Necode: An Environment for In-Class Programming Activities

Major Qualifying Project

Advisor:

Charles Roberts

Written By:

Trevor Paley

A Major Qualifying Project
Worcester Polytechnic Institute

Submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science.

August 25, 2021 - March 4, 2022

Abstract

Running in-class programming activities is difficult with existing software. In this paper, we present Necode, a web-based, open-source environment for creating and delivering these kinds of activities. Necode's APIs enable developers to create new activities and programming challenges, and to integrate additional programming languages into the environment. We present some sample activities to showcase Necode, including one in which students solve a programming problem, run their code through instructor-defined automated tests, and submit it to the instructor so that different solutions can be compared and discussed as a whole class. To assess Necode, we asked students to recreate displayed shapes and patterns using GLSL and surveyed students about their experiences; feedback on the tool was generally very positive.

Table of Contents

List of Figures

Figure **Page**

Executive Summary

Writing code is an important part of learning computer science, and so it is valuable to have software that assists students write code in educational contexts. We investigated several software solutions for providing an environment for students to program in, but an area we found was lacking was software to facilitate in-class programming activities, a problem we created Necode to solve.

The software we investigated mostly focused on supporting tasks which students might do on their own or in groups for an assignment, but mostly not activities intended for completion during class time. However, there were still some valuable insights that these software provided. First, the tools we looked at with educational use cases in mind, such as CodeCircle[1] and Replit[2], are browser-based. Jupyter notebooks too, though not specifically intended for educational use, have a web client [3]. This is very convenient for both students and instructors since it means that no external software needs to be installed, reducing both the cost of dealing with unusual software configurations and the time it takes to set up. We took this approach for Necode as well. Additionally, they all had visual output of some kind, though it took different forms. Being able to visualize code, or the output of code, can aid understanding, and in some cases can introduce a level of creativity to coding which improves student engagement. Necode uses visual output in a way that most closely resembles Replit's output pane, though that could be changed depending on the type of activity.

With Necode, we made a general framework for instructors to create in-class activities, and for developers to create new kinds of in-class activities. We wanted to isolate programming languages from activities so that activities could automatically work with programming languages we added support for in the future, especially as the possible combinations of programming languages and activities grew multiplicatively. For this, we designed a concept of "features" in which activities declare which features they require and languages declare which features they support, and every activity will therefore have a guarantee that the languages it will be used with will support everything it needs to be able to do.

Our primary "DOM Programming" activity provides a code pane as well as, depending on its configuration, an HTML and CSS pane, which allows a student to create small web pages that will appear in the output pane. While the conventional language to use in the code pane would be JavaScript, due to the feature API, it can be used with any supported language (currently JavaScript, TypeScript, and Python 3). This works by compiling the various

languages to JavaScript before execution, using Babel for TypeScript and Brython for Python 3 [4].

For use in class, the intended way to use Necode, we have a submission system by which students can submit their code to the instructor and the instructor can load up student submissions and show them to the class, likely with discussion. If the instructor chooses, they can also include automated tests that run when a student tries to submit their work; these tests are implemented using a DSL inside TypeScript.

Necode also supports interaction between users in an activity via WebRTC, using an API we call "RTC policies." In order to showcase this, we created a "Canvas Ring" activity in which students can write code to draw on an HTML canvas, and then their canvas is sent to the next student who can draw the previous student's canvas onto their canvas, and so on, in a "ring" or circle formation. This activity fosters creative coding and collaboration, while also teaching about the HTML Canvas API.

In order to assess Necode in practice, we created a new activity that enables students to write GLSL shaders and see them rendered to the output pane. Despite some technical issues when configuring the activities, Necode worked very smoothly when we tested it on an actual class of students, and the feedback was very positive, with students especially liking the fast feedback from having the output pane integrated into the editor and the full-class discussion of student submissions.

In the future we plan to fix some bugs and interface issues with Necode as it currently exists, and then explore supporting languages like Racket and Java, which are often used in education but do not have easy ways to compile and run them in JavaScript. We also want to explore more possibilities with RTC policies, and use them to implement collaborative editors.

Necode's source is available at https://github.com/TheUnlocked/Necode, and an instance of Necode set up for WPI is available at https://code.cs.wpi.edu/.

Chapter 1

Introduction

Over the past few decades, computer science education has evolved. No longer are the days of punch cards and shared terminals—students can now write and execute code on demand using portable laptops that they carry around with them throughout the day. Despite these advances however, lectures have largely remained one-sided: the instructor explains the topic of the class, and they may even write and execute code in class. Yet student participation in the in-class code-writing process often remains restricted to more indirect forms of engagement like asking for student suggestions for what code to write, as if computing was still some scarce resource. While some amount of indirect participation may be useful, its dominance in contrast to direct participation is strange. In-class individual and group exercises are a staple of math and engineering courses; if we have the computing power for it, why are they not just as present in computer science too?

Necode attempts to help close this gap. Just as a calculus teacher might ask students to find the area under a surface, a computer science teacher might use Necode to ask students to find the unique elements in a list. At the start of the activity, they could introduce the problem to students and then have students open up their laptops and start trying to solve it. When a student thinks they have a solution, they could run it through automated tests written by the teacher to see if their code is correct (an advantage that computer science has over calculus), and when they get it, the student can submit their solution to the teacher. When the teacher feels enough time has passed, they can show student solutions to the class so that the whole class can discuss various students' approaches to the problem, just as a calculus teacher might call up student volunteers to explain their work. The benefits of this have been proven in math classes, we believe it is equally appropriate for computer science [5].

Computer science is broad, and the set of activities that a computer science teacher might want to do with their students is similarly broad. One teacher might want to have their students

build a small game, while another might want their students to navigate a binary-search tree, while yet another might want their students to implement specular highlights for a basic 3D renderer. While Necode cannot possibly cover the entire set of possible activities, we aim to cover a large swath of them by making Necode a framework for in-class computer science activities in general, rather than forcing all interactive coding exercises into a particular rigid format. While we spend most of our time in this paper on one type of activity that facilitates the kind of individual exercises we mentioned earlier, we will also present an alternative type of activity that focuses more on creative and collaborative elements of programming in order to better demonstrate Necode's full capabilities. These particular out-of-the-box activities are helpful for demonstrating Necode's features, but fundamentally Necode is the infrastructure that ties the activities together, not the activities themselves. Thus, our contribution is not just the end-user experiences that we will show screenshots of, but also the set of APIs which allow developers to use Necode to create new kinds of activities for instructors to use in their classes.

Chapter 2

Background

The concept of using interactivity and live coding to teach programming is not itself novel, and many different approaches have been taken to execute it. In this section, we will discuss a few of these approaches, and what each of them brings to the table before we dive into Necode itself.

2.1 Industry Collaboration Tools

Collaboration is commonplace in professional software development, and much software has been created to make it easier in an industry setting. If that software was effective in classrooms, it would greatly simplify the problem of classroom collaboration—using off-the-shelf software is much simpler than making new software, and if it is used in industry, it would also likely be more stable and come with better support.

As such, many instructors have tried using professional collaboration tools. In 2020, Ying and Boyer investigated the effectiveness of many of these tools in actual classrooms, such as Git and other version control for asynchronous collaboration, Visual Studio Live Share and other real-time collaborative editors, screen sharing and remote access tools, and more. They found that in general, these industry tools were often too complex for classroom use and that there was a need for more user-friendly interfaces for novice programmers. Rather than use highly-featured but more involved tools like Git and Live Share, they found that some students were using considerably underpowered tools like Google Docs and email, neither of which are ideal for collaborating on a programming assignment [6].

While it may be valuable to teach about industry collaboration tools in school, particularly when teaching about software engineering, they are not necessarily ideal in cases where there is a significant risk that they could distract from the intended material.

2.2 CodeCircle

Researchers have also explored creating new specialized software for classroom use. One example is CodeCircle,[1] created by Fiala, Yee-King, and Grierson at Goldsmiths, University of London, whose primary purpose is supporting social "creative" coding in a web browser, such that it can be used as a teaching tool in classrooms. In their paper describing CodeCircle, the authors point out six core features that they claim are desirable for interactive coding software [7].

2.2.1 Live Mode

"Live mode" is described in contrast to a form of interactive coding tool where some additional gesture is required to push the code into the live environment beyond just writing code (e.g. pressing a "refresh" button). With live mode, changes are automatically rendered as they are written, without any additional user input being required. The authors of CodeCircle claim that live mode increases interactivity by having lower feedback delay compared to systems that require a separate gesture.

2.2.2 Collaboration

Specifically live collaboration, or the ability to have a shared editor that multiple users can edit at the same time. This is a core feature of many collaborative programming systems (including some of the industry tools mentioned earlier) since it allows for multiple people to collaborate on a single piece of code at the same time.

2.2.3 Sharing & Forking

Sharing allows documents to be made public and shared with other users (in either an editable or read-only state) via a URL, while forking allows a user to duplicate a shared project and work on their own version separately from the original document. Subsequent research on CodeCircle by Yee-King, Grierson, and d'Inverno found that a "Fork and Customize" approach to teaching in which students would fork a program and then change it for their own personal tastes (as opposed to more traditional "fill-in-the-gaps" and "implement a specification" programming activities), was correlated with higher student creativity, as well as indirectly correlated with higher final grades (though the authors say more research is required on that topic) [8].

2.2.4 Code Validation

In order to improve developer experience and avoid simple run-time errors, the authors decided to validate code statically using JSHint, a JavaScript linter, and only push the user's code when

[1] In this section we are specifically focusing on the original version of CodeCircle published in 2016. Since then, CodeCircle has undergone several changes and been rebuilt as CodeCircle V2, but the core emphases remain the same [1].

it validates.

The authors also mention that code validation would provide security and privacy benefits, which could be true, for example if the validation restricted code to using only a known-safe subset of JavaScript,[2] but the authors do not mention any such restriction in CodeCircle.

2.2.5 Assets

Since CodeCircle was designed for creative coding of audiovisual environments, the ability for users to import assets (such as images, audio, etc.) into their program is useful, though it is unclear how necessary this feature is in other educational contexts. CodeCircle also provides a preview window for viewing these assets once uploaded.

2.3 Jupyter Notebooks

Jupyter notebooks, while most often used for data science applications, are also sometimes used as an educational tool. Rather than the more directly collaborative tools seen earlier, Jupyter notebooks allow a student to walk through a lesson with text, visual, and code components, almost like a set of interactive lecture notes. Because of this, they are potentially well suited for use in flipped classrooms [10].

There are several barriers to use of Jupyter notebooks in classrooms, however. Initial setup on its own can be cumbersome, either to the student if they must locally install and run the notebook server and client, or the instructor if they need to set up a Jupyter notebook server for the class. Additionally, Jupyter notebooks maintain a hidden state that is preserved when any code section is run, which creates an unintuitive disconnect between the layout of the code in the notebook and the actual state of the environment [3]. This has the potential to be extremely confusing to students, especially if their code works when they try it, but is reliant on transient state that would not exist on a restart (either due to the relevant code being deleted, modified, or being out of order from the order the code sections need to be run in for correct behavior).

Johnson from the University of New Hampshire also points out some pedagogical issues frequently associated with Jupyter notebooks, such as teaching poor software development practices. Automated testing of code cells is often not a focus, and neither are style conventions. In order to maximize the benefit of Jupyter notebooks, instructors need to be aware of the capabilities of Jupyter notebooks in order to properly utilize them for their instruction, which is an additional learning curve [3].

[2]For example, using Google Caja, a now-defunct JavaScript transpiler that sanitizes JavaScript at the cost of decreased capabilities and sometimes falsely rejecting safe code [9].

2.4 Integrated Commercial Products for Coding in Classrooms

Some companies have created products specifically to facilitate programming education, both in and out of class. Two examples of these products are Replit's Teams for Education[2] and CodingRooms [11]. While not identical, these products are similar, so we will focus our attention on Replit's Teams for Education in this section.

The main benefit that integrated classroom solutions like Teams for Education bring is the asymmetry between the instructor and the students. Unlike tools we have looked at before (other than some applications of Jupyter notebooks), in Teams for Education the students and instructor are not using the same end-user perspective but in different ways. Instead, the instructor has special control over their students and can interact with them directly, such as by observing their code as they write it and commenting on, or even editing, their code live in front of them. Additionally, instructors can assign students to groups and place them into collaborative editors directly, taking control out of the hands of students, but also streamlining the process [2].

In addition to greater instructor control over students, there are also a number of other useful features for educational contexts that products like Teams for Education provide.

2.4.1 Multi-lingual

The core Replit infrastructure that powers Teams for Education runs code on a virtual machine on a server rather than in the user's browser, which allows it to run code in practically any language, not just JavaScript [12].[3] This is significant, as only a tiny fraction of computer science classes use JavaScript—less than 1% of introductory classes in the US [16]. Supporting a wider variety of programming languages is necessary if an educational programming product is intended to be integrated into existing courses.

2.4.2 Automated Tests & Auto-Grading

Teams for Education also supports creating automated tests that students can run to verify that their solution to a particular problem is correct before submitting it (and which can also be used for automatically grading student submissions). These tests can either take the form of "Input/Output testing" in which some string is fed to "standard in" and then the resulting "standard out" is tested against an expected output, or traditional unit testing for the few languages which support it [2].

[3]Some languages like Java and other JVM languages, C# and other .NET languages, Racket, and so on can be compiled to WebAssembly, which can be run in a browser, but the compilation step itself can not usually be performed in a browser, at least not without significant effort and potentially untenable performance cost [13–15]. Some languages like Python 3 have compilers/interpreters written in JavaScript[4] which eases those concerns but are still more complicated to use than executing JavaScript directly.

2.4.3 Graphics Pane

Replit (and by extension Teams for Education) supports a graphics pane whose content is streamed from the virtual machine to a user's browser over a WebSocket,[17] or is rendered directly in the user's browser with an iframe if the user is doing an HTML/CSS/JS project. Like previously seen with CodeCircle, this enables a more visually interactive experience than simply implementing algorithms and running them against test cases.

