f3.js: A Parametric Design Tool for Physical Computing Devices for Both Interaction Designers and End-users

Jun Kato **Masataka Goto**
National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Japan
{jun.kato, m.goto}@aist.go.jp

ABSTRACT
Although the exploration of design alternatives is crucial for interaction designers and customization is required for end-users, the current development tools for physical computing devices have focused on single versions of an artifact. We propose the parametric design of devices including their enclosure layouts and programs to address this issue. A Web-based design tool called **f3.js** is presented as an example implementation, which allows devices assembled from laser-cut panels with sensors and actuator modules to be parametrically created and customized. It enables interaction designers to write code with dedicated APIs, declare parameters, and interactively tune them to produce the enclosure layouts and programs. It also provides a separate user interface for end-users that allows parameter tuning and dynamically generates instructions for device assembly. The parametric design approach and the tool were evaluated through two user studies with interaction designers, university students, and end-users.

Author Keywords
Integrated development environment; physical computing; parametric design; personal fabrication.

ACM Classification Keywords
H.5.2. Information interfaces and presentation (e.g., HCI): User interfaces – GUI; D.2.6. Software engineering: Programming environments – Integrated environments.

INTRODUCTION
Recent advances in personal fabrication have lowered the threshold for creating three-dimensional (3D) models, which has enabled physical objects to be fabricated. Tools for novices to program physical computing devices have also been proposed, although current tools have typically focused on creating single versions of and devices.

It is important for interaction designers to create various kinds of design alternatives during the prototyping process [9]. However, manually exploring and managing design options are tedious. In addition, compared to the fabrication of 3D models with computer-aided design (CAD) tools, the creation of physical devices usually involves programming in an integrated development environment (IDE), making exploration of design alternatives more difficult.

There are tools for novices and users without knowledge on how to use CAD tools or IDEs to create devices with a predefined set of modules and functions [27]. However, there is a certain ceiling above which they still need to rely on professionals. Thingiverse Customizer [17] allows to create parameterized 3D models that can be customized by end-users, but they cannot fabricate computing devices.

Tool support is required for both interaction designers and end-users to efficiently generate and manage variations in devices. This paper proposes to satisfy these needs with a parametric design tool (**Figure 1**). A Web-based design tool called **f3.js** (***f**orm **f**ollows **f**unction(), written in **J**ava**S**cript*) is presented as an example implementation, which is publicly available at http://f3js.org. Given the programmatic nature of parametric design, we chose an IDE rather than a CAD tool as the baseline method. The current f3.js design tool supports the creation of devices consisting of planar surfaces rather than that of general 3D shapes.

Interaction designers write JavaScript programs within the f3.js design tool with our dedicated APIs that define both the hardware layout and software of physical computing devices. They can declare the parameters for generating the design alternatives with a variety of shapes and features and tune the parameter values to explore the design space. Furthermore, f3.js provides a separate user interface for end-users with the

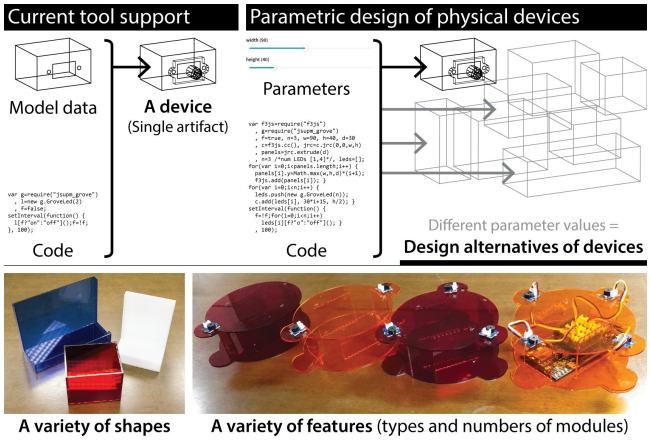

Figure 1. Parametric design of physical computing devices allows their variations to be easily explored and fabricated.

DIS 2017, June 10 - 14, 2017, Edinburgh, United Kingdom

ACM 978-1-4503-4922-2/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3064663.3064681

interactive parameter-tuning interface and dynamically generates instructions for device assembly so that they can build the customized devices by themselves.

The rest of the paper is organized as follows: First, our observations on three aspects of the design process are explained, and our design goal is presented according to these observations. Then, our approach is compared to prior work to clarify research contributions. Next, the interaction design of the tool is presented in detail. Finally, we explain the tool evaluation through two user studies with people from diverse technical backgrounds, which is followed by lessons learned and implications for future work.

PRELIMINARY OBSERVATIONS

This section discusses three aspects of the design process for physical computing devices, each of which motivated us in our goal of achieving parametric design for the devices. We obtained these insights by collecting and analyzing the photographs of 200 devices, using our experience (one of the authors had more than seven years of physical computing research), and conducting informal interviews with interaction designers and software engineers with varied levels of experience in personal fabrication.

Design Alternatives for Hardware Layouts and Software

A physical computing device is a combination of hardware (a device enclosure with sensor and actuator modules) and software (a program that drives the modules), both of which are designed with different tools – CAD tools for the former and IDEs for the latter.

This separation might work well in creating a single artifact of the device, but it is cumbersome to create and manage multiple variations of the device with different enclosure shapes and program features. Exploring design alternatives involves numerous iterations of splitting and merging the two independent workflows. For instance, testing different numbers of buttons in a device requires operations with both tools. A tool should provide integrated design support for both hardware layouts and software to address this issue.

There has been much research on investigating [19] and improving [24,27,37,38,25,26] the process of fabricating physical objects with sensors and actuators. However, few researchers have investigated the programming process, and no work, to the best of our knowledge, has focused on the parametric design of both hardware and software, as is reviewed in the related work section.

Mental Gap between Software and Hardware

As was discussed in the last subsection, the devices are typically created with two independent tools. It is difficult to imagine the appearance of devices while writing code in an IDE. For instance, code that initializes a button driver does not infer any hardware layout, and requires additional operations in a CAD tool to specify it.

In contrast, conventional graphical user interface (GUI) applications can solely be created within IDEs, which provide integrated support for the entire development process. While interface builders in IDEs are used for defining the views and code editors for functions, a programmer can seamlessly switch back and forth between these tools. This is because they are just two different representations of the definition of applications features that are written in text-based code.

These observations inspired a design tool that supports code-centric development similar to that in GUI applications.

Design Patterns in Physical User Interfaces

We asked seven people to send photographs of devices that ran on electricity that they used in their daily lives to enable us to better understand physical user interfaces (PUIs) within existing computing devices. The photographs of 200 devices were collected (**Figure 2**).

As a result, we found that most of them (187) had simple geometries constructed from mostly planar surfaces, while 13 manually operated devices that did not need glancing, such as gaming mice and car handles with numerous buttons, tended to have more complex geometries. We also found certain design patterns for such PUIs. The most common pattern found in 139 devices was to align sensor and actuator modules along a certain straight line, and the second pattern found in 51 devices was to align modules around a certain circle.

These observations suggested that a design tool for physical devices should have dedicated support for these typical design patterns. Such support would further make the layout definition adaptable to changes accompanied by the prototyping process.

We decided to focus on the design process of *completely* planar user interfaces (as opposed to *nearly-flat* or *freeform* user interfaces) as an initial step. The prototypes were assembled from multiple panels, which were cut out from a larger panel with laser cutters or cutting plotters. Interaction

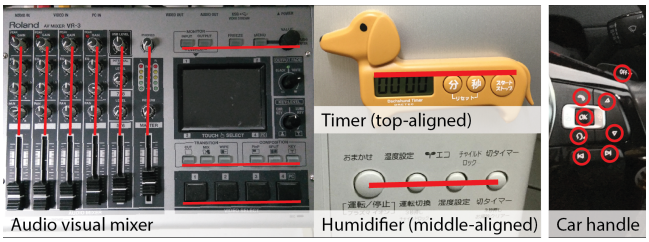

Figure 2. Physical devices following design patterns (and car handle that does not follow pattern).

designers and researchers often use two-dimensional (2D) drawing tools such as Adobe Illustrator to define cutting paths for enclosures. Since they are designed for general graphics, they have often induced mistakes of mismatched joints between planes, which we intend to address.

DESIGN GOAL: PARAMETRIC DESIGN OF DEVICES

Here, given the preliminary observations, we introduce a code-centric development process in which every aspect of a physical computing device is defined in a single codebase, which enables parametric design and addresses three kinds of difficulties in the design process (**Figure 3**).

First, GUI-like APIs can not only define the behaviors of modules but also their placements on an enclosure to automatically generate support structures (e.g., holes for screws). A constant value can be declared in the code to control the kinds, numbers, layouts, and behaviors of modules on a device. The parametric design is easily achieved by exposing such values as parameters. Second, an interface builder next to the code editor is expected to narrow the mental gap between the code and resulting device. Third, layout manager APIs can help in creating physical user interfaces that follow typical design patterns.

The code-centric development process itself requires prior knowledge of programming and is targeted at interaction designers. Meanwhile, the design tool can also allow end-users to customize the parameter values in the codebase to meet their needs. This would be enable by lightweight GUI widgets, such as sliders, and the system would dynamically generate layouts and programs as well as customized instructions to assemble the device.

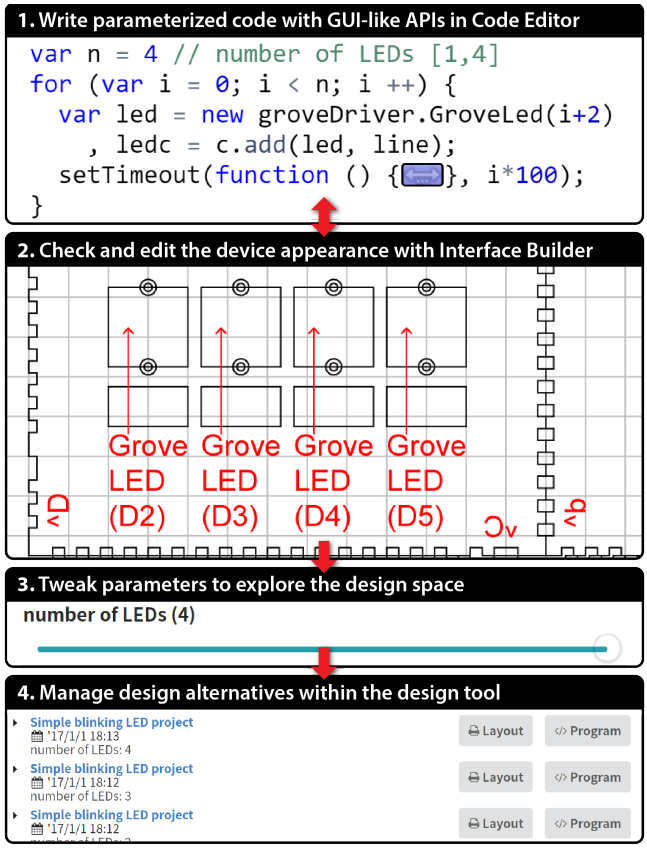

Figure 3. f3.js screenshots of code-centric development process for interaction designers (Steps 1–4) and users (3–4).

RELATED WORK

This section introduces prior work to highlight the main research contributions in our approach and the design tool.

Parametric Design of Physical Objects

The effort in generating content by specifying rules and tuning parameters is called parametric design within the field of architecture and procedural art. Notable examples include Maker Case [10], which allows us to specify the width, height, and depth of a box shape and create its development view that consists of six panels. The user does not need to worry about details such as joint matching between panels and he/she can focus on design exploration. f3.js provides similar but more detailed support for the enclosure design.

Apart from such a simple example, most tools for the parametric design of 2D and 3D models of physical objects require explicit programming. DressCode [13] allows designers to write code to generate 2D artifacts or directly manipulate the artifacts to reflect changes back in the code. Grasshopper [7] is an extension to the Rhino 3D modeling software that enables 3D models to be generated with a visual programming language. OpenSCAD [23] and ImplicitCAD [11] both provide a text-based programming environment and Shape.js [32] is a JavaScript library for generating 3D printer-ready models. Magic Box [33] provides a much simpler domain-specific language for the parametric design of box shapes. Our tool also utilizes programming to not only parameterize physical metrics but also the features of computing devices.

Some GUI tools utilize constraint-based modeling and do not necessarily require prior knowledge of programming. Typical examples are professional CAD tools such as Autodesk Inventor [2], which allow to specify constraints and interactively explore parameter space while satisfying the constraints. Direct manipulation in GUI tools eases the creation of complex shapes, but they cannot be extended to parameterize invisible parts of devices, i.e., their features.

These tools assume a single user who has a certain expertise in their usage. In contrast, Thingiverse Customizer [17] is a Web-based platform designed for a community of: 1) experts in OpenSCAD programming who create parametric 3D models and upload their source code and 2) novice end-users who use lightweight widgets such as sliders to tune the parameters to customize the models. A prior investigation [22] revealed the importance of such parametric design tools as well as design implications. f3.js is a novel application of this approach to physical computing devices, which led to the design goal noted in the previous section. It follows design implications, such as those of tracking versions, and provides reference models of existing sensor and actuator modules.

Enclosure Design of Physical Computing Devices

To the best of our knowledge, f3.js is the first tool for the parametric design of computing devices with enclosures. However, there have been prior efforts to create single versions of the enclosures.

Prior IDEs for physical computing devices have typically not provided graphical representations of the devices. Autodesk 123D Circuit [1] is one of the exceptions that has provided iconic representations of the modules and their logical connections. It has also allowed their behaviors, such as blinking LEDs, to be simulated. Such simulation is not implemented in f3.js but can be easily integrated and thus considered in future work. Microsoft .NET Gadgeteer [36] is a microcontroller and module system, whose development is well supported by the Visual Studio IDE that also has iconic representations. PaperPulse [27] goes beyond the symbolic representations and allows designers to create the actual layouts as well as the programs of paper-based applications. Programs are specified as pairs of sensor and actuator events and continuous relationships between sensor- and actuator-related values. Our tool also provides integrated support but takes the code-centric approach and allows programmers more detailed control of the device design while enabling end-user customization.

There are also external tools to IDEs that help to fabricate enclosures for devices. The .NET Gadgeteer plugin for the SolidWorks CAD tool [5] allows a Visual Studio project to be imported as 3D models of the components. Enclosed [37] serves the same purpose without using a professional CAD tool. The user specifies the position and orientation of the components and the system generates a development view of the enclosure to host them. PacCAM [30] helps the user to interactively optimize the placements of 2D shapes to be cut out on a development view to reduce material usage. Maker's Marks [24] allows to annotate physical objects with printed markers that specify the locations of corresponding components. Then, the tool scans the objects and fabricates the same objects but with holes to host the components. Our tool also utilizes the physical metrics of the components to calculate their placements with layout managers and generate support structures, and our approach can be used in conjunction with these prior methods.

Development Tools for Physical Computing Devices

Programming support for physical computing devices has been studied [8,16]. Our tool is built on top of the prior work in that they supported programming features and ours additionally supports design of the physical layout.

While this paper is aimed at supporting the development of general microcontroller applications, it should be noted that recent work such as PaperPulse [27] has focused on particular kinds of applications and reduced the amount of prior knowledge that is required. Midas [25] helps to lay out printable touch sensors on device surfaces and define events on the sensors by demonstration. RetroFab [26] is a design tool to augment existing physical user interfaces by annotating their 3D scans and generating instructions to assemble retrofitting user interfaces.

Along with recent advances in Web-based technologies, there have been various JavaScript libraries to control physical devices. Johnny-Five [14] and Cylon.js [4] abstract various kinds of microcontrollers, sensors, and actuators and provide JavaScript APIs to access them. While their applications typically run on a personal computer (PC) connected to microcontrollers, an increasing number of tiny computers and microcontrollers can execute JavaScript programs such as Intel Edison [12], Raspberry Pi [29], and Tessel.io [34].

These JavaScript libraries and platforms rely on a package manager called `npm` [21], which provides access to various kinds of APIs through unique package names. Our design tool supports the development of applications that use `npm` to load module drivers. While the current implementation has built-in support to develop JavaScript programs for Intel Edison, Raspberry Pi, and Tessel.io, it can easily be extended for use with other JavaScript libraries.

CREATING DESIGNS WITH CODE

This section introduces an overview of the tool interface and provides details of features that contribute to the design goal of easily creating physical computing devices and their design alternatives.

f3.js Design Tool Overview

As shown in **Figure 3**, f3.js provides a code editor to write JavaScript source code that produces both the layouts and programs for physical computing devices. It also provides an interface builder next to the editor just as IDEs for GUI applications do. The editor is capable of live programming [18] that continuously evaluates the code and keeps the interface builder up-to-date. The interface builder shows a development view of the device and visualizes warnings of interference between sensor and actuator modules, which are calculated from module metrics information. It also supports direct manipulation, such as the selection of shapes, the addition or removal of modules, and the dragging-and-dropping of modules to change their positions. The operations are reflected back in the code to maintain a bi-directional relationship between the code and view.

GUI-like APIs for Hardware Layouts and Software

The following subsections describe the GUI-like APIs available in the f3.js design tool. More details on the APIs can be found in the documents on the f3.js Website [6].

API for Initializing and Controlling Sensors and Actuators

f3.js does not provide APIs for controlling sensors or actuators. Instead, it relies on existing APIs of the target platforms [12,29,34] for such features. Interaction designers can benefit from their prior knowledge or learn the usage from rich resources for the existing APIs.

f3.js assumes that these APIs represent the drivers of sensors and actuators as a JavaScript class. This is akin to the APIs for GUI applications where GUI widgets are initialized by

the constructors. The main difference is that the driver instances are not used to specify how they should be mounted on the physical enclosure, which results in a mental gap between software and hardware. f3.js aims to fill the gap by addressing missing links from the driver instance to its physical instance through an `npm` package called "f3js," whose APIs are introduced below.

2D API for Path Drawing and Module Placements

The enclosure layout design in the f3.js design tool is the process of creating a tree structure of the component object model (COM, similar to the document object model in HTML), which can be rendered on an HTML canvas or as a PDF file. Sensor and actuator modules can easily be added to the COM by passing their driver instances to the `add` method of the f3.js API. It accepts optional parameters of `x`, `y`, and `rotation` to place the module at the specified location or the layout manager, as will be described later.

The APIs for drawing paths are similar to those found in 2D graphical applications such as `drawLine`, `drawRectangle`, and `drawCircle`. These methods return `Line`, `Rectangle`, and `Circle` instances, whose `stroke` and `fill` properties define whether the path should be cut or engraved with a laser cutter. More complex paths such as Bezier lines can be drawn with the `drawPath` method.

A `Container` created with the `createContainer` method can group multiple modules and paths. It has its transformation matrix that allows child modules and paths to be moved and rotated together. Unlike GUI applications, it is common in the physical user interface (PUI) to rotate shapes since the resulting device can be used in various positions and angles.

3D Modeling by Extruding 2D Paths

The panels need to have matching edges that are connected by a certain joint to assemble 3D shapes from planar panels. A simple joint shape with notches and recesses is currently supported. Although it can easily be drawn with the `drawJointLine` method, manually drawing lines often result in mismatches of the notches and recesses from both edges. It gets more complicated at the corner of boxed shapes where three joint lines need to match one another.

The `extrude` method can be called on a closed path to generate panels for constructing a 3D enclosure with a specified depth to prevent such mistakes. The panel on the opposite side can be optionally omitted (left of **Figure 4**), with the side panels not having joint lines to hold the panel (red dashed lines). The panels in the depth direction are not only generated for joint lines (**Figure 4**: red lines), but for curves and straight lines (black lines). The panels are connected to each other with joint lines, which eliminates the need for concern about matching.

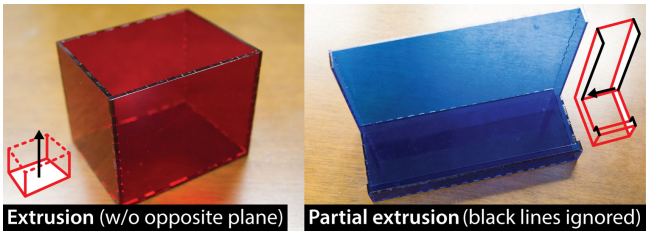

Figure 4. Example usage of extrusion methods.

f3.js assigns a numerical identifier to each joint line when evaluating the code. Then, the interface builder displays corresponding alphabetic labels next to the lines, which assists understanding of the resulting 3D shape. In addition, f3.js generates an identical notch pattern for each joint line to prevent it from being connected to the mismatching joint lines. This is a repeating binary pattern of notch-recess (0) or notch-notch (1) encoded from the numerical identifier.

Parametric Design with Lightweight GUI Widgets

One-to-another correspondence between JavaScript code and the layouts and programs of devices has already eased the creation of design alternatives. Furthermore, lightweight GUI widgets (**Figure 5**) enable an effective exploration of device variations, just as Juxtapose [9] did on an exploration of graphical and physical interaction designs.

When a variable declaration is made with a text comment in the code editor, a corresponding lightweight GUI widget is instantly populated below the interface builder. The type of the widget is dependent on the type of the initial value of the variable and the text comments.

When the GUI widgets are manipulated, the source code is edited to reflect the updated values. Manipulating the widget can affect every aspect of the device to be generated. First, the interface builder is also updated with warnings of module interferences, if any, which helps with module placements. Second, relevant files (PDF files to be sent to laser cutters and ZIP/TAR files to be installed on the target platform) are also updated, which can be downloaded by clicking on corresponding buttons.Every time the files are downloaded, corresponding design alternatives are assigned a unique numerical identifier that eases their management. f3.js records and lists all design alternatives made during the prototyping process (bottom of **Figure 3**) in this way.

Layout Managers for Flexible Layouts

According to the design patterns found in existing PUIs, our layout manager API supports the placement of modules and paths that follows design patterns (**Figure 6**).

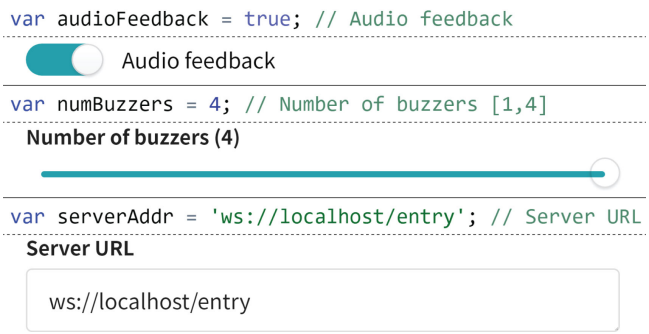

Figure 5. Variable declarations with special comments and corresponding GUI widgets.

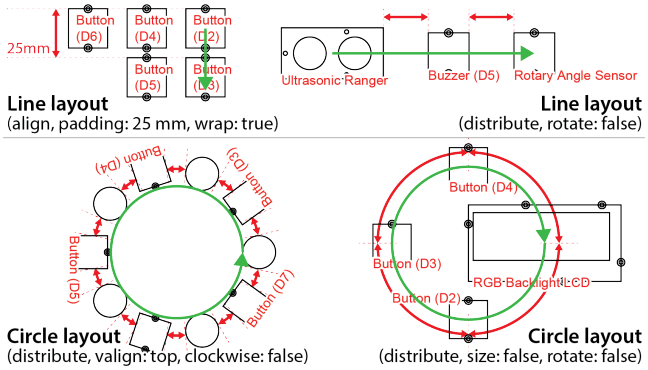

Figure 6. Example usage of PUI layout managers.

Unlike GUI layout managers that arrange components to fill rectangles, our layout manager aligns components along a specified path. We consider that this difference results from their different constraints; while GUI design is aimed at not wasting pixels in rectangular boundaries (as the name of the Java 2D API `pack()` suggests), PUI design does not need to fill spaces. The most prominent constraints are derived from users' physical properties such as their hand sizes, and it is relatively important to assign appropriate spaces.

Given the preliminary observation on PUI design patterns, the current implementation allows modules and paths to be aligned along a guide path of a line or circle with some optional parameters. The modules and paths can be aligned with fixed margins (`name: "align"`) or distributed to fill the path length (`name: "distribute"`). The vertical alignment of modules and paths against guide paths can also be chosen (`valign: "top"|"middle"|"bottom"`). The modules and paths can maintain their orientation or be rotated toward the path (`rotate: true|false`). Whether to wrap the module or path placement at the end of the line and continue onto the next line or to continue along the line regardless of its length (`wrap: true|false`) can be specified in line layout. Whether the modules and paths around the circle can be aligned clockwise or counterclockwise (`clockwise: true|false`), as well as the direction offset of the first module (`offset: n[rad]`), can also be specified in circular layout.

CUSTOMIZING DESIGNS VIA GUI

The f3.js design tool enables customization, printing, and the use of devices without prior knowledge by providing lightweight GUI widgets for customization and detailed instructions on building of devices (**Figure 7**). Building physical prototypes that perfectly match the need is *"as easy as assembling a plastic model kit"* with f3.js (quoted from a user study).

Parametric Design with Graphical User Interfaces

Users first search for existing projects. Once an interesting project is found, a project page displays its details when it is accessed. The page provides lightweight GUI widgets that allow users to interactively explore device variations.

Tweaking parameters with the widgets silently updates variable declarations in the codebase, and the print preview

Figure 7. Customization interface and instructions for users.

is consequently updated. Unlike the interface builder for interaction designers that can edit the codebase, the print preview for end-users can only be used to check the development view, which prevents unintentional changes breaking in the core functionality of the device.

Preparing Hardware Modules and Materials

There are detailed instructions on which sensor and actuator modules to purchase below the print preview. The list is dynamically generated simultaneously as the print preview is updated by aggregating the type and number of modules used in the project. The list also continues to other tools and materials such as Phillips' screwdrivers, sheets of acrylic panels, and glue for connecting the panels.

Each module name is a link to the module information page, where the description, metrics information, and relevant links, such as those for introducing specifications and for shopping, are presented. With such concrete guidance, users can confidently prepare the required materials.

Printing Layouts and Assembling Devices

Next to the list of modules is a link to download PDF files that can be directly sent to the laser cutter. It provides the option of printing labels near adjacent edges. As was discussed previously, joint lines have unique shapes that does not match with wrong edges. These features help the user to assemble the panels without confusion.

If users do not want these annotations to be engraved on the acrylic panels, they can still print them on paper and refer to them while assembling the panels without the annotations.

Installing and Running Programs

The final step is to install and run a program on the target platform. f3.js archives each project in TAR or ZIP format whose content can be directly executed on the target.

The installation consists of downloading the archive file, transferring it to the target platform, optionally installing dependencies, and launching a daemon that continuously runs the program. All the steps can be handled by a single command-line tool (`f3-projects`) that can be installed on computers with a command of `npm install -g f3js-cli`.

IMPLEMENTATION

This section briefly describes the implementation of the f3.js design tool. It is a Web-based application consisting of a Web server and an HTML/JavaScript-based client. It can be accessed with any Web standard-compliant browser on a desktop computer, tablet, or smartphone. While f3.js helps with physical computing, it is a software solution and the hardware part is handled by existing printers and physical computing toolkits.

f3.js JavaScript Interpreter

The current implementation of f3.js can be used to develop JavaScript programs for a tiny computer or microcontroller that hosts a JavaScript execution environment called Node.js. It has a de facto standard module system called `npm` for loading packaged JavaScript libraries.

The JavaScript source code in our design tool is utilized in two ways by running it on two separate interpreters, as shown in **Figure 8**. While one interpreter was implemented by us to run within the browser and render the hardware layout, the other was provided by third parties and embedded in the supported tiny computers and microcontrollers. JavaScript programming language was chosen because of its popular usage by interaction designers, its package manager `npm` is widely used to load drivers of sensors and actuators, and its high affinity with the Web technologies used for the implementation of our design tool.

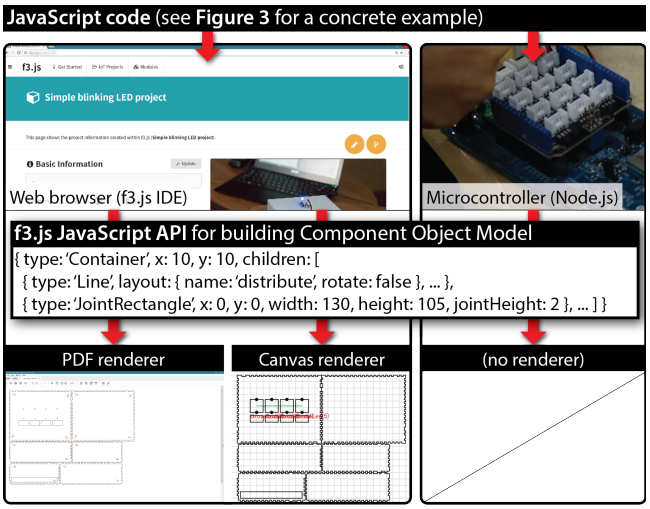

Figure 8. Single codebase runs on Web browsers for rendering and on microcontrollers for physical computing.

f3.js automatically executes the Node.js code with its own JavaScript interpreter after every edit on the code. While the interpreter is capable of running the source code written for microcontrollers, it does not load full `npm` drivers for sensor and actuator modules. Instead, when a driver class is instantiated, the interpreter returns a dummy object whose method calls nothing. As noted in the related work section, we intend to implement these methods in future work and simulate the behaviors of physical modules.

Since the interpreter currently only collects information on hardware properties, it ignores exceptions, such as calling undefined functions. The execution times out after a certain period of time (500 ms) to keep the tool responsive and addresses user bugs and incomplete code that occasionally contains infinite loops.

f3.js Sensors and Actuators Repository

f3.js utilizes the metadata of sensor and actuator modules for various purposes. This subsection introduces where they are stored, what sort of modules can be registered, and how the information is utilized.

Such metadata are not defined in the `npm` module and need to be independently stored in the repository in the f3.js Web server. The repository is shared among f3.js users, and any users (including non-programmers) can edit the metadata. While interaction designers can register their preferred sensors and actuators on the Website, those in the Grove system [31] are pre-registered. It is a modular system that does not require soldering and it supports various modules.

Sensor and actuator modules need to be represented as `npm` packages (e.g., a serial camera driver `jsupm_grovescam_js`) or JavaScript classes defined in the packages (e.g., a temperature sensor driver `GroveTemp` class defined in a driver collection package `jsupm_grove`) for them to be registered. This one-to-one relationship between the class and the physical module allows the interpreter to accumulate the kinds and numbers of physical modules that are used when the code runs.

After the code is executed, the interpreter has a reference from the instantiated JavaScript objects of modules drivers to their metadata. The metadata contain their shapes and names that allow a graphical preview to be provided and module names to be displayed in the development view. It also contains their support structures to enable screw holes that hold the modules and optional holes to be printed to expose modules or to enable cables to be inserted. This shape information is used for interference detection with other modules, which appears as a warning in the interface builder. The metadