples so That Students Can Solve Novel Problems. 127 (12 1998), 355–376.
15. Pei-Yu Chi, Bongshin Lee, and Steven M. Drucker. 2014. DemoWiz: Re-performing Software Demonstrations for a Live Presentation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 1581–1590.
16. Darren Edge, Sumit Gulwani, Natasa Milic-Frayling, Mohammad Raza, Reza Adhitya Saputra, Chao Wang, and Koji Yatani. 2015. Mixed-Initiative Approaches to Global Editing in Slideware. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3503–3512.
17. Darren Edge, Joan Savage, and Koji Yatani. 2013. HyperSlides: Dynamic Presentation Prototyping. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 671–680.
18. Alessio Gaspar and Sarah Langevin. 2007. Restoring "Coding with Intention" in Introductory Programming Courses. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education (SIGITE '07)*. ACM, New York, NY, USA, 91–98.
19. Lance Good and Benjamin B. Bederson. 2002. Zoomable User Interfaces As a Medium for Slide Show Presentations. *Information Visualization* 1, 1 (March 2002), 35–49.
20. Philip J. Guo, Jeffery White, and Renan Zanelatto. 2015. 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)*. 79–87.
21. Suz Hinton. 2017. 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).
22. 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. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.
23. Leonhard Lichtschlag, Thorsten Karrer, and Jan Borchers. 2009. Fly: A Tool to Author Planar Presentations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 547–556.
24. Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. 2012. 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)*. ACM, New York, NY, USA, 71–78.
25. Richard E. Mayer. 2009. *Multimedia Learning* (2nd ed.). Cambridge University Press, New York, NY, USA.
26. Alok Mysore and Philip J. Guo. 2017. 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)*. ACM, New York, NY, USA, 703–714.
27. John Paxton. 2002. Live Programming As a Lecture Technique. *J. Comput. Sci. Coll.* 18, 2 (Dec. 2002), 51–56.
28. Larissa Pschetz, Koji Yatani, and Darren Edge. 2014. TurningPoint: Narrative-driven Presentation Planning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 1591–1594.
29. Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 715–725.
30. Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of Live-coding in Learning Introductory Programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18)*. ACM, New York, NY, USA, Article 13, 8 pages.
31. Scott Rosenberg. 2015. The Strange Appeal of Watching Coders Code. Backchannel: WIRED **https://www.wired.com/2015/08/the-strange-appeal-of-watching-coders-code/**. (2015).
32. Marc J. Rubin. 2013. The Effectiveness of Live-coding to Teach Introductory Programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 651–656.
33. Jeremy Warner and Philip J. Guo. 2017. 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)*. ACM, New York, NY, USA, 1136–1141.
34. Sarah Weir, Juho Kim, Krzysztof Z. Gajos, and Robert C. Miller. 2015. Learnersourcing Subgoal Labels for How-to Videos. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work &#38; Social Computing (CSCW '15)*. ACM, New York, NY, USA, 405–416.
35. Greg Wilson. 2018. How to Teach Programming (and Other Things): Live Coding. **http://third-bit.com/teaching/live.html**. (2018).