2.4.4 Proprietary

A potential downside of these products is that they are closed-source, paid, proprietary software. In terms of cost, Replit's Teams for Education is $1000/year for an institution or $35/month if a teacher wants to purchase a license for themselves.[4] While that might be a considerable price to pay for an individual teacher, the expense for an entire institution is likely negligible in most cases. The bigger concern is that these are proprietary services, which means it is not possible for others to contribute to them or fork them with new features, and their use is entirely dependent on the faith that the company behind the software will stay alive and continue to provide the service.

[4] CodingRooms does not publicly advertise its price.

Chapter 3

Methodology

In this chapter, we will first look at the design philosophy guiding Necode during its development, after which we will dive deep into the primary type of activity in Necode, the DOM Programming Activity, which we will use to show off many of Necode's primary features. Afterward, we will look at a different kind of activity in order to demonstrate Necode's Real-Time Communication (RTC) capabilities, and finish off with an overview of our Classroom management tools for instructors.

3.1 Design Philosophy

Necode is intended to, at its core, be an extensible piece of educational programming infrastructure. However, the word "extensible" is vague, and over-eagerness on that front can lead to software that is hard to use, hard to write, and does not even achieve a useful level of extensibility in practice. While Necode is not a Unix application, the Unix philosophy (at least in part) still applies: "Make each program do one thing well" [18].

Therefore, Necode's extensibility only aims to reach to the extents of one narrow slice of educational programming software: active learning through social in-class programming activities. To achieve this goal, Necode's design takes an opinionated stance as to the existence of classrooms, as well as to the hierarchical nature of the student-teacher relationship. An in-class activity cannot occur without a classroom, and student participation cannot occur without students. The precise nature of the activity and student participation may vary, but those facts hold true no matter the case.

Note that since Necode does not try to do everything, it is *necessarily* less featured in certain areas than the software previously discussed in Chapter 2. Since all student work is performed in the context of in-class activities, student creations are not shareable like they

are in CodeCircle, nor can they be viewed independent of the activity they were created in. Similarly, because the activities are intended for in-class use, Necode does not support the assignment and auto-grading system of Replit's Teams for Education.[1] Finally, because Necode is focused on student participation, it is not suitable for asynchronous, independent work like Jupyter notebooks are. Instead, Necode tries to tackle its own niche, such that it can pair with other software (like those just mentioned) in an orthogonal manner.

3.2 The DOM Programming Activity

In order to demonstrate the features of Necode in action, we will take a look at one of the primary activities that we have implemented, the DOM Programming activity. This is not the only activity that exists, and we will discuss the general ActivityDescription API in more detail in Section 3.3, but this is the best place to start from to give an idea of what is currently available in Necode out of the box.

3.2.1 In Action

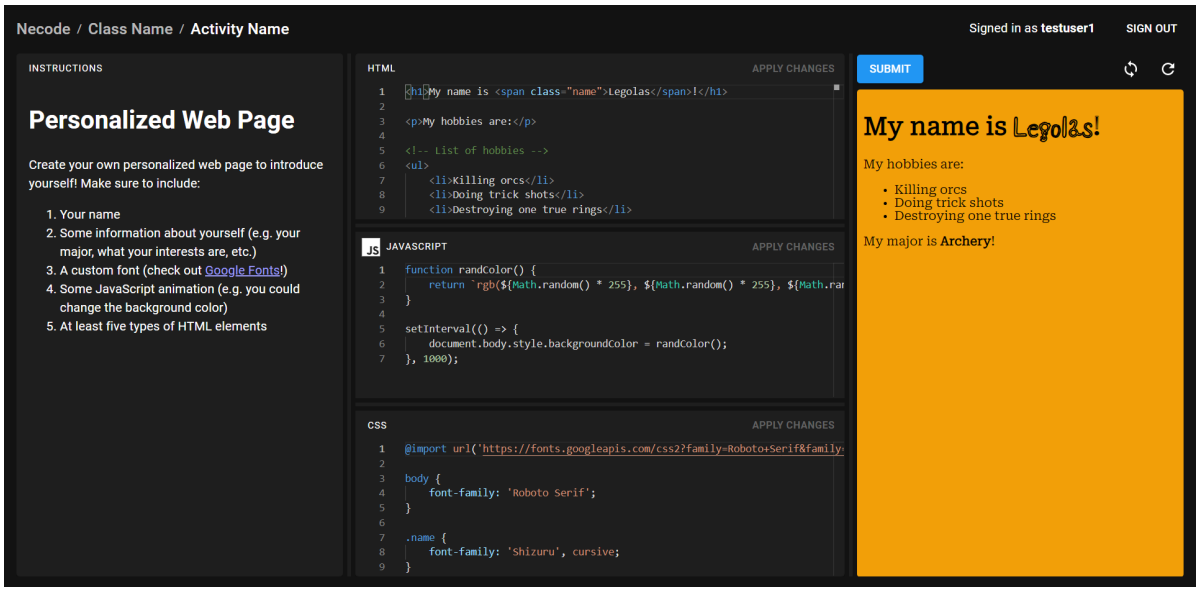


Figure 3.1: *A sample DOM Programming activity from a student's perspective*

When a student joins a classroom (Section 3.4) and a DOM Programming activity starts, they will see instructions, a set of HTML, JavaScript, and CSS input panes (these can be configured—see Section 3.2.3), and a visual output pane, as shown in Figure 3.1. All of these

[1] Though it does support a variant of this tailored for in-class activities.

panes are resizable using drag handles on the edges, and will change their layout responsively depending on the user's screen size.[2]

By default the input panes will contain some text configured by the instructor, but their contents can be changed, and the student can see the output pane update live by clicking "Apply Changes," or by pressing the Ctrl/Cmd+S (or "save") keyboard shortcut. Note that this is a contrast to CodeCircle in which changes are propagated instantly as soon as they are written (Section 2.2.1). One of the big reasons for this is that updating code as it's written can cause loss of transient state even when a student did not intend to commit their changes yet, which may be frustrating.

3.2.1.1 CSS Hot Reload

Programs can often have complex state that takes time to set up, and so when someone writes code, we would ideally want the ability for that code to be replaced live without resetting the entire program state. This is called "hot reload" (or "hot replacement," "hot swap," etc.), and can be a very desirable feature.

Unfortunately, hot reloading the JavaScript content is not possible (see Appendix A: Hot Reload), and hot reloading the HTML content is not possible for the same reasons. However, styling is usually segregated from behavior, so swapping in and out style sheets is something we are able to do. This can be very useful, as it allows students to experiment with different ways of styling their program while their program has meaningful state active, and we believe it will significantly improve the development experience with respect to writing styles. If, for whatever reason, changing the styling does break the program behavior (e.g. if the program reads information about an element's rendered size, and a student changes layout-related styles in the CSS pane), the student can still reload the entire environment with the refresh button in the top right.

3.2.1.2 Submissions

Once a student has created something they are satisfied with, they can press the "Submit" button to send their work to the instructor. When they do so, a pink indicator will appear on the instructor's screen to show them that a student has submitted (Figure 3.2). Then, by clicking the "View Submissions" button, the instructor can see everything that students have submitted. Note that in Figure 3.3, Frodo's submission is version 2. Students can submit as many times as they want, and all submissions are kept.[3]

[2]Due to the code editor we use, Monaco, not supporting touch devices (such as phones and tablets), Necode does not currently have support for them either. However, a responsive design helps for smaller desktop displays, and it means we will be ready if we add a backup code editor for touch devices or Monaco adds support.

[3]There is currently no interface for an instructor to access submissions other than the latest one for each student, though the API to do so does exist.

Figure 3.2: *The instructor's view after two students have made submissions*

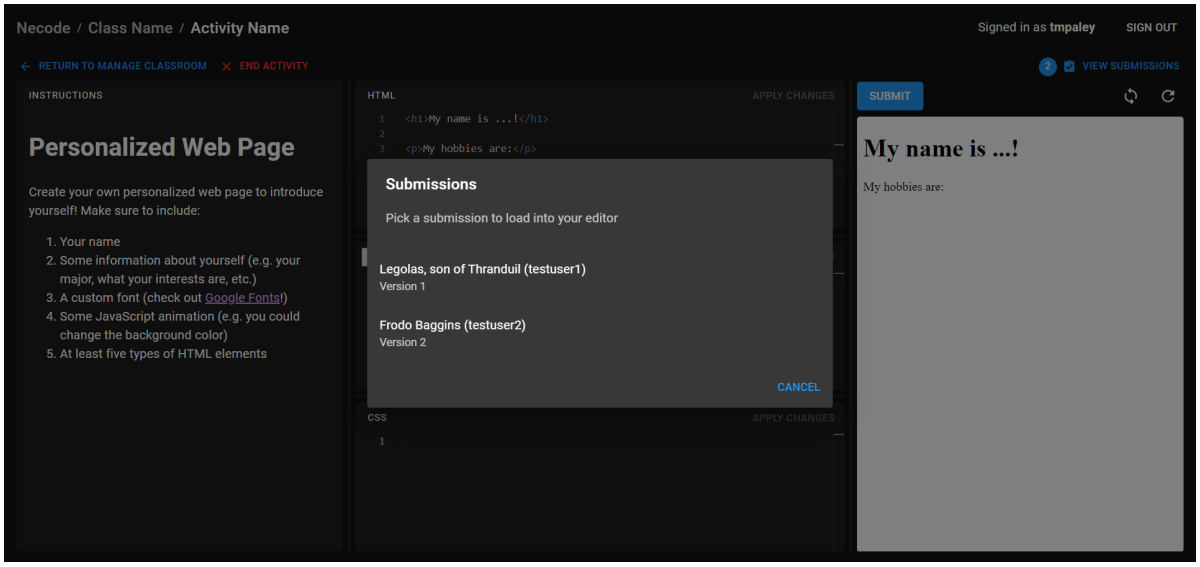


Figure 3.3: *The dialog shown after pressing the "View Submissions" button*

When an instructor selects a user in the submissions dialog, it loads their submission into the instructor's editor, though it does not apply those changes. The instructor can press the "Apply all changes" button in the top right (the one with the "sync" symbol), or apply the changes in each pane individually. This is partially a security consideration—we do not want to run untrusted code on an instructor's computer before they have the opportunity to look at it—though there could also be pedagogical benefits in some cases, for example if the instructor wants to examine a student's code as a class to see if students can figure out what it will do before actually running it.

3.2.2 Security

In order to improve interactivity and decrease load on servers, all code in Necode is run locally on the user's computer. However, whenever running code on your computer, especially when

that code is something that someone else may have written, ensuring that the code cannot do harm should be a top priority. While defending against malicious code (such as code trying to extract sensitive information from the victim) is important, we are also interested in mitigating the risk of unintentionally harmful code.

3.2.2.1 Sandboxing

Sandboxing is our main mitigation strategy for harmful code. By running all user code inside of an iframe with the sandbox attribute set to only have `allow-scripts` enabled, the browser blocks all attempts from the iframe to interact with the containing window, meaning attackers should not be able to access sensitive information from the user's Necode account by getting them to run code in the Necode editor panes.[4]

Not being able to directly interact with the iframe does present some technical challenges. For example, while the authors of CodeCircle describe placing user content directly into an iframe with `iframe.contentDocument.write`, this will not work in a sandboxed iframe since we are unable to directly access the iframe's content document [7]. We can however pass messages with `window.postMessage`. In order to send user data to the iframe, we load a script containing some scaffolding code in the iframe with the `srcdoc` attribute, and then after receiving an initialization message from the iframe, we send all of the user data in a message to the iframe, which loads it into the iframe's document. The reason we do not just load the user's code with `srcdoc` directly is so that we can have finer control for CSS hot reload (Section 3.2.1.1), and for loading in tests later on (Section 3.2.4).

3.2.2.2 Denial of Service

While it is impossible to defend a user against themselves if they really want to hinder their own experience, Necode can put in place some safety rails to prevent common kinds of self-inflicted experience degradation. The main kind we target in Necode is denial of service, when a user may inadvertently slow down or freeze Necode by running some kind of long-running or forever-running computation. Unfortunately, because sandboxed iframes still run code in the same thread as the surrounding window, they could freeze the user's entire tab with something as simple as `while (true) {}`.

To mitigate this issue, we use a babel transformer that adds iteration count checks to loops

[4] In reality, side-channel timing attacks such as the infamous Spectre vulnerability could still be used. While V8, the most popular JavaScript engine and the one used by Chromium, has some mitigations to this kind of attack, none of them are perfect, and the only truly reliable way to block such an attack is with process-level site isolation [19]. Unfortunately, while cross-origin iframes do run in a separate process, just being sandboxed is not enough for an iframe to be considered cross-origin, even if it does not have `allow-same-origin` in the sandbox attribute. There is a feature issue on the Chromium bug tracker to apply site isolation to sandboxed iframes, though this feature has not yet made it out of development at the time of writing [20].

and cuts them off after 10001 iterations.[5] While this is not a perfect solution to all types of infinite or near-infinite loops, it will hopefully eliminate the most common issues. An alternative possibility for more comprehensive self-DOS protection could be to inject a timer check before every statement (or even every expression) that resets based on the event loop, but the overhead for this could be quite large, and it still would not protect against single expressions that take a long period of time to execute, such as in regex denial of service attacks [22]. Ultimately, we believe that capping the iteration count of loops is a reasonably effective measure with comparably manageable overhead.

There is a risk that capping iteration count could be undesirable, and so an option to disable this behavior or increase the maximum iteration count could be useful. Currently that kind of configuration is not possible in Necode, though adding it could be a possibility for future work.

