University of New England

ASCILITE 2020

ASCILTE's First Virtual Conference

ASCILITE
SHAPING THE FUTURE OF TERTIARY EDUCATION

Revisiting the intelligent book: Towards seamless intelligent content and continuously deployed courses

William Billingsley
University of New England, Australia

In the early 2000s, colleagues and I developed *The Intelligent Book* – a suite of technologies for adaptive materials, that let students work with smart graphical exercises as if the AI was their partner rather than their marker. We envisaged a future where online content would be brimming with interactive models, lettings students explore and tinker with problems alongside AI that would guide students in their thinking. The browsers of the day were technically limited, but since then, the technological landscape of the web has transformed. Meanwhile, online education (especially during the Covid-19 pandemic) has grown the need for interactive materials that "understand what they teach" and can make explanations explorable and "proddable". In online education, physical group activities (e.g., programming robots) are not available to us, and we see a growing need for digital experiences and models to replace the responsiveness that comes from tangible interaction with a device or experiment. Over the last two years, I have begun revisiting the ideas of the Intelligent Book for the modern technology landscape. This paper gives an early overview of the project, working once again towards infrastructure for self-publishable courses that can be full to overflowing with proddable and explorable models.

Keywords: Explorable explanations, game-based learning, course self-publishing.

The search for tangible responsiveness online

In this paper, I describe a mission to create self-publishable courses that can be replete with a broad variety of specialised, smart, interactive components. However, let me begin by motivating that with a personal story about the kinds of in-person experience that we still struggle to replace online.

In 2020, universities faced the loss of many of our face-to-face and physical activities. Academics around the world have raced to find ways to replace physical labs and experiments with online experiences. It is not just the teaching of skills that needs replacing, however. Online, we still struggle to recreate the sense of community that comes from group problem solving in a physical environment. We also struggle to recreate the authenticity, tangibility, and immediacy that comes from interacting with physical items. For example, when a student programs a physical robot, they do not just get pedagogical feedback on their work – they get to watch how it behaves in real-time, including its unexpected interactions with the physical environment (such as wheels slipping or sensors failing to see a black line because the material it is printed on is too shiny).

Figure 1: Computational thinking experiences in 2019 were group-based and included physical aspects that drew students away from their screens. Left: The Coding Escape. Right: Cisco Coders Challenge

Many computational thinking experiences, for instance, are designed to avoid portraying programming as a solitary activity trapped in front of a screen. CS Unplugged (Bell, Witten, & Fellows, 2015) introduces

computer science concepts away from the computer, using cards and physical activities. RoboCup Junior (Sklar, Eguchi, & Johnson, 2002) encourages groups of students to solve physical robotics challenges – again, introducing a tangible element that draws students out of their chairs and into problem-solving discussions around physical artifacts. The activities we have run at my own institution, a couple of which are illustrated in Figure 1, similarly always include some group or physical aspect so that programming is not simply about keyboards and pixels. *The Coding Escape* intersperses maze-solving with sticky notes on a large vinyl poster of a maze and games played with giant cards with programming puzzles. The *Cisco Coders Challenge*, run in conjunction with engineers from Cisco, involves maze-solving challenges using inexpensive robots sold in hobbyist stores.

In late 2019, I submitted a proposal for an introductory computational thinking unit through our university's academic approvals processes. This would take the role of a "CS0" unit – a unit introduced before the formal "CS1" introduction to programming that also serves as an engaging introduction to computing for non-majors (Reed 2001; Rizvi et al., 2011), and we wanted it to include some form of these rich, tangible, outreach-style experiences. The unit would sit alongside an introductory unit in digital electronics I also proposed, which is another topic where ideally we would like to bring students into the room – in that case, so they can have the tangible experience of putting circuits together and probing how physical devices behave.

By mid-2020, the country was in Covid-19 lock down and any notion of students gathering in groups around robots or electronics was a distant dream. It wasn't just programming that was locked up in a solitary room with a keyboard and a screen; it was everyone. For us, however, we already faced this problem – the majority of our students are online so would never have been able to attend physically anyway.

There are, of course, online computational thinking and digital electronics courses that are also delivered in person. In computational thinking, for instance, UC Berkeley's AP Computer Science Principles class, the *Beauty and Joy of Computing* (Garcia et al., 2015) is particularly popular. However, it is not simply the content we wished to recreate – we wished to carry the rich and tangible sense of tinkering with devices from physical outreach into the online realm. One of the key differences I would draw is in the size and variety of experience. Online courses tend to be many weeks long, with one dominant environment (in the Beauty and Joy of Computing, the Snap! Programming environment). Physical outreaches are much shorter, but each one might contain more than one physical activity. So, our mission to create a sense of outreach-style tinkering online becomes partly a mission for vastly more plentiful, more varied, smaller, and cheaper interactive models and environments in our courses.