3.2.3 Configuration

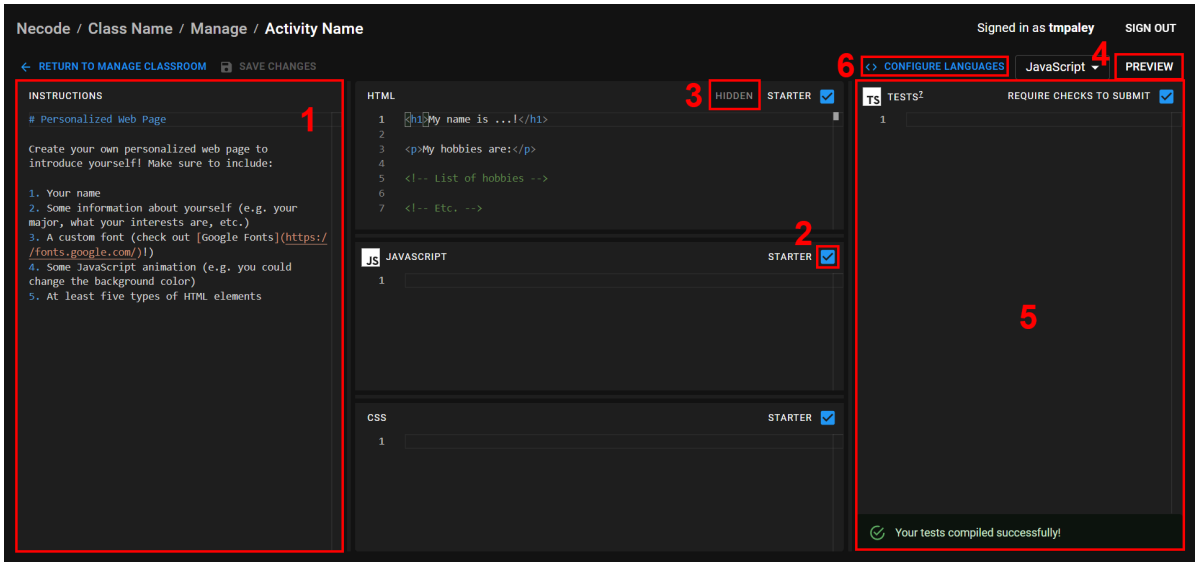


Figure 3.4: *The configuration view of a sample DOM Programming activity*

The configuration view (Figure 3.4) is similar in layout to the live activity view, which we believe creates an intuitive link between changes in the configuration and changes in the live activity. For example, the instructions pane (labeled 1 in the figure) is replaced with a markdown editor that allows an instructor to write the instructions. This editor supports most of Github-Flavored Markdown,[6] allowing instructors to embed images, tables, syntax-highlighted code blocks, and more to fit whatever their requirements are. Any changes here will be reflected in the rendered output in the student view.

[5] The code for this is adapted from a similar transformer made by CodeSandbox, which is in turn adapted from code in React, which was originally adapted from code in a 2017 blog post by Amjad Masad at Replit [21].

[6] Though HTML support is highly limited, even more than it is in GFM.

The three HTML, JavaScript, and CSS panes represent the starter code that will appear to a student when they load the activity. These can be edited to provide any amount (or no) starter code to students, as the instructor sees fit. If the instructor does not want to use a particular pane at all in their activity, they can also disable the pane by toggling the checkbox on the top right of the pane (labeled 2).[7]

Sometimes, an instructor may want to add their own HTML to whatever the student writes, either to provide scaffolding for the student's code or simply to move some clutter out of the student's view in order to simplify the activity. In order to facilitate this, we provide a "hidden HTML" tab (Figure 3.5), which can be accessed by clicking on the "Hidden" tab in the HTML pane (labeled 3 on Figure 3.4). If the instructor wants to embed the user's HTML somewhere inside the hidden HTML, they can do so with the `<user-content>` element.

An example of what the activity in Figure 3.5 would look like can be seen by pressing the "Preview" button (labeled 4 on Figure 3.4). This will load up a version of the live activity based on the current configuration. In Figure 3.6 we can see the effect of the hidden HTML on some user-inputted HTML. We can also see that only the HTML pane is visible since the checkboxes of the JavaScript and CSS panes were both disabled (the instructions pane is still available, but has been collapsed with the drag handles on the sides of each pane).

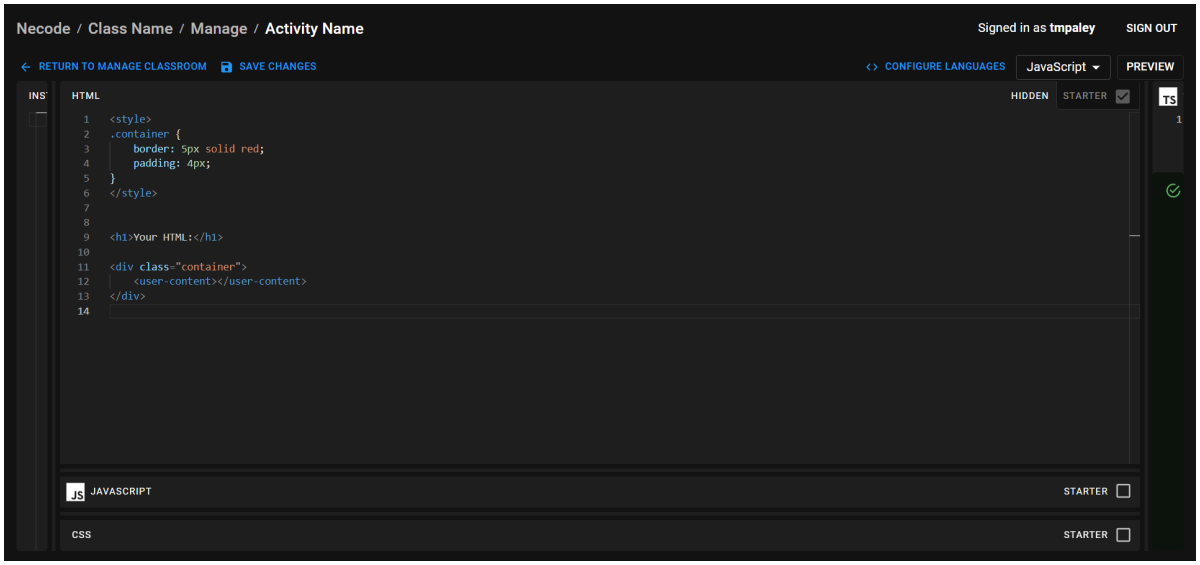


Figure 3.5: *Configuring an activity that uses hidden HTML*
Notice the "hidden" tab is selected in the top right corner of the HTML pane.

There are two other parts of configuration, the tests pane (labeled 5) and language configuration (labeled 6), that will be discussed in Section 3.2.4 and Section 3.2.5 respectively.

[7] Technically all three panes can be simultaneously disabled if the instructor does not want the student to write any code, but it would not make for a particularly interactive activity.

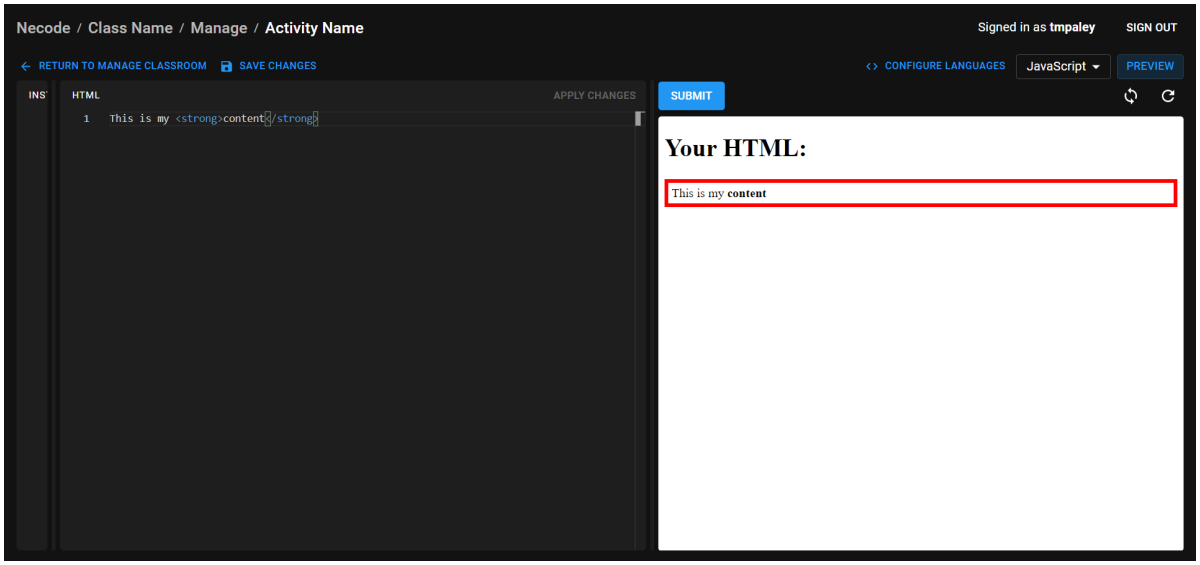


Figure 3.6: *Previewing an activity that uses hidden HTML "Your HTML" and the red box around the student's HTML content are caused by the hidden HTML.*

3.2.4 Tests

The DOM Programming activity allows the instructor to run automated tests before a student submits their code by writing a series of assertions in the tests configuration pane. Tests are written using a custom TypeScript DSL described in Appendix B: Testing DSL. The purpose of these tests is not for grading (since all code is run on the client, a student could inspect the code in the tests and over-fit their own code to the specific test cases), but instead to provide checks and guidance to students as they are writing their code. If the instructor wants students to be able to submit code even without passing all of their tests, they can also uncheck the "Require Checks to Submit" button in the top right of the test configuration pane (Figure 3.7); this enables instructors to view partially completed work and help debug student work.

Tests run in the global context and so they can be used for testing many things. They can be used to test specified behavior in the DOM, like if an assignment says that a button should increment some counter every time it is clicked, but more likely tests will be used for verifying some kind of algorithm implementation. That alone is not particularly novel, but when paired with the output pane, a student can visually experiment with their code and sample inputs before running tests on them. For example, Figure 3.8 shows a linked list activity in which the output pane is being used to help students visualize the data structure as they implement an algorithm.

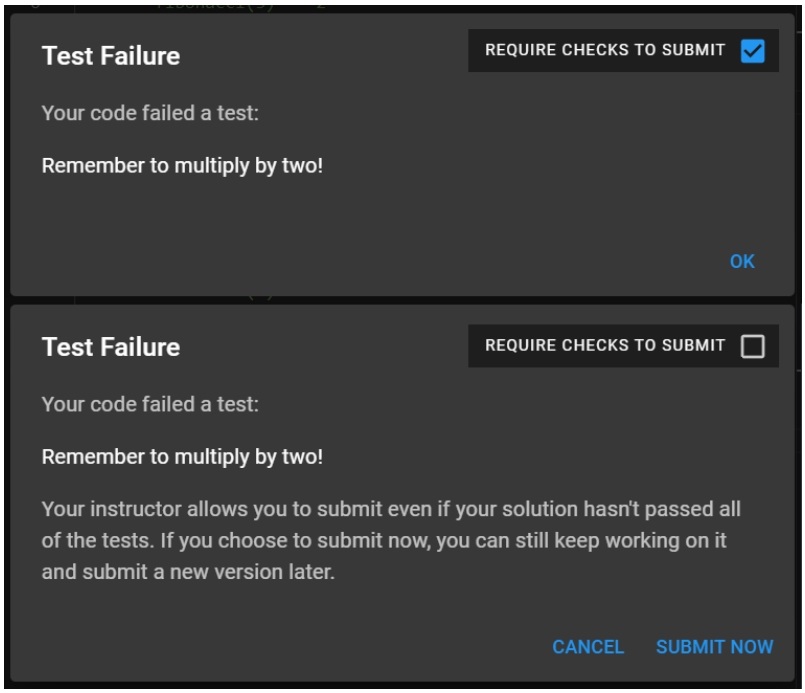


Figure 3.7: *A comparison of the test failure dialog depending on whether checks are required to submit*

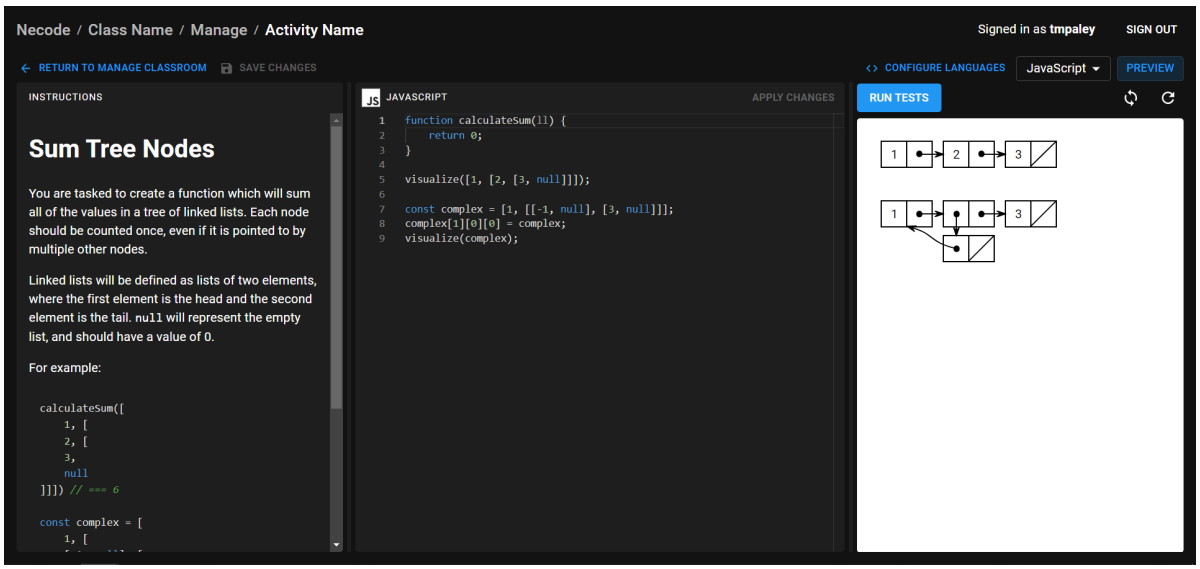


Figure 3.8: *An example activity in which students are asked to implement a linked list algorithm and shown a visualization in the output pane*

3.2.5 Languages and Features

This is the point at which we will begin to shift away from this specific activity and start to talk about how Necode really works behind the scenes. Throughout this section, we have referred to the DOM Programming activity as having HTML, JavaScript, and CSS panes. In truth, it does not. Instead, the JavaScript pane is really a generic code pane. While JavaScript is the default language, any language can be used via the LanguageDescription API (Figure 3.9), assuming these two requirements are satisfied:

1. The language can be either compiled to or interpreted in JavaScript.
2. The language's implementation in JavaScript has the necessary features (Figure 3.10), in this case, `supports:global` and `supports:isolated`.

```
interface LanguageDescription {
    name: string;
    monacoName: string;
    displayName: string;
    features: FeatureDescription[];
}
```

Figure 3.9: *A simplified version of the LanguageDescription interface*

```
interface FeatureDescription {
    name: string;
}
```

Figure 3.10: *A simplified version of the FeatureDescription interface*

`supports:global` means that declarations (such as functions and variables) in the language can exist in the global environment, and `supports:isolated` means that the language can be run in a sandboxed iframe or Web Worker, and does not require access to browser APIs. Note the word "can"; it is entirely possible for a language to support both being run in the global environment and in a non-global environment. "Supports" features indicate that the language is able to opt into the desired behavior upon request, not necessarily that it always has that behavior.

Currently in Necode, three languages have both `supports:global` and `supports:isolated`: JavaScript, TypeScript, and Python 3 (via the Brython library). We can see this when we go to configure languages for a DOM Programming activity (Figure 3.11). In theory, students would be able to opt into using any of the enabled languages, but currently Necode uses the first enabled language in the list. Therefore, it is best for an instructor to disable all of the languages they do not plan to use in an activity, leaving only the language that they want to support.

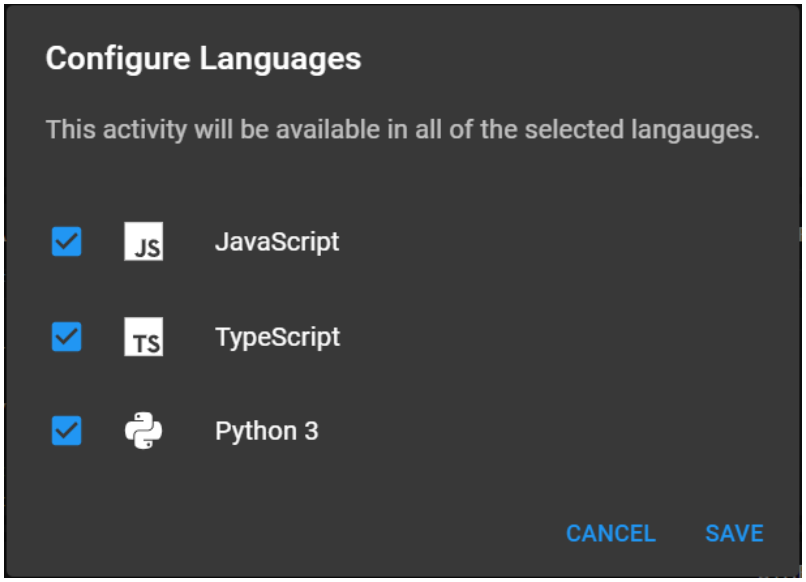


Figure 3.11: *The dialog for configuring the programming language for an activity*

Requiring the `supports:global` and `supports:isolated` features are not "hard-coded" into Necode, but are rather configured as part of the ActivityDescription API that governs all activities (Figure 3.12). The DOM Programming activity tells Necode that it requires `supports:global` and `supports:isolated`, and then Necode searches for all languages which include both of those features. In this way, languages and activities can be implemented independently of each other and will be automatically linked up whenever possible.

```
interface ActivityDescription {
    id: string;
    displayName: string;
    requiredFeatures: FeatureDescription[];
}
```

Figure 3.12: *A trimmed version of the ActivityDescription interface demonstrating the `requiredFeatures` field*

3.2.5.1 Compilation

There are a number of languages that can be compiled to either JavaScript or WebAssembly (which can be run in JavaScript), but whose compilers are not written in JavaScript or WebAssembly. This presents a dilemma where the language could theoretically be used in Necode if compiled, but cannot be in practice because there is no mechanism to compile it in the browser.

There are a couple of possible solutions to this. One is the solution used by Asano and

Kagawa in their 2019 paper that involved running user-written Java programs in the browser[13]. In their paper, they send the user's code to the server for compilation, then they convert it to JavaScript with the assistance of TeaVM, and then they send the compiled JavaScript back to the user's browser for execution. This is a model that could work for any language that supports compilation to JavaScript or WASM, though it places a dependency on the server for code execution, which could harm feedback time and strain the server's resources.

An alternative possibility could be to compile the compiler itself to JavaScript or WASM one time, possibly using itself if the compiler is hosted in the target language, or using a tool like Emscripten if not [23]. However, compilers can be quite large, and sending the entire compiler to the browser could cause significantly more strain than just compiling server-side and sending the output to the browser. More research on this topic would be required to make a determination about the impacts of both options.

3.2.6 Variations

Once again, things are not quite as they appear. The DOM Programming activity is really just the default mode of the broader HtmlTestActivity infrastructure. While DOM Programming provides a wide set of features and configuration options, instructors may sometimes want a more tightly tailored experience for what they are trying to teach. Rather than completely re-implement a new activity from scratch, it can often be easier to just create a new configuration of HTMLTestActivity.[8]

```
interface HtmlTestActivityOptions {
    hasTests?: boolean;
    hasHtml?: boolean;
    hasCss?: boolean;
    hasCode?: boolean;
    hiddenHtml?
        : { configurable: true }
        | { configurable: false, value?: string };

    typeDeclarations?: string | URLString[];
}
```

Figure 3.13: *The configuration interface for HtmlTestActivity*

[8]Keep in mind that while each instance of HtmlTestActivity is a type of activity, the ActivityDescription API itself has no knowledge of HtmlTestActivity, and HtmlTestActivity is simply a factory that creates activities implementing ActivityDescription.

3.2.6.1 p5.js

There was a request during the creation of Necode for a p5.js [24] activity. While very similar to the standard HTML/Code/CSS activity, this would be streamlined for p5.js, and could include p5.js-specific features. For example, because p5.js is such a large library, it would be valuable to provide type declarations[9] to Monaco so that auto-completion and tooltips would be available to students.

This led to the creation of the `typeDeclarations` customization field in `HtmlTestActivityOptions` (Figure 3.13), which allows type declarations to be provided either in one big source string or as an array of URLs to fetch the type declarations from. For the p5.js activity, we just pull the latest p5.js type declarations from the DefinitelyTyped repository.[10]

```
const [activityPage, configPage] = createTestActivityPages({
    hasCss: false,
    hasHtml: false,
    hasTests: false,
    hiddenHtml: {
        configurable: true
    },
    typeDeclarations: [
        'global.d.ts',
        'index.d.ts',
        'src/color/creating_reading.d.ts',
        'src/color/setting.d.ts',
        // ...
        'constants.d.ts',
    ].map(x => `https://raw.githubusercontent.com/.../types/p5/${x}`)
});
```

Figure 3.14: *A trimmed excerpt from `src/activities/p5js/index.ts`*

Because user-written code runs globally, p5.js works out of the box when the library is imported with a `<script>` tag in the hidden HTML,[11] and so students can use p5.js the exact same way they might on their local machine. Just like with the HTML/Code/CSS activity, in

[9] Specifically TypeScript (.d.ts) type declarations.

[10] https://github.com/DefinitelyTyped/DefinitelyTyped

[11] Actually, while Necode makes it easy for the instructor-user, importing scripts with a `<script>` tag is surprisingly hard. Trying to create a script tag by assigning to `innerHTML` causes the scripts to not run because of browser-enforced injection attack mitigation (removing the element from the document and putting it back also does not alleviate this issue). To get around this, after assigning `innerHTML` inside our iframe, we create new script elements with the exact same content and attributes for every non-running script element we just inserted with `innerHTML`, and then replace the non-running elements with those fresh elements using safe DOM APIs.

the p5.js activity students can modify some starter template provided to them in class, and then submit it to the teacher to show the class.

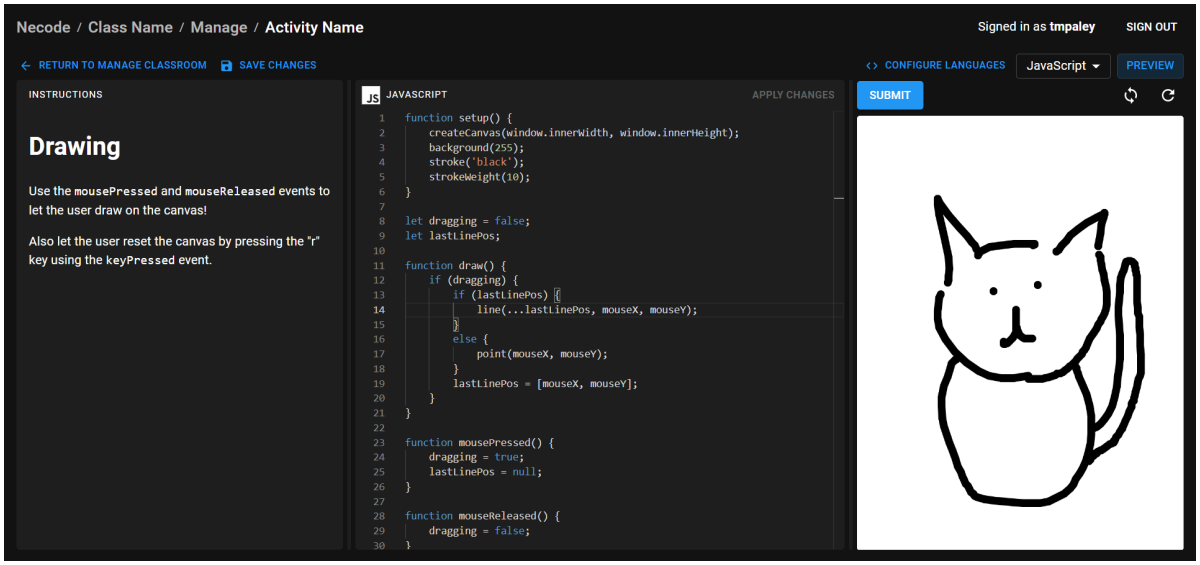


Figure 3.15: *A sample p5.js activity*

3.2.6.2 GLSL

To use Necode in an upper-level graphics class, we needed the ability for students to write GLSL shaders and view their output. While this could be done with the standard DOM Programming activity, we felt it would be far more intuitive if we instead had a single GLSL pane which students could write their shader in, without needing to worry about the surrounding WebGL infrastructure written in HTML and JavaScript.

The problem is that GLSL does not exactly support being run in the global JavaScript context, which means we could not approach GLSL integration in the same way we had approached something like Python 3 integration. Instead, we made the GLSL language provide an object to the global JavaScript scope representing the GLSL input. Additionally, instead of targeting the `supports:global` and `supports:isolated` features, we used a new "is" feature type, `is:glsl`. This allows activities to target specific languages by their name (every language provides the `is:language_name` feature by default). In this way, the GLSL language can be provided specially for the GLSL activity using the same FeatureDescription API as usual, but without interfering with activities like DOM Programming and p5.js.

3.3 An Activity With Real-Time Communication

In this section, we will look at a different, more experimental type of activity called Canvas Ring. Unlike the HtmlTestActivity family of activities, this activity is not about individual

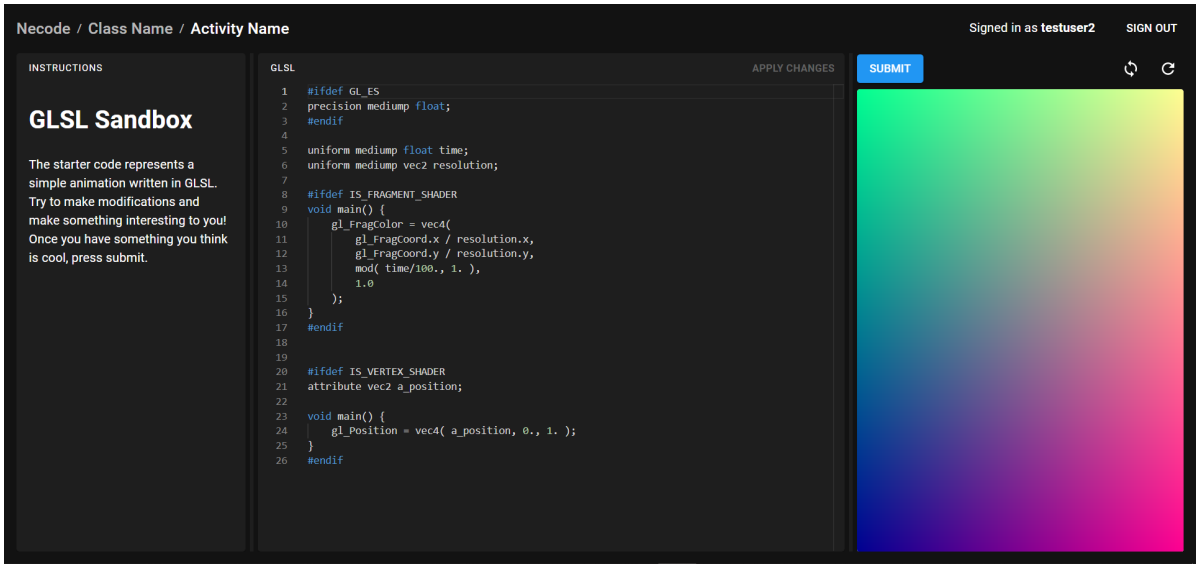


Figure 3.16: *A sample GLSL activity*

work but is instead about students interacting with each other in real-time through a visual medium.

3.3.1 In Action

The concept behind the Canvas Ring activity is that every user (students and instructors) who joins the activity is placed into a "ring" data structure.[12] Each user can then draw on a local canvas using the normal CanvasRenderingContext2D API in JavaScript. Every "frame" (10 times per second), the user's local draw method will be invoked, and then their canvas will be sent to the user following them in the ring. At the same time, the user will receive the canvas of the user preceding them in the ring, which they can use to draw their next frame. In this way, there is a circular feedback loop between all of the users, each making their own contribution to the canvas and then passing it onto the next user, in a large circle.

Though we have not yet had the opportunity to use this activity with a class, the aim is to foster collaboration and creativity between students, while also giving an opportunity to explore parts of the CanvasRenderingContext2D API that students may not frequently interact with, such as blending modes and the alpha channel. Since canvases are sent as video data over WebRTC, the activity can also demonstrate the effects of lossy compression over large numbers of compression cycles.

[12] In computer science nomenclature, a circular doubly-linked list. We like the term ring here for the image it evokes of a number of people linked together in a circle, so we will continue to use it. "Ring" as it is used in this paper has nothing to do with the circular buffer data structure, which is sometimes also called a ring buffer.

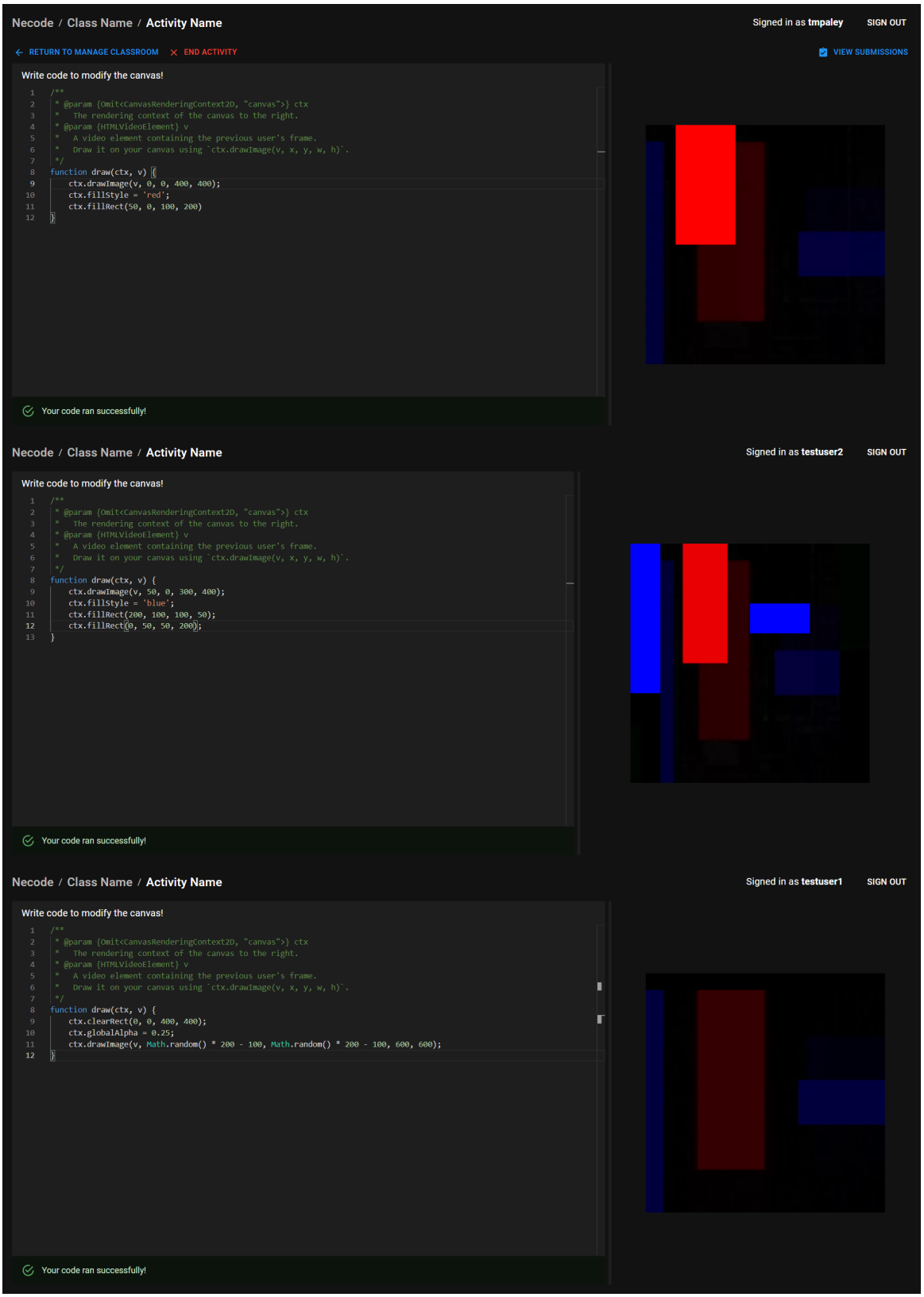


Figure 3.17: *An instructor and two students doing the canvas ring activity*

3.3.2 Guardrails

Unlike in the HtmlTestActivity family of activities, Canvas Ring is much more about live interaction, and so code is executed as it's written, rather than after receiving some kind of gesture. Additionally, code is *not* sandboxed in an iframe due to technical complications from interacting with the canvas and WebRTC from inside an isolated environment. For this reason, a number of guardrails are put in place to protect the user from harming themselves.

Firstly, any syntax error or other fatal issue will halt execution of the code completely. The details of this error are given to the user in an alert box at the bottom of the code editor. However, syntax errors are actually one of the less insidious kinds of bugs, since they just cause everything to stop. More dangerous are bugs involving assignment, in which the user might reassign an important property on the canvas rendering context object. If they do so by accident, recovering that function is hard. For that reason, all built-in functions the on rendering context object have had assignment disabled, and throw an error if the user attempts to reassign them. For example, if a user attempts to reassign the `fillText` method, the code will fail to run and they will see an error message:

> Error: For your own safety, reassigning the methods of `ctx` is forbidden. If you really want to reassign `ctx.fillText`, look into `Object.defineProperty`.

During development, we found that sometimes we would accidentally assign `ctx.stroke` and `ctx.fill` (which are methods) instead of the intended properties, `ctx.strokeStyle` and `ctx.fillStyle`. Because of that, if a user attempts to reassign one of these, we present a special message:

> Error: For your own safety, reassigning the methods of `ctx` is forbidden. If you really want to reassign `ctx.fill`, look into `Object.defineProperty`.
>
> Did you mean `ctx.fillStyle`?

Lastly, directly messing with the canvas element itself could interfere with the activity's functionality if the user does not know what they are doing. If the user attempts to access the canvas element via `ctx.canvas`, we once again stop them, but let them know how they can access the element if they really want to:

> Error: For your own safety, `ctx.canvas` is forbidden. If you really want to access the `HTMLCanvasElement` object, it has id `"canvas-activity-canvas"`.

3.3.3 RTC Policies

Previously in Section 3.2.5 we showed a trimmed version of the ActivityDescription interface (Figure 3.12). The actual ActivityDescription interface is far larger—after all, just knowing

the name and required feature set of an activity is nowhere near enough information to make the activity actually run. A more comprehensive illustration of the interface is provided in Figure 3.18.

```
interface ActivityDescription {
    id: string;
    displayName: string;
    requiredFeatures: FeatureDescription[];
    defaultConfig: any;
    rtcPolicy?: string;
    configWidget?: ComponentType<ActivityConfigWidgetProps>;
    configPage?: ComponentType<ActivityConfigPageProps>;
    activityPage: ComponentType<ActivityPageProps>;
}
```

Figure 3.18: *A simplified version of the ActivityDescription interface*

There are a few things to note here. First is the `activityPage` and `configPage` properties, which are used to provide the elements to React to handle displaying the activity page and configuration page. Note as well that the `configPage` property is optional. While the HtmlTestActivity activities have configuration pages, Canvas Ring does not. There is also a `configWidget` property, which relates to how the activity is displayed in the classroom management view (Section 3.4). And there is a `defaultConfig` property, which allows configuration of what an activity's configuration should look like when it is first created.

But most relevant to this section is the `rtcPolicy` property. If we take a look at the Canvas Ring activity's definition (Figure 3.19), we can see that its RTC policy is `'ring'`, which makes sense given how the activity behaves. This policy name is associated with an RtcPolicy (Figure 3.20) implementation of the same name on a server is responsible for WebRTC signaling, as well as for notifying users about activities starting and sending and receiving submissions. By specifying the desired RTC policy in the ActivityDescription, it instructs the server how to coordinate WebRTC connections between users.

The purpose of this abstraction is similar to the purpose of the FeatureDescription abstraction: to make activities more declarative and flexible. If we wanted another activity like Canvas Ring, but where instead the instructor's canvas was sent to all of the students, we could just implement a new RTC policy, called "command" for example, and then we would just have to change the RTC policy in the activity's ActivityDescription to be "command."

3.3.4 Entry Point

Something that was ignored in the previous section, but is important not to gloss over, is the presence of the `supportsEntryPoint`—or rather `supports:entryPoint`—feature in Canvas

```
const canvasActivityDescription = activityDescription({
    id: 'core/canvas-ring',
    displayName: 'Canvas Ring',
    requiredFeatures: [
        supportsEntryPoint
    ],
    activityPage: CanvasActivity,
    rtcPolicy: 'ring',
    defaultConfig: undefined
});
```

Figure 3.19: *A simplified version of the Canvas Ring ActivityDescription*

```
interface RtcPolicy {
    policyId: string;
    new(
        users: Iterable<string>,
        settings: RtcPolicySettings
    ): RtcCoordinator;
}

interface RtcCoordinator {
    onUserJoin(user: string): void;
    onUserLeave(user: string): void;
}
```

Figure 3.20: *The RtcPolicy and RtcCoordinator interfaces*

Ring's ActivityDescription required features list (Figure 3.19). This is an alternative form of execution from `supports:global` in which an activity can specify an entry point (i.e. a function) that it wants to invoke, and the language implementation must create a variable called `entry` which that function is assigned to. In this case, the entry point is called `draw`, meaning the language must supply the activity with a function in the user's code called `draw`. This is what allows the activity to control the execution of the user's code, rather than the user's code controlling execution like was the case throughout Section 3.2.

3.3.5 Pending Issues

A major issue currently blocking the use of this activity in a classroom is reliability. When attempting to transfer data over WebRTC across networks, or between Chromium-based browsers and Firefox, we encountered issues where WebRTC would refuse to connect, or connect but refuse to transmit data. A hypothesis for what could be causing this is the absence of a

TURN server, but we have not yet had the opportunity to test this. We also faced issues with Chromium blocking autoplay of the video element used to pull in RTC data. While we have fixed it for Chromium, that could be contributing to our issues with Firefox, though the precise nature of the problem and potential remedies are still unclear.

3.3.5.1 Alternative Ways to Transmit Video Data

An alternative method for sending video data was considered during the development of this activity, in which instead of sending video from user to user directly, we could instead send each user's code to everyone else, and let each user compute the entire ring on their local machine. This would greatly reduce the necessary bandwidth for users to communicate since code is far more compact than video data, and it could also eliminate a number of technical issues encountered with trying to send video data via WebRTC. However, this was ultimately decided against. As mentioned earlier, we do not sandbox code in this activity, removing the strongest protections in Section 3.2.2, and even if we did perform sandboxing, unintentional denial of service would be an even bigger issue than it was previously—one student accidentally running some exponentially recursive code could take down the entire class' browsers.

3.4 Classrooms

So far we have been exclusively looking at activities, but tying all of the activities together (that is, the specific configured instances) is classrooms. The flow of setting up a classroom is fairly straightforward: the instructor creates a classroom at a special admin page, they get a join code for their classroom, they send the join code to their students, their students use the join code to join the classroom.

When an instructor goes to manage their classroom, they are presented with a calendar which they can use to select a particular day that they want to make a lesson for. After selecting a day, they can name that day's lesson, add activities, and even add short notes, either in code or normal text. All together, a configured lesson might look like Figure 3.21.

3.4.1 Widgets

Each of the components in the lesson plan is called a "widget," and widgets are directly tied to activities via the ActivityDescription API (Figure 3.18). In fact, a widget cannot exist without an associated activity. Even the text widgets have an activity, that activity is just a no-op (Figure 3.22–3.23).

The theory behind custom widgets is that they should be used either when they are being used by the instructor to convey some information to themselves (such as with the text input widget), or when there are only one or two configuration options for an activity and having a

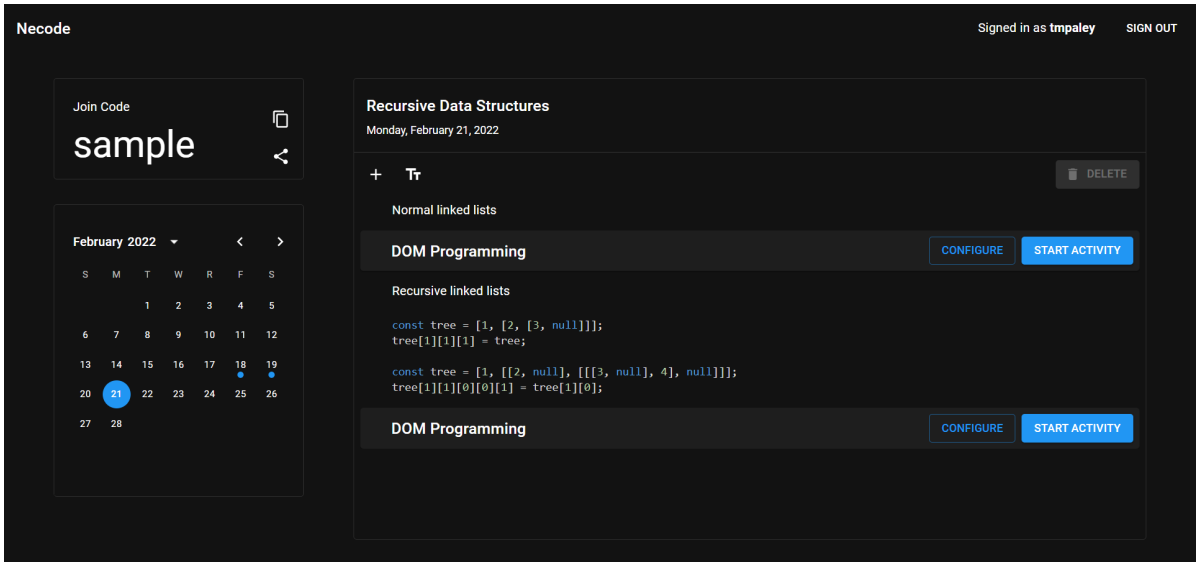


Figure 3.21: *A sample lesson in the manage classroom page*

```
const textInputActivityDescription = activityDescription({
    id: 'core/noop/text-field',
    displayName:
        'If this text is showing, you have encountered a bug :)',
    requiredFeatures: [],
    defaultConfig: {
        language: null,
        value: 'Write lesson notes here...'
    },
    activityPage: NoopActivity,
    configWidget: TextInputWidget
});
```

Figure 3.22: *A simplified version of the ActivityDescription for text widgets*

```
import NotFoundPage from "../../../pages/404";

export default function NoopActivity() {
    return <NotFoundPage />;
}
```

Figure 3.23: *The entire NoopActivity implementation*

full configuration page would be unnecessary. Otherwise, one should just use the default widget, which automatically populates with the necessary buttons to start or configure an activity.

3.4.2 Running Activities

Running an activity is as simple as pressing the "Start Activity" button, which will send all students to the activity automatically. Sending special links for different activities to students is unnecessary—students just need to log in and wait for it to start. Similarly, when an activity ends by the instructor pressing the "End Activity" button, it will also automatically close for all of the students.

The instructor is not required to be present in an activity for it to be running. As long as the instructor has not ended an activity (and the server is still running), the activity will stay living, which means that if, for example, the instructor loses internet access or has their computer crash, the activity will not be interrupted. When the instructor returns to the classroom, there will be a large banner letting them know that an activity is live and that they can return to it or end it (Figure 3.24).

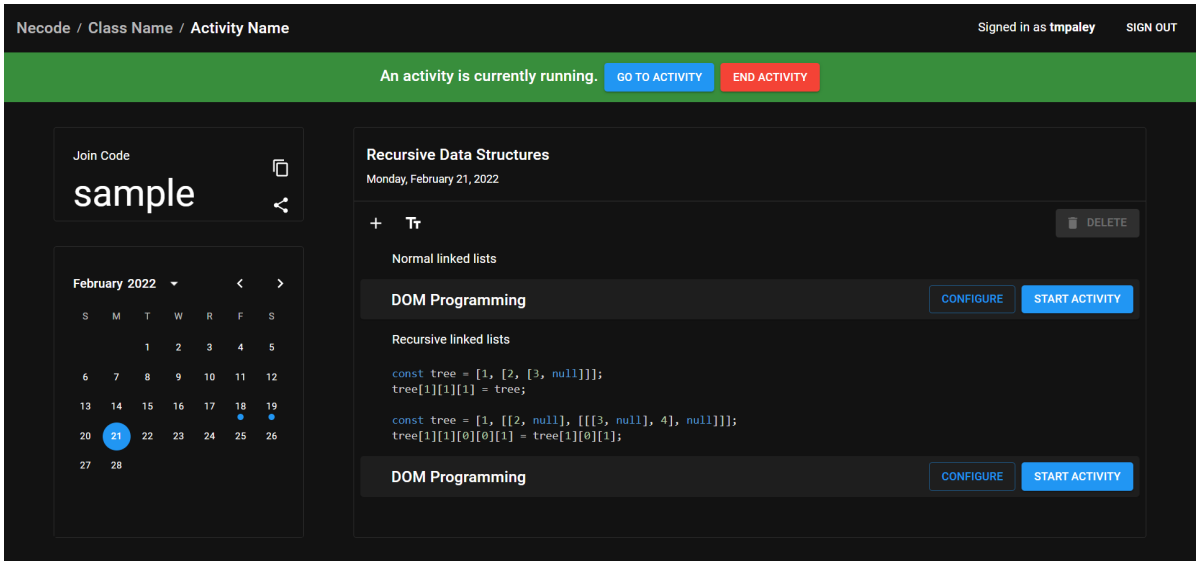


Figure 3.24: *The manage classroom page with the live activity banner visible*

Chapter 4

Results

To evaluate Necode in the classroom, we ran a series of activities relating to writing GLSL in a class with thirteen students and then asked them to fill out a survey about their experience (Appendix D: Necode Survey). These students had prior experience with GLSL from earlier in the term, though it had been a few weeks since they had done the kinds of exercises presented to them. Each of the activities was configured with a target output which represented what the student's result was intended to look like, along with the actual output of their shader (Figure 4.1).

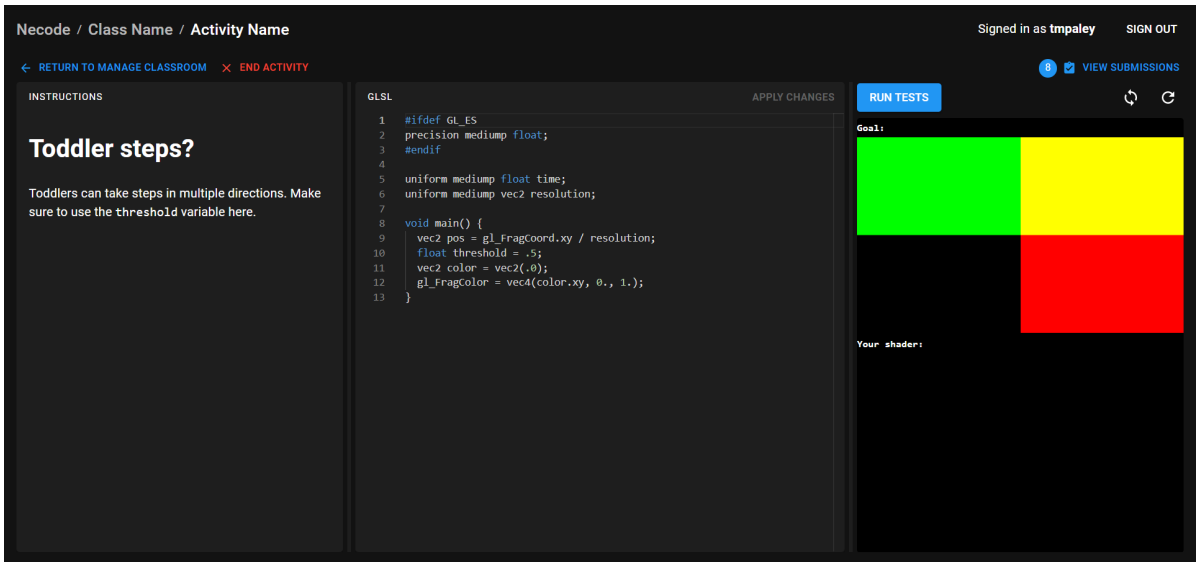


Figure 4.1: *One of the activities used in our study*

4.1 Instructor Feedback

The execution of these activities in the classroom was flawless, though there were some quirks with the instructor configuration that made creating the activities more challenging. These were primarily bugs and not-yet-implemented features whose individual impact was relatively minor but whose combined impact was more significant.

There were also some requests for new previously-unplanned features. One of these was duplication of activities. All five activities we ran had very similar infrastructure, so in order to make a new activity, the contents of each code pane had to be copy-and-pasted from one activity to the next. While doing that is not particularly hard, it provides a poor user experience that could be easily remedied with a new button.

The other major request we had was the ability for an instructor to create "dummy" accounts which they could use to test out their activity from a student perspective. Even if we tell instructors how Necode will work with students, allowing them to test the entire student flow for themselves could provide the confidence needed to actually use Necode in their classroom.

4.2 Student Feedback

Ten of the thirteen students who participated in some form filled out our survey. This is not a large enough sample to make a statistical claim with any kind of confidence, but it is enough to provide meaningful feedback on using Necode in a real classroom.

4.2.1 Necode's Interface

While enjoyment is not the primary metric of interest when evaluating this tool, it would be a bad sign if using Necode made classes less enjoyable, so we wanted to assess the students' feelings on the matter. Nine of the ten students said Necode made the class more enjoyable, and the remaining one said it did not impact their enjoyment at all (Figure 4.2).

In terms of Necode's usability, students were overall quite pleased once again (Figure 4.3), though a couple of students reported some initial confusion over needing to explicitly apply changes. One student reported that they would have liked a small reference of GLSL functions.

We also wanted to see how much of an issue it would be for students to jump between the programming environment they were used to and Necode. All ten students reported that it was easy to transition to Necode from their normal programming environment, with six of them saying it was very easy (Figure 4.4).

4.2.2 Individual Aspects

In addition to asking about their experience with Necode as a whole, we also asked students a series of questions about specific features of Necode, and how much they improved the student's

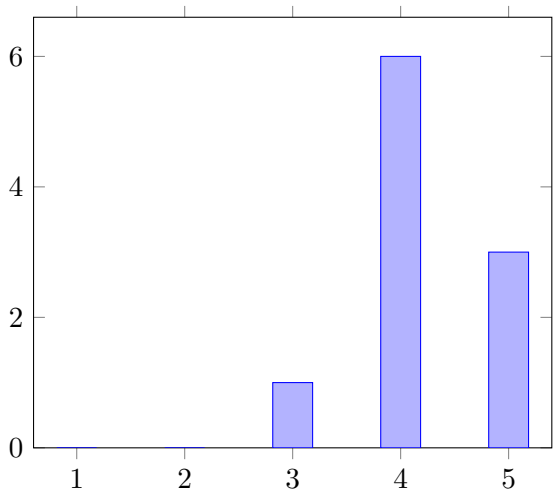


Figure 4.2: *Chart of responses on how enjoyable it was to use Necode*
Based on the Likert scale where 1 represents "decreased enjoyment" and 5 represents "increased enjoyment"

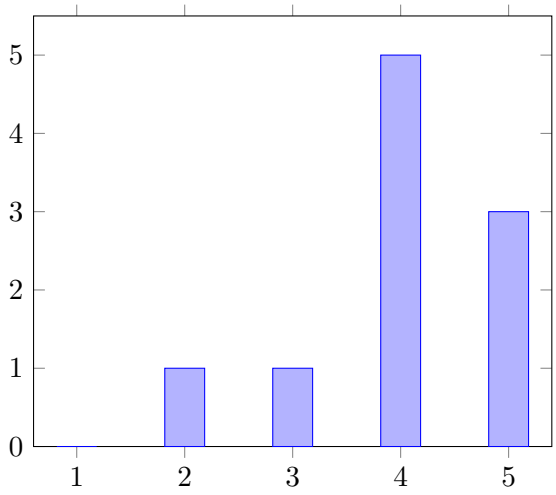


Figure 4.3: *Chart of responses on how easy it was to use Necode*
Based on the Likert scale where 1 represents "very difficult" and 5 represents "very easy"

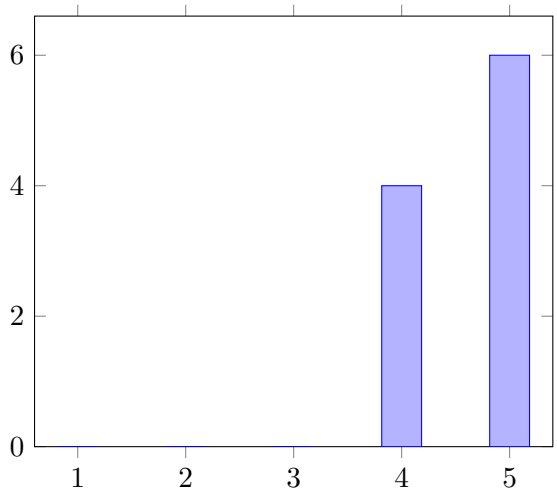


Figure 4.4: *Chart of responses on how easy it was to transition to Necode*
Based on the Likert scale where 1 represents "very difficult" and 5 represents "very easy"

understanding of the content being presented.

Having a code editor and output pane on the same screen was found to be extraordinarily helpful, with all ten students saying it helped them understand the content, and nine of them saying it helped a lot (Figure 4.5). A couple of students specifically mentioned that they found the ability to rapidly make changes useful.

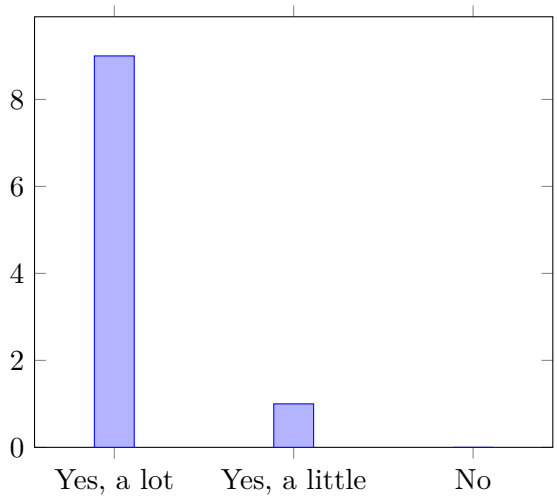


Figure 4.5: *Chart of responses on whether the code editor/output pane combination improved understanding*

The automated testing had much more of a mixed response, with only three of the students saying it helped a lot and an equal number saying they did not help at all (4.6). Referencing the goal image in the output pane, one student said, "[the] tests didn't really impact my understanding since it was more of a 'I can see that it's right, this is a formality.' " Another

mentioned that the tests were unable to catch them using an if statement rather than a GLSL function, which while not behaviorally wrong, has significant performance impacts in GPU code.

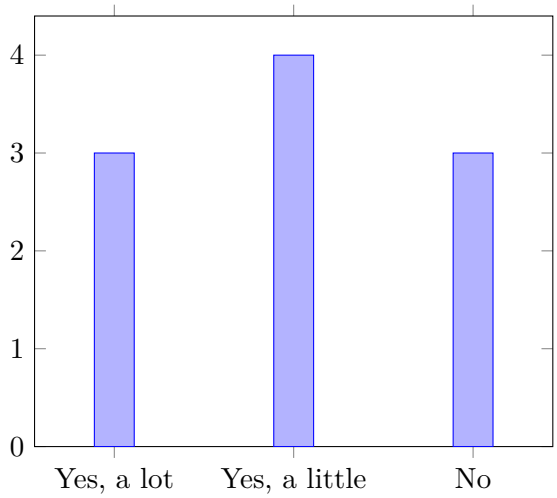


Figure 4.6: *Chart of responses on whether the automated tests improved understanding*

When we asked students whether that discussion improved their understanding of the content, all ten said yes, and seven of them said it helped a lot (Figure 4.7). The free-response comments to this question told a similar story, with one student saying "[it] was interesting seeing the other ways students did things," and another saying "I got to see how others did things so I could improve my own solutions."

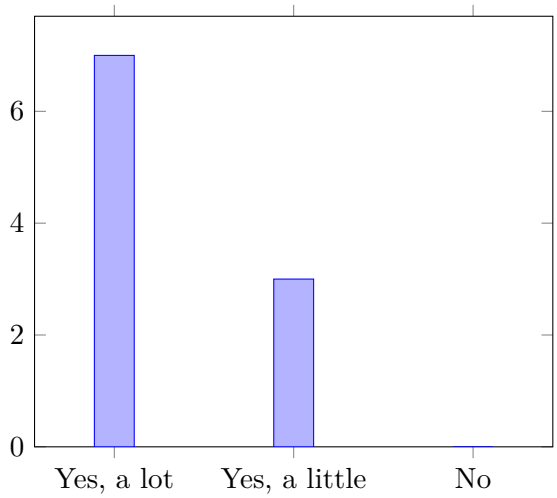


Figure 4.7: *Chart of responses on whether the discussion improved understanding*

4.2.3 Use In Other Classes

The last piece of quantitative data we wanted was information on whether students wanted to see Necode used in other classes. To determine this, we asked students if they wanted to see Necode used in most cases, only when visual output was emphasized, only when automated tests are emphasized, or only in rare cases if at all.

When combined with the students who said they wanted Necode used in most cases, eight of the ten students said they wanted Necode to be used in classes where visual output was emphasized and seven said the same for when automated tests were emphasized (Figure 4.8–4.9).[1] One student said that they thought Necode was best for "classes/examples that can be tested automatically, but it can be very helpful for students to see a visual representation of a program's output (e.g. the result of a pathfinding algorithm on a graph)." Another mentioned that while they liked the idea of using Necode in other classes, they did "not want to be graded on the exercises as it is not similar to my normal coding practices."

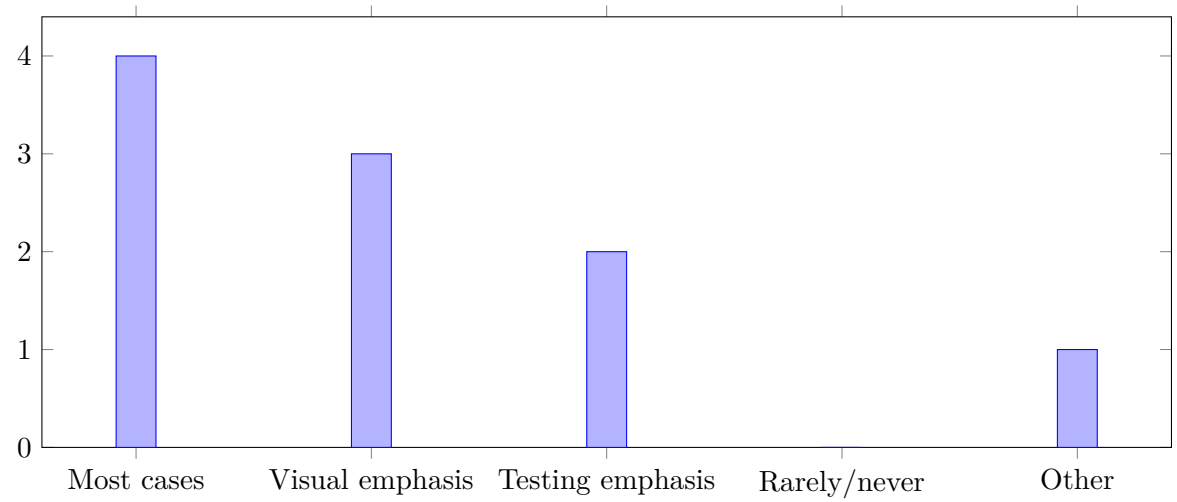


Figure 4.8: *Chart of when students wanted to see Necode used in other classes*

4.2.4 General Feedback

We also received some general feedback on Necode as a whole from some students. One student mentioned an issue with the lack of feedback from the interface when making a submission. Another student suggested a feature where students can refer back to their code and submissions from previous activities (presumably from that overall lesson).

[1] One student said they wanted to see Necode used when there was a visual emphasis or a testing emphasis, but not when there was neither. We do not include them as wanting Necode to be used in most cases, but we do count them towards the cumulative support for using Necode in classes that have a visual emphasis and testing emphasis.

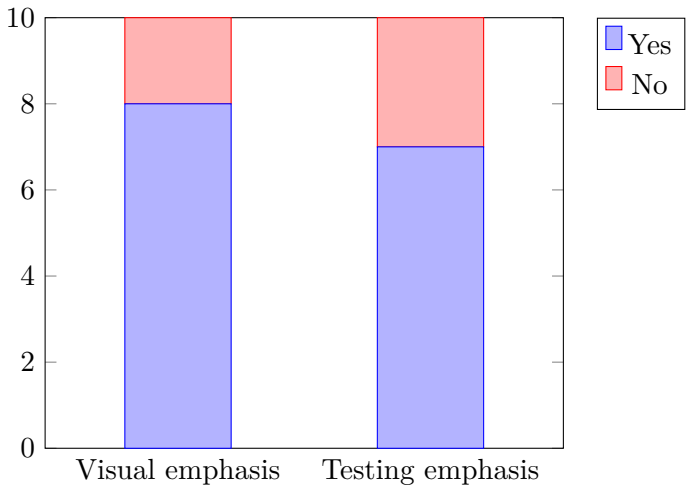


Figure 4.9: *Aggregate chart of when students wanted to see Necode used in other classes*

Chapter 5

Discussion and Conclusion

5.1 Discussion of Results

5.1.1 The Interface

Overall, Necode's interface was well-received from the student perspective, and while there were some bugs with the instructor side, we believe we have a fairly good handle on most of them, and some have already been addressed. Some of the other feedback will be addressed further in Section 5.2.

One area that was interesting to see some critical feedback of was students needing to manually apply changes. We consider manually applying changes to be an important feature for reasons mentioned in Section 3.2.1, though implementing a configurable toggle to enable auto-apply changes may be an option. However, because this is mostly a concern when students are first exposed to Necode, a configuration option is not likely to fix the underlying issue. Clearer explanation of the user interface, either from the instructor or from Necode itself (or both), is more likely to be the most effective solution. We can also investigate ways of changing the interface in order to create a more intuitive flow.

The extremely positive feedback about having a code editor and output pane integrated into one interface is encouraging, though it should not be particularly surprising given how many other educational software tools offer a similar feature. However, it is a good reminder that having an interface conducive to rapid prototyping is immensely valuable, especially in educational contexts. It is similarly encouraging to see that students found it easy to transition to Necode, as it bodes well for using Necode alongside other software like we stated was a goal in Section 3.1.

5.1.2 Automated Tests

The use of automated tests was viewed significantly more negatively than any other part of Necode, though we believe that this may be a side effect of the particular activities that were used in this study. Because the activities visually showed the correct solutions, a student would only need to compare their image to the one they saw on the screen to know if they were right. Some of the free-response feedback also echoed this view. Seven of the ten students said they wanted to see Necode used in other classes that have an emphasis on testing, which suggests that the students actually liked the idea of automated testing, but only when necessary. While we have not yet tested this kind of activity with students, the comment by one of the students saying that the output pane could be useful for seeing a visual representation of an algorithm directly supports a use case we outlined in Section 3.2.4 and Figure 3.8.

While we understand the concern a student raised regarding accepting semantically valid but non-ideal code, disallowing it would both be difficult and could raise some pedagogical concerns. There is no way for Necode to know what the "right" way to write code is supposed to be in all cases, and even if we provided parse tree or AST-traversal tools to the instructor, setting up tests would be both labor-intensive and high-skill, two things we were trying to avoid when creating the testing DSL. Additionally, allowing sub-optimal code to pass provides opportunities for discussion and therefore learning when the instructor goes over student solutions with the class. Even if we could forbid such code from passing, doing so might take away that benefit.

5.1.3 Class Discussion

As mentioned in Chapter 1, one of the key motivating factors behind Necode, and especially behind putting so much focus on the HtmlTestActivity category of activities, was providing the ability for students to work on a problem individually[1] and then to be able to discuss different solutions as a whole class. The response to that part of the lesson was very positive, which supports our hypothesis that this kind of exercise is valuable. While we did not investigate whether the class discussion improves any quantitative performance metric, based on the results we did receive we would still encourage any instructor who uses Necode in their class to use the submission feature for full-class discussion of student solutions.

5.2 Future Work

Beyond bug fixes and interface tweaks, there are a number of areas that we would like to expand on Necode in the future. Firstly, new tools to help instructors use Necode are crucial. Dummy accounts as was mentioned in the feedback would be useful, but we also want to be able to supply template configurations of various activities, and perhaps allow instructors to make

[1] Or in small groups, though the RTC policy and front-end UI for that has not yet been implemented.

their own and share them with other instructors. The ability to move activities between lessons and duplicate activities would also be valuable, and we would also like to add the ability to export and import Necode data so that instructors can back up their course data and process it externally.

We also want to create more activities, and at the same time change how activities are loaded. Currently, the code for every activity type that Necode supports is loaded whenever any activity is loaded, which is manageable for now since there is a large amount of code reuse and a small number of activity types, but is not scalable. We want to restructure how we load activities such that they can be loaded on demand, and potentially so that instructors can design and implement their own activities, and then import them into Necode for use in their class.

Along with more activities, we also want to make more RTC policies, and use our RTC framework to implement a collaborative editor system like many other educational tools have. We believe that there is significant potential in real-time networked classroom activities that currently remains untapped, and we would like to explore it further.

Lastly, while JavaScript and Python (the two general-purpose languages we currently support) are very popular languages, they are much less popular in computer science classrooms, at least here at WPI where we expect Necode to be most used initially. Getting support for Racket, especially the How To Design Programs variants of Racket that we use in our intro CS class, would be immensely helpful, as would getting Java support, the language used in our Algorithms class, and C, the language used in our systems classes. Getting support for these languages would require more investigation into running compiled languages in Necode, as we discussed in Section 3.2.5.1.

5.3 Conclusion

Necode is a valuable tool for teaching computer science using in-class activities that allows students to write code in their browser and see its output live in a visual format. Instructors can write automated tests for students to run their code against, and look at multiple student solutions for full-class discussion. We encourage instructors who are interested, particularly computer science faculty at WPI, to contact us about using it in their classroom. Necode has been set up for WPI and is available at https://code.cs.wpi.edu/. Instructors at other schools and universities may also contact us about how to get Necode set up for their school.

For those who are interested in looking at Necode's source code, filing bugs/issues, or contributing to Necode, the full source is available at https://github.com/TheUnlocked/Necode, along with documentation on setting up an instance of Necode.

Appendix A: Hot Reload

Hot reloading code generally requires a decoupling of state from behavior, so that the behavior can be replaced independently of state. The problem with coupled state and behavior is that when behavior changes, state must also be reset, since they cannot be changed independently of each other. For example, consider this code:

```
// <button id="counter">0</button>
const counter = document.getElementById('counter');

counter.addEventListener('click', e => {
    counter.innerText = +counter.innerText + 1;
});
```

As we clicked the button, we would see its text change in the sequence:

$$1, 2, 3, 4$$

Now suppose we want to change the `+ 1` into `+ 2`. We could try to remove the old JavaScript and add our new updated script back in. However, now if we click the button, we would see the sequence continue after 4:

$$7, 10, 13, 16$$

It's increasing by 3 instead of by 2. This is because the old event listener is still attached to the button even after removing the script, so on every click, it adds 1, and then adds 2 after that, for a total of 3. Unfortunately, that is almost certainly not the intended effect.

While we could do some magic behind the scenes to automatically remove event listeners when code is swapped out (assuming that was even the desired behavior in all cases), there would still be other, far trickier issues to solve. For example, if we make a slight change to the previous code:

```
const counter = document.createElement('button');
counter.innerText = '0';

counter.addEventListener('click', e => {
    counter.innerText = +counter.innerText + 1;
});

document.body.append(counter);
```

We now have a real problem if we try to re-run the code. If we remove the old script and add back the new one, there end up being two buttons. We could try to remove all inserted elements when a script is removed, but that is only a solution to this specific case, not in general. Inserted elements might be an important part of program state.

Hot reload works in Java by replacing method bodies, since method bodies themselves are not involved with storing any state, and therefore can safely be replaced (stateful members like fields still cannot be replaced). This is possible due to how Java is compiled as a set of classes containing methods known at compile time, and it relies on a method being uniquely identified by a combination of its name and signature so that methods in the old and new version of a class can be linked up for swapping out the body [25]. It would be much harder to pull off in JavaScript where functions are dynamically created at run-time, since there is no way for the language to know if two functions are supposed to be the "same" function.

That is not to say there is no way to hot reload code in JavaScript, it just requires significantly more care. If a particular module is pure (that is, loading the module itself doesn't have any side effects), the entire module can be safely swapped out in a practice called "Hot Module Replacement" or HMR. In web frameworks, componentizing content and making state declarative can also go a long way in helping. For example, functional components in React can swap out behavior at will by letting the framework handle state itself, and just having components request the state they need with the `useState` hook (among others). This way, a component only needs to reset its state if the order or number of hooks changes, and because React is componentized, only that component's subtree will need to be reset, anything above it in the component tree is safe. Additionally, in situations where external state not governed by React must be used, React provides a mechanism for clean-up with the `useEffect` hook [26].

However, the DOM Programming activity just uses a single file script (and that script will usually have side effects), rendering HMR useless, and it does not enforce any kind of framework, so the kinds of guarantees that React (and other web frameworks) take advantage of to allow hot reload also do not apply here.

Appendix B: Testing DSL

Note: Interactive documentation is available at https://necode.vercel.app/docs/tests.

Necode's testing DSL is a minor extension of TypeScript that assists in writing simple tests by providing two new features: checks and waits.

Checks

A check is an assertion. The simplest kind of check, literally the function `check`, checks if a condition is true, and if not, reports a test failure with a provided message. That message is then shown to the user via the test dialog (like in Figure 3.7).

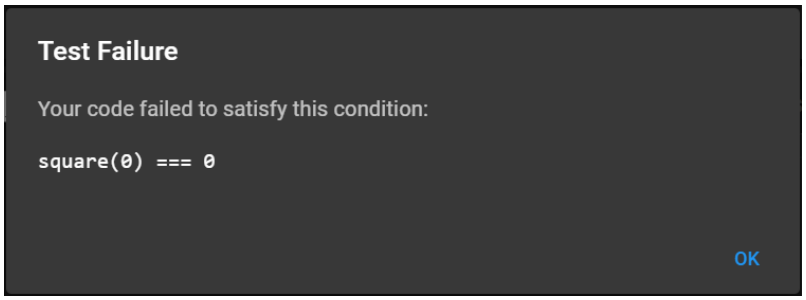
Checks have some special properties however, enabled by a babel transformer. The first is auto-closures:[2] the condition in a check is automatically wrapped in a closure so that it can be more carefully invoked when needed, and guarded by a try-catch statement in case of runtime errors. For example, if a check attempts to invoke a student-written function, but the student has not written that function, the test which invokes the missing function will fail gracefully with the appropriate error message rather than spitting out a ReferenceError that may be meaningless to the student.

```
// If square doesn't exist, this test will fail.
check(square(0) === 0, 'Squaring 0 should be 0!');
```

In the example above, the error message is really just restating the condition. To remove this redundancy, a second feature is made available in checks: the `SHOW_TEST` keyword. By using `SHOW_TEST` in place of a hand-written error message, students will see the actual code written in the condition, which both saves the instructor time and gives students something more concrete to debug their code with. Using `SHOW_TEST` over something hand-written is not always better, especially when a check's condition might involve additional code that the student doesn't have access to, but it can greatly improve the instructor and student experience when conditions are simple.

[2] Swift has a similar feature where a function parameter can be annotated with `@autoclosure` in order to automatically wrap arguments in a closure [27]. While we were not thinking of Swift specifically while implementing checks, there may have been some subconscious inspiration.

```
check(square(0) === 0, SHOW_TEST);
```



While normal checks fail when they encounter an error, `checkError` is also made available in case the instructor wants to verify that a particular piece of code does throw an error. It has the same auto-closure behavior as `check`, but rather than failing if the condition is not truthy, it fails if the condition does not throw an error. A callback can also be given as a third argument if the instructor wants to examine the error before deciding whether the test should pass.

```
const bankAccount = new BankAccount(100);
checkError(
    bankAccount.withdraw(500),
    'BankAccount.withdraw should throw if too much money is removed',
    err => err.name === 'NotEnoughFunds'
);
```

Waits

The `wait` function does what it sounds like it would do—wait. In particular, it performs a non-blocking wait for a specified amount of time, which allows the instructor to test for changes in a student's program across time.[3] In order to make waits non-blocking, the babel transformer is required once again in order to automatically await waits when they appear, since `wait` actually uses `setTimeout` wrapped in a promise behind the scenes:

```
function wait(ms) {
    // 500 ms max to improve responsiveness.
    return new Promise(resolve => setTimeout(resolve, Math.min(ms, 500)));
}
```

[3]While the instructor can enter a number of milliseconds for a wait statement to wait for, there is no guarantee that Necode will wait exactly that amount of time, so checking for specific time-dependent values may be less effective than checking for broader trends or checking if a value lies within an interval.

`waitFor` is provided as a more advanced variant of `wait` that lets the instructor wait for when a particular condition is met. In case the condition is never met, it also comes with a timeout argument, and will return false if the timeout expires without the condition being met (returning true if the condition is met within the timeout). This can pair quite nicely with checks:

```
check(
    waitFor(() => document.getElementById('points'), 300),
    'An element with id "points" didn't appear in time.'
);
```

Unlike with checks, `waitFor` does not use auto-closures. This is because the condition is frequently polled, and so using auto-closures could feel too "magical" to instructors. By having an explicit closure, we indicate to instructors that `waitFor` can evaluate the condition as much as it wants.

Appendix C: Investigative Study

In 2020, Trevor Paley (the author of this paper), along with Lindberg Simpson, Stephen Lucas, and Yongcheng Liu (who are not authors of this paper), created a piece of software called Adversary for a class project. In the early stages of developing Necode, we adapted Adversary into a rudimentary version of what Necode would eventually become, and it was used one time with students in a web development class to solicit feedback to help advise the direction of Necode. There were 11 respondents out of an unknown but much larger number of students. The following is the survey used.

Survey

Several of the questions will refer to "this activity" and "the software." To clarify what those mean,

"This activity" refers to the code-writing and discussion part of the study, in which you attempted to solve programming problems and then discussed your solutions as a class. These questions are aimed at assessing the value of and soliciting feedback on the activity itself, independent of any quirks in the software.

"The software" refers to the actual software tool you used to perform the activity. These questions are aimed at assessing how effective the software was at facilitating the activity, including any technical or user experience issues you may have faced.

1. **Did you do this activity in person or remotely?**
    - In Person
    - Remotely
2. **How would you rate the difficulty of the problems you completed, considering what you have learned so far?**
   Likert scale from 1 (too easy) to 5 (too hard)
3. **Comments**
   Free response
4. **Did the code-writing part of this activity improve your understanding of the content?**

- Yes, a lot
- Yes, a little
- No

5. **Did the discussion of solutions improve your understanding of the content?**
   - Yes, a lot
   - Yes, a little
   - No
6. **How could this activity be changed to improve your learning experience?**
   Free response
7. **Imagine this activity could be used with any programming language. Would you find it helpful for Computer Science classes other than Webware to use this kind of activity in their instruction?**
   - Yes
   - No
8. **If you answered yes, which classes do you think this kind of activity would be well suited for?**
   Free response
9. **If you answered no, or if you answered yes but there are some classes that you think this kind of activity would not be helpful for, why not?**
   Free response
10. **What other classroom activities do you think could be interesting to do with a code editor like the one you used in this activity?**
    Free response
11. **How easy was the software to use?**
    Likert scale from 1 (very easy to use) to 5 (very hard to use)
12. **What would you want changed to improve the usability of the software?**
    Free response
13. **Do you have any other comments about the activity, the software you used, or your experience?**
    Free response

Appendix D: Necode Survey

The following is the survey that was given to students to help evaluate Necode.

1. **How much did Necode impact your enjoyment of this class?**
   Likert scale from 1 (decreased enjoyment) to 5 (increased enjoyment)

2. **How easy/difficult was Necode to use?**
   Likert scale from 1 (very difficult) to 5 (very easy)

3. **Comments**
   Free response

4. **Did having access to a code editor and output pane in Necode improve your understanding of the content?**
   - Yes, a lot
   - Yes, a little
   - No

5. **Comments**
   Free response

6. **Did Necode's automated tests improve your understanding of the content?**
   - Yes, a lot
   - Yes, a little
   - No

7. **Comments**
   Free response

8. **Did discussion of student solutions improve your understanding of the content?**
   - Yes, a lot
   - Yes, a little
   - No

9. **Comments**
   Free response

10. **How easy/difficult was it to transition from your normal programming environment to Necode?**
    Likert scale from 1 (very difficult) to 5 (very easy)

11. **Comments**
    Free response

12. **Would you like to see Necode used in other Computer Science classes that involve programming, potentially using other programming languages?**
    - Yes, in most cases
    - Yes, but only in classes where visual output is emphasized (e.g. Webware, Computer Graphics)
    - Yes, but only in classes where automated tests are emphasized (e.g. Introduction to Program Design, Algorithms)
    - No, or only rarely
    - *Other (free response)*

13. **Comments**
    Free response

14. **Do you have any other comments about using Necode in this class?**
    Free response

Bibliography

[1] M. Grierson, "The codecircle platform," *The Routledge Research Companion to Electronic Music: Reaching out with Technology*, p. 312, 2018.

[2] "Replit docs - introduction to teams for education," [Accessed 2022-02-07]. [Online]. Available: https://docs.replit.com/teams-edu/intro-teams-education

[3] J. W. Johnson, "Benefits and pitfalls of jupyter notebooks in the classroom," in *Proceedings of the 21st Annual Conference on Information Technology Education*, 2020, pp. 32–37.

[4] "Brython," [Accessed 2022-02-07]. [Online]. Available: https://brython.info/

[5] C. K. Lo, K. F. Hew, and G. Chen, "Toward a set of design principles for mathematics flipped classrooms: A synthesis of research in mathematics education," *Educational Research Review*, vol. 22, pp. 50–73, 2017.

[6] K. M. Ying and K. E. Boyer, "Understanding students' needs for better collaborative coding tools," in *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–8.

[7] J. Fiala, M. Yee-King, and M. Grierson, "Collaborative coding interfaces on the web," in *Proceedings of the International Conference on Live Interfaces*. REFRAME Books, University of Sussex, 2016, pp. 49–57.

[8] M. Yee-King, M. Grierson, and M. d'Inverno, "Evidencing the value of inquiry based, constructionist learning for student coders," *International Journal of Engineering Pedagogy*, vol. 7, no. 3, pp. 109–129, 2017.

[9] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized javascript," *Google, Inc., Tech. Rep*, 2008.

[10] K. O'Hara, D. Blank, and J. Marshall, "Computational notebooks for ai education," in *The Twenty-Eighth International Flairs Conference*, 2015.

[11] "Coding rooms - the first virtual classroom for teaching programming," [Accessed 2022-01-31]. [Online]. Available: https://www.codingrooms.com/

[12] "Start coding instantly with replit's browser-based ide - replit," [Accessed 2022-02-07]. [Online]. Available: https://replit.com/site/ide

[13] Y. Asano and K. Kagawa, "Development of a web-based support system for object oriented programming exercises with graphics programming," in *2019 18th International Conference on Information Technology Based Higher Education and Training (ITHET)*. IEEE, 2019, pp. 1–4.

[14] "Mono and webassembly - updates on static compilation | mono," Jan. 2018, [Accessed 2022-02-07]. [Online]. Available: https://www.mono-project.com/news/2018/01/16/mono-static-webassembly-compilation/

[15] D. Yoo and S. Krishnamurthi, "Whalesong: running racket in the browser," *ACM SIGPLAN Notices*, vol. 49, no. 2, pp. 97–108, 2013.

[16] R. M. Siegfried, J. Siegfried, and G. Alexandro, "A longitudinal analysis of the reid list of first programming languages," *Information Systems Education Journal*, vol. 14, no. 6, p. 47, 2016.

[17] "Replit - repl.it gfx: Native graphics development in the browser," Mar. 2019, [Accessed 2022-02-07]. [Online]. Available: https://blog.replit.com/gfx

[18] M. McIlroy, E. Pinson, and B. Tague, "Unix time-sharing system," *The Bell system technical journal*, vol. 57, no. 6, pp. 1899–1904, 1978.

[19] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

[20] "510122 - site isolation for sandboxed iframes - chromium," 2022, [Accessed 2022-02-19]. [Online]. Available: https://crbug.com/510122

[21] "Replit - infinite loops," Jan. 2017, [Accessed 2022-02-19]. [Online]. Available: https://blog.replit.com/infinite-loops

[22] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[23] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 301–312.

[24] L. McCarthy, C. Reas, and B. Fry, *Getting started with P5. js: Making interactive graphics in JavaScript and processing.* Maker Media, Inc., 2015.

[25] A. Villazón, W. Binder, D. Ansaloni, and P. Moret, "Advanced runtime adaptation for java," in *Proceedings of the eighth international conference on Generative programming and component engineering*, 2009, pp. 85–94.

[26] "Introducing hooks - react," 2018, [Accessed 2022-02-18]. [Online]. Available: https://reactjs.org/docs/hooks-intro.html

[27] A. Tsadok, *Pro IOS testing : XCTest framework for UI and unit testing.* Apress, 2020, ch. 3, pp. 49–78.