This paper's role is not to describe either our computational thinking or our digital electronics course. Rather, it is to describe and motivate the broader mission – of creating self-publishable courses that can be replete with explorable explanations, low-cost simulations, and interactive models.

Revisiting the Intelligent Book

In the early 2000s, we developed *The Intelligent Book* (Billingsley et al., 2004; Billingsley & Robinson, 2005; Billingsley & Robinson, 2007a) – a suite of technologies for adaptive materials that let students work with smart graphical exercises as if the AI was their partner rather than their marker. Even at this time, we envisaged a future where online content would be brimming with interactive models, lettings students explore and tinker with problems alongside AI that would guide students in their thinking. These were inspired by John Seeley Brown's much earlier concepts of reactive learning environments (Brown, Burton & Bell, 1975). However, in the early 2000s, browsers were technically limited. Displaying interactive vector graphics was unreliable between browsers, JavaScript was slow, and implementing genuinely bidirectional conversations between client-side diagrams and a smart model (which then typically needed to reside on the server) required special techniques. The ecosystems and mechanisms for developing and publishing web libraries were also much more limited than today. Although we were able to develop smart interactive exercises, practically we were always hampered by a cost-benefit trade-off. More specialised models are more intricate to develop, but address a narrower range of questions within a course (Billingsley & Robinson, 2007b).

Since then, the technological landscape of the Web has transformed. Browsers are technically more capable, most programming languages include JavaScript as a compilation target, and modern visualisation and front-end frameworks such as *D3* (Bostock, Ogievetsky, & Heer, 2011), *React* (Walke, 2013), and *Vue* (You, 2014) have made it common practice to use a client-side component model to construct composable complex web

interfaces. Infrastructure for publishing and consuming web libraries, even across programming languages, have also improved greatly.

Over the last two years, I have been exploring how a revisited Intelligent Book might apply to the modern Web. Back in the early 2000s one of the motivations we expressed for the project was that we wanted intelligent adaptation and interaction, but most Web materials were static displays of content with limited interactivity. In the early 2020s, the wider Web is highly interactive, but many course materials still are not. Smart materials that understand and model what they teach are rare and costly to produce, and adaptive learning materials still predominantly use drag-and-drop multiple choice or other styles of question that require no domain-specific model. More complex domain-specific environments are used in tutorials and assignments, but usually as a separated environment. Integrating these holistically into the course environment is seen as a technical challenge worthy of research – for instance in the United States, the NSF-supported *Standards, Protocols, and Learning Infrastructure for Computing Education (SPLICE)* project has been organising workshops on interoperability between 2017 and 2020. A brief early proposal for this project was presented at one of these workshops in 2019 (Billingsley, 2019).

From an engineering perspective, I would argue that Learning Management Systems (LMSs) may have had an isolating effect on university web materials. In the early 1990s, it was common for computer science subjects to have a course website containing their materials and these were often developed bespoke by the lecturer using the prevalent techniques of the day. Quality varied and in the late 1990s, LMSs were introduced and over subsequent years have come to dominate the digital experience of university subjects. Although this enabled universities to take a more professional, broader, and more consistent view of the design of its digital teaching experiences, it also isolated course website development practices from broader web development practices. No longer are course websites developed for the web *at large*, but for the smaller, more specialised, and often pay-walled confines of the institutional LMS. These have become different ecosystems with different practices. Over the last decade, terms such as *build pipelines, dependency management, and continuous deployment* have become commonplace in website development, but are rare in course site development.

The story of revisiting the Intelligent Book for the modern web, then, begins at the front end – exploring modern engineering approaches to how content and interaction can be compiled, published, and delivered to students' browsers.

Veautiful – an Experimental Front-End Framework

This particular story of development begins with a "front end framework". Although browsers are highly capable, their document model is low level. HTML and SVG deal with individual text and graphical items on the screen, and it is left to front end frameworks to define higher level concepts such as components and how the view elements should be reconciled as the data in a program changes. Many front end frameworks have developed over the last decade, such as D3 for diagrams (Bostock, Ogievetsky & Heer, 2011), and React (Walke, 2013) and Vue (You, 2014) for more general components.

When combining different kinds of interactive content, one of the challenges can be from incompatibilities between the different frameworks they use. The difficulty is not simply that there are many different frameworks, but that they are *opinionated* frameworks that tend to shape your code. For example, React and Vue use their own "*virtual DOM*", so a front-end program primarily interacts with the framework's virtual data model rather than with the elements showing in the browser, whereas D3 is based on synchronising the browser's own elements with arrays of data. If the goal of our teaching materials is that we can combine a wide variety of complex (and sometimes third party) components in quick succession, then an opinionated front-end framework may become confining: whichever you pick, most components might be better developed differently.

Veautiful is a less opinionated framework that allows components to take different approaches to how they perform "reconciliation" (updating the page). This seems to make it easier to write pages that include plentiful diverse componentry. This may be best illustrated with some of the examples I have deployed in outreach and teaching materials over the last year: from *The Coding Escape* outreach, from *Circuits Up!* materials for digital electronics, and from *Thinking About Programming* materials for computational thinking.

Figure 2 shows two very simple components from *The Coding Escape*. These each hold an internal model of their problems, inside a hand-written function, and are dynamically rendered as elements in a declarative style. They are similar to React or Vue components, and would be as easy to write in any other framework.

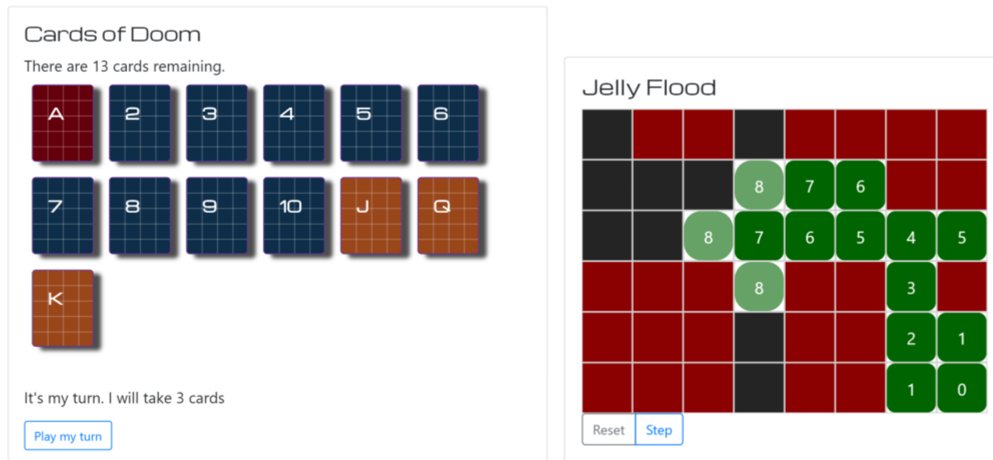


Figure 2: Simpler components from The Coding Escape that are interactive graphics with a short custom model. Left: A card game demonstrating algorithm. Right: A stoppable example of flood-fill pathfinding.

On many occasions, however, we need to drive two different styles of component from the same model to produce an interactive and responsive illustration. Figure 3 shows two elements from a page on MOSFETs in *Circuits Up!* Behind the diagrams, the model uses a constraint propagator that understands electronic circuit laws, to simulate the MOSFET. This is simultaneously shown in the circuit diagram on the left (written quickly in a declarative style similar to React) and in the chart on the right (as a chart, written quickly in a style more similar to D3). Drag the sliders to draw the chart, while also seeing changes reflected in the circuit diagram – for instance, the thickness of the pink conductance channel at the gate grows or shrinks with the gate voltage.

This link between the chart and the diagram also helps us visualise three variables using a two-dimensional chart. The chart on the right is currently showing the response of the current (IDS) against the drain voltage (VDD) when the gate voltage (VGB) is 4.6V. Move the VGB slider, and the point you are looking at (currently for VDD=1.7) will slide up and down the chart with you, while the other points on the chart are cleared. Sliding VDD back and forth then charts the response curve for the new value of VGB. Interactively, you can explore how all three variables relate to each other.

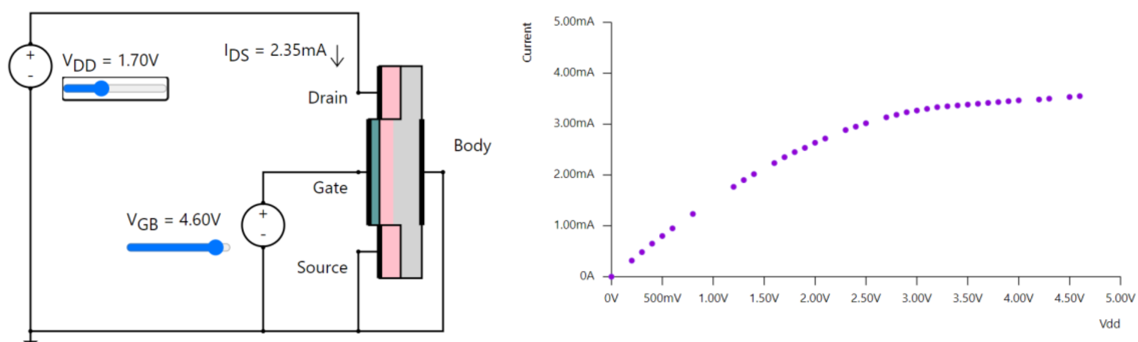


Figure 3: Different styles of component driven from a single more-complex model.

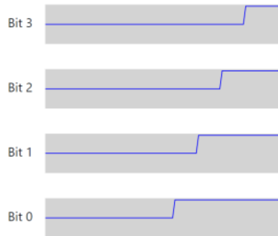
The number of live elements in a tutorial page can quickly start to proliferate. Figure 4 shows a page from *Circuits Up!* demonstrating the timing of ripple carry adders. The output of the simulation here is reflected across all elements of the page – being shown in timing charts for the bits, views of the output as a “nibble” (half a byte) in memory, values and wire-colourings in the circuit diagram, and the instruction text also automatically updates when the circuit has finished rippling up or down.

Ripple Carry Timing

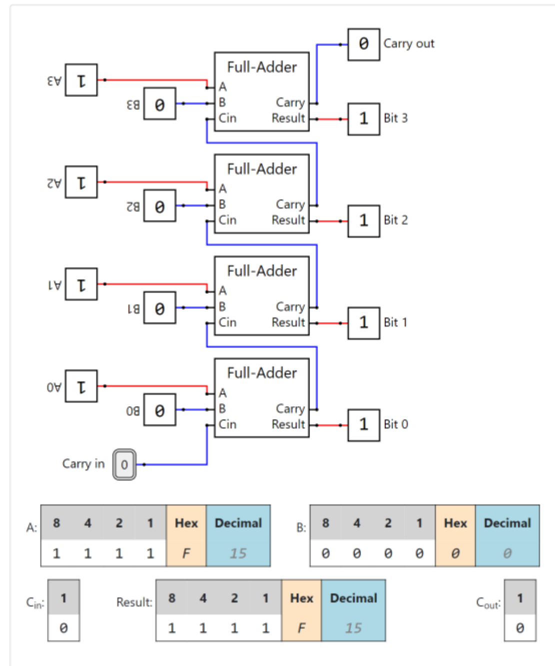
Electronics is fast, but it's not instant. Let's set up the Ripple Carry Adder that we saw previously again, only this time let's fix all its inputs except the carry, so we can watch what happens as the carry ripples through. I've also artificially slowed down the logic resolution, to make it more visible.

This start/stop button will start and stop the animation (including the operation of the circuit). When you're ready, start the animation and notice the logic lines in the graph (which will start in the muddy middle)

Start



Now, go and toggle the "carry in" on the circuit up and down, and see what happens to the chart, the outputs, and even the "result" shown under the circuit.



8	4	2	1	Hex	Decimal
1	1	1	1	F	15

8	4	2	1	Hex	Decimal
0	0	0	0	0	0

8	4	2	1	Hex	Decimal
1	1	1	1	F	15

Figure 4: The Ripple Carry timing page from Circuits Up!

Control: if ... else ... / do ... while ... / while ... / for ...

Functions: Name myFunction / Params a, b, c / Declare / Call

```
while (canGoDown()) {
  down()
}
do {

} while ()
if () {

} else {

}
```

Trash

Text

You can print to this text area using println("Hello"). You can also print to the browser's own "console" using console.log("Hello").

Arithmetic

Comparisons

```
1  addLineSensor(20, -6, 0, 255, 0) // line se
2  addLineSensor(20, 6, 0, 255, 0) // line ser
3  addLineSensor(8, 8, 255, 0, 255) // junctic
4
5  addLineSensor(0, 0, 255, 0, 0) // stop sens
6  addLineSensor(0, 0, 255, 255) // stop se
7
8  let stop = false
9
10 while (!stop) {
11   let l = readSensor(0) > 0.5
12   let r = readSensor(1) > 0.5
13
14   let jr = readSensor(2) < 0.5 // if this c
```

Tiles

Figure 5: Mixing and matching programmable game environments allows us to show a broader variety of experiences. Above: A grid-like game similar to Hour of Code experiences, using our Scatter blocks programming environment. Below: A simulation based on RoboCup Junior Rescue Line

On other occasions, we may need to mix and match components. Figure 5 shows two examples from *Thinking About Programming*. In the uppermost picture, a blocks programming language (Scatter) is being used to program a robot in a simple grid-like game. In the lower example, the programming has been toggled into text-mode using a foreign component (the Ace.js editor from Amazon's Cloud9 development environment) while the game has been switched for a simulation of RoboCup Junior's Rescue Line robotics task. In each case, however, they are combined into a *codable* widget that lets students write and run programs from inside the lecture notes.

By making interactive components smaller, more plentiful, and easier to adapt, we also make it easier to introduce variations to illustrate new topics. In Figure 6, all four screenshots from use small adaptations of a programmable turtle graphics environment. In the top-left, this is used to demonstrate recursion (generating a Hilbert curve). In the top right, by adding a background picture, we can introduce "dead reckoning" by trying to sail the HMS Turtle into the English Channel. In the lower two frames, we give the turtle sensors so it can read from the canvas, and illustrate how the stability of a robot can depend on the placement of its sensors. Other variations we have created include introducing physics engines – allowing us to drive robots into obstacles and simulate Moon landings.

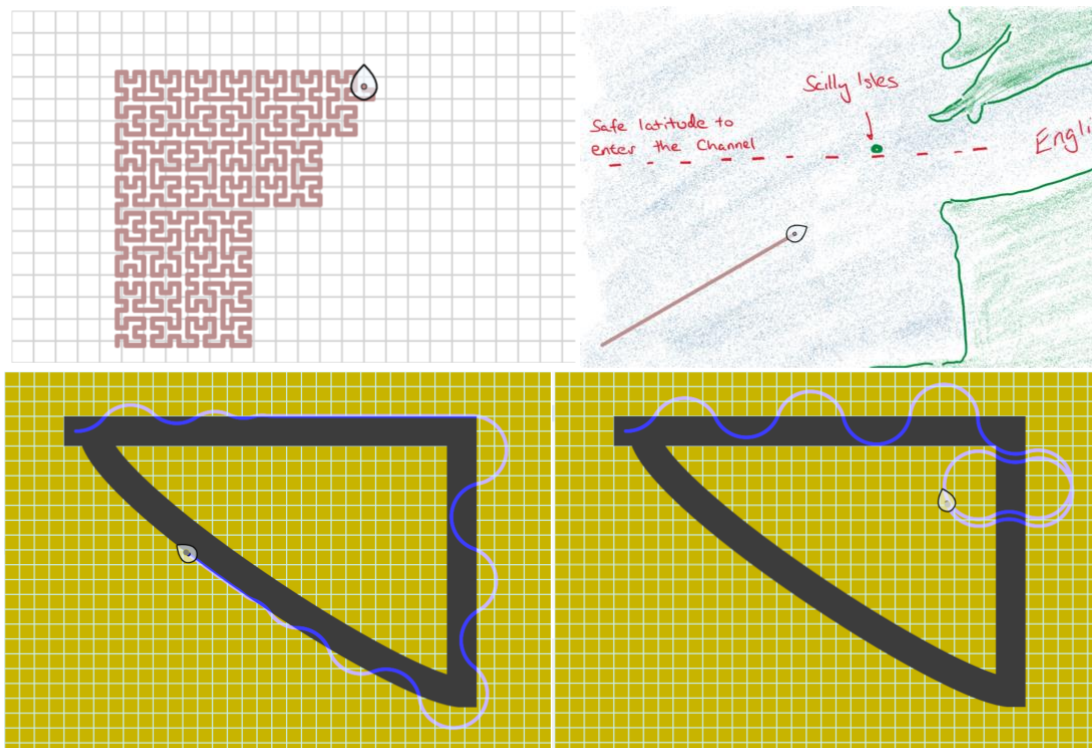


Figure 6: Different adaptations of a turtle graphics environment in Thinking About Programming

End-User Programmable Interactive Slide Decks

The explorable explanations and interactive environments I have been developing are not solely targeted at tutorials and practicals – they are intended to be pervasive, including in the lecture and video slides. Already, we have had feedback from some students that having seen something demonstrated in a video, they appreciate that they can open the slide deck and try it themselves.

One question this produces is how these can be authored – whether content developers can take on processes that are more similar to programming. Here, the project takes inspiration from the fact that many academics already do. At our institution, approximately a quarter of computer science courses have used *Remark.js* (Bang 2011) to generate slides and notes from Markdown since 2016. Mathematicians, physicists, and other scientists around the world are familiar with the *Beamer* package for LaTeX. Each of these works in a similar way – slides are written in a text format that is then compiled or interpreted to produce the visual deck. Essentially, this is programming.

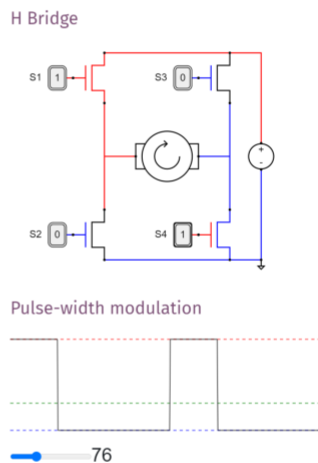
Accordingly, I wished to examine whether we can simplify the programming of interactive decks, so that it is comparable in complexity to writing Markdown or Beamer slide decks. To examine this aspect, I created a component (VSlides) that renders a web slide deck from a script using a simple builder notation. Figure 7 shows one such example, together with its code.

H Bridge Motor Driver

The motor driver takes our digital outputs and uses them to let us drive a DC motor in a forward or backward direction.

This switches the motor between running *at full speed* forwards or backwards.

To do speed control, we're going to need to add *pulse-width modulation*, where we pulse the output so it is being driven for a period in proportion to how fast we want the motor to run.



< 12 / 13 >

```
.veautifulSlide(
  <.div(
    <.h2("H Bridge Motor Driver"),
    Challenge.split(
      Common.markdown(
        """
          |The motor driver takes our digital outputs and uses them to let us drive a DC motor in a forward or
          |backward direction.
          |
          |This switches the motor between running *at full speed* forwards or backwards.
          |
          |To do speed control, we're going to need to add *pulse-width modulation*, where we pulse the output
          |so it is being driven for a period in proportion to how fast we want the motor to run.
          |
          |""".stripMargin)
    )(
      <.h3("H Bridge"),
      HBridge(),
      <.h3("Pulse-width modulation"),
      PWM()
    )
  )
)
```

Figure 7: A live slide interactively illustrating H Bridge motor drivers and Pulse-Width Modulation. Above: the rendered slide. Below: the deck code defining the slide in a declarative style.

Markdown slide decks are used by computer science lecturers perhaps more than by other disciplines, so the goal at the moment is to have decks defined in a concise declarative programming style, rather than to make them completely amenable to non-programmers. Veautiful itself is written in Scala.js, a mixed-paradigm language that has very good JavaScript interoperability as well as good support for “domain specific languages”. This allows us to intersperse logic and markup in a readable manner without requiring dedicated file formats

such as React's JSX or Vue's format for single file components. HTML components (such as div and h2) are interspersed with higher-level layout (*Challenge.split* divides the slide left and right), content rendered from markdown, and live widgets (the *HBridge* and *PWM* components). Defining slides and components from JavaScript will also be forthcoming in future versions.

Browsers, of course, can be resized and "responsive design" (how a webpage responds in order to ensure a good user experience regardless of its aspect ratio) is a complication that content-developers should not need to deal with. Consequently, VSlides mimics an aspect of traditional slide decks: the deck is always defined as having fixed dimensions and therefore a fixed aspect ratio. The deck component uses the browser's scale transform to resize its visual display without affecting the content's internal dimensions. For example, the H Bridge circuit model and pulse-width modulation chart can simply be defined in a fixed coordinate space without having to worry about what happens when the window is resized.

As well as slide layouts, we also frequently use a "challenge" layout for tutorials, although the syntax for this has not been sufficiently simplified to show it in the paper yet. Challenges also use a fixed aspect ratio for simplicity (typically 1080p) but add navigation components, so that students can progress through named "levels" made up of a sequence of "stages". Stages can be of different types (e.g., video or exercise stages) and track their completion progress so that students can see what has been fully or partly completed. Our eventual goal is to make scripting interactive tutorials as simple as scripting slide decks.

Although the framework tries not to be opinionated in programming, there are some design aspects where assumptions do creep in. For instance, the challenge layouts keep all the stages of one level loaded at a time. This ensures that if you flick forward from a video stage to an exercise stage, flicking back again will not require the video to be re-loaded by the browser. However, we only keep one level at a time loaded in the browser's internal model, so that challenges do not become memory hogs.

Composability, Reuse, and Continuous Course Deployment

When course materials are written in a software-like manner, some integration problems that are traditionally "hard" for course materials to achieve are made easier by common public software engineering infrastructure.

As a framework, Veautiful is published to GitHub. So are many of my exercise components, as well as the sites themselves. With modern software infrastructure, this does not simply mean that the source code is available for someone to clone or download; it also makes them simple to integrate into other sites. *JitPack.io* is a public service that allows programs to express a library dependency on a GitHub repository. When a program's build tool requests the library, JitPack automatically checks out the relevant revision of the repository, builds the sourcecode in the cloud, and supplies the compiled library for the build tool do download.

As our sites are built using Scala.js, they are compatible with this tool, and importing from one course into another simply requires adding a dependency on the course's JitPack artifact in the build file. In Figure 7, the DC motor in the circuit is an electronics component defined inside *Thinking About Programming*, using a circuit system that (at the time of writing) is defined in a different site – *Circuits Up!* To import this circuit framework, *Thinking About Programming* simply had to express a one-line dependency on *Circuits Up!* in its build file, shown in Figure 8.

```
libraryDependencies ++= Seq(
  "org.scala-js" %%% "scalajs-dom" % "1.0.0",
  "com.github.(author).veautiful" %%% "veautiful" % "master-SNAPSHOT",
  "com.github.(author).veautiful" %%% "veautiful-templates" % "master-SNAPSHOT",
  "com.github.(author).veautiful" %%% "scatter" % "master-SNAPSHOT",
  "com.github.(author)" % "lavamaze" % "master-SNAPSHOT",
 "com.github.theintelligentbook" % "circuitsup" % "master-SNAPSHOT"
 )
```

Figure 8: Thinking About Programming expresses a one-line dependency (in bold) on Circuits Up! to import its circuit system and other componentry

Self-publishing and continuous deployment of materials are likewise already supported by public infrastructure. *Circuits Up!* and *Thinking About Programming* are each published to GitHub. Another public service, *Travis-CI*, monitors the repositories. When any change is made, Travis-CI downloads the latest changes and builds the

site in the cloud, then deploying it to GitHub Pages (technically, by pushing the built site to the *gh-pages* branch of the repository).

This allows a *continuous deployment* style of development. The content author can develop decks and on their computer. When they commit a change to the source-code repository, it is automatically compiled and published into the public site. Figure 10 shows a fragment of the build history for *Thinking About Programming*.

Within the university, we also take advantage of the fact that this means the built site can also be cloned – it is contained in the *gh-pages* branch in the repository. *Thinking About Programming* is produced as research and outreach and is personally published publicly. Within the university, the decks and exercises that are reused in teaching courses are brought onto our computer science student development server by cloning the built site. This allows the course LMS to link directly into those examples without depending on an academic's personal publication site, as well as ensuring a full history of my open source code is kept for preservation reasons. As the repository allows versions to be tagged, it would also for instance allow the version that was current in a given year to be archived efficiently, or for a course to control when it retrieves changes from the public version.

Commit	Build	Duration / Date
Added examples for H Bridge and PWM	#30 passed f54cb85	2 min 51 sec 6 days ago
Started adding sensors deck	#29 passed 9f5efe4	2 min 55 sec 7 days ago
Fix bug where sensor limit wasn't applied in exercises	#28 passed 8f0ea06	2 min 43 sec 12 days ago
Slightly turned down LineTurtle inaccuracy	#27 passed 267d37d	2 min 48 sec 12 days ago
Added challenge stages to RescueLine	#26 passed 3db4972	2 min 45 sec 12 days ago

Figure 10: A fragment of the Travis-CI build history for Thinking About Programming.

Conclusions

It should not be surprising that courses can benefit from being regarded as engineered software products. They are digital artifacts experienced and interacted with over the same platforms and channels (browsers and HTTP) as other common software products. There is, of course, a deep challenge that most courses are written by academics in non-computing disciplines, so the forms and functionality of software development are unfamiliar to those academics.

As the project progresses, I intend to examine ways this can be made more amenable to non-programmers. However, in the interim there are many fields (such as computer science) where academics are technically adept and are used to using systems that have programming-like features. For example, the LaTeX typesetting system that is used for many journal articles and conference papers in computing, mathematics, and physics conferences is also a programming-like environment with particular workflows. Markdown has seen significant uptake for presentation slides in computing and other fields. Static site generators for personal blogs, such as Gatsby and Jekyll have also grown in use and popularity in recent years, and again are based on declarative, text-based definitions of content.

Just as our courses are continuously deployed, so too this project is continuous and this paper represents a snapshot in time from it. Though the technology is under development, it has been deployed live in multiple contexts. *The Coding Escape* outreach ran using it in 2018 and 2019, each time with a little over two hundred primary and secondary school students. *Circuits Up!* is used by colleagues for the first four tutorials of a first year digital electronics and computer architecture subject that runs twice annually. Decks, tutorials, and exercises from *Thinking About Programming* are used in our online computational thinking unit, which also runs twice annually, though at the time of writing we are mid-way through its first offering.

At the time of writing, I would offer the following conclusions:

- Interactive educational examples, especially for broad topics such as computational thinking, can quickly find they require a broad and complex variety of components. These may lend themselves to a variety of development styles and require a great deal of flexibility from the component systems they are created in.
- When the development of them is made small enough, interactive models can become pervasive through course materials. They can become fine-grained enough that individual "live" slides within individual classes can embed multiple models.
- Although the physicality of devices is hard to replace, by compiling these small models to JavaScript and embedding them within materials, they can be responsive enough that they give a mechanical style of feedback, rather than only pedagogical feedback. With the inclusion of simulated inaccuracy or physics libraries, they can also some of the realism of control systems – as wheels slip and sensors are not always perfectly accurate.
- Interactive materials can be expressed in a manner that mixes declarative content with interactive components. Although this is programming and requires some technical skill, it is not wholly alien to other forms of document preparation – for example, the use of static site generators for blog publishing or LaTeX typesetting for paper preparation.
- When course materials are viewed as a software product, common techniques from software development (such as library reuse and continuous deployment) can allow complex material to be developed and delivered incrementally, in an agile manner, by a resource-constrained team. Apart from the inclusion of external libraries, the examples in this paper were written by a lone academic with a high teaching load who was also building the framework.

References

Bang, O.P. (2011). Remark.js [Computer Software]. https://github.com/gnab/remark.

Bell, T., Witten, I., & Fellows, M. (2015). *CS Unplugged: An enrichment and extension programme for primary-aged students*. Canterbury, New Zealand: University of Canterbury.

Billingsley, W., Robinson, P., Ashdown, M., & Hanson, C. (2004). Intelligent tutoring and supervised problem solving in the browser. In *Proceedings of the IADIS International Conference WWW/Internet 2004*, Madrid, Spain, 806–810.

Billingsley, W. & Robinson, P. (2005). Towards an intelligent online textbook for discrete mathematics. In *Proceedings of the 2005 International Conference on Active Media Technology (AMT 2005)*, Kagawa, Japan: IEEE, 291–296. https://doi.org/10.1109/AMT.2005.1505353.

Billingsley, W. & Robinson, P. (2007a). Student proof exercises using MathsTiles and Isabelle/HOL in an Intelligent Book. *Journal of Automated Reasoning, 39*(2), 181–218. https://doi.org/10.1007/s10817-007-9072-3.

Billingsley, W. & Robinson, P. (2007b). Searching questions, informal modelling, and massively multiple choice. In *Beyond Control: Learning Technology for the Social Network Generation.* Research Proceedings of the 14th Association for Learning Technology Conference (ALT-C 2007), Nottingham, England, 159–168.

Billingsley, W. (2019). A Technologist's Agenda for Scriptable, Smart, Social, and Republishable Courses. In *CS Education Infrastructure for All II: Enabling the Change*, SPLICE Spring 2019 Workshop, Minneapolis, Minnesota, USA. https://hdl.handle.net/1959.11/26766.

Bostock, M., Ogievetsky, V., & Heer, J. (2011). D3 – Data Driven Documents. *IEEE Transactions on Visualization & Computer Graphic*s, *17*(12), 2301–2309. https://doi.org/10.1109/TVCG.2011.185.

Brown, J.S., Burton, R.R., & Bell, A.G. (1975). SOPHIE: A step towards a reactive learning environment. *International Journal of Man-Machine Studies 7*, 675–696.

Garcia, D., Harvey, B., & Barnes, T. (2015). The Beauty and Joy of Computing. *ACM Inroads, 6*(4), 71–79.

Kibler, S. G., Hauer, A. E., Giessel, D. S., Malveaux, C. S., & Raskovic, D. (2011). IEEE Micromouse for mechatronics research and education. In *2011 IEEE International Conference on Mechatronics*, 887–892.

Reed, D. (2001). Rethinking CS0 with JavaScript. *ACM SIGCSE Bulletin*, *33*(1), 100–104.

Rizvi, M., Humphries, T., Major, D., Jones, M., & Lauzun, H. (2011). A CS0 course using Scratch. *Journal of Computing Sciences in Colleges, 26*(3), 19–27.

Sklar, E., Eguchi, A., & Johnson, J. (2002, June). RoboCupJunior: learning with educational robotics. In *Robot Soccer World Cup* (pp. 238–253). Springer, Berlin, Heidelberg.

Walke, J. (2013). *React.js* [Computer Software], Menlo Park, California: Facebook. https://reactjs.org [Accessed August 2020].
You, E. (2014). *Vue.js* [Computer Software], https://vuejs.org [Accessed August 2020].

Billingsley, W. (2020). Revisiting the Intelligent Book: Towards Seamless Intelligent Content and Continuously Deployed Courses. In S. Gregory, S. Warburton, & M. Parkes (Eds.), *ASCILITE's First Virtual Conference*. Proceedings ASCILITE 2020 in Armidale (pp. 230–240). https://doi.org/10.14742/ascilite2020.0144

Note: All published papers are refereed, having undergone a double-blind peer-review process.
The author(s) assign a Creative Commons by attribution licence enabling others to distribute, remix, tweak, and build upon their work, even commercially, as long as credit is given to the author(s) for the original creation.

© Billingsley, W. 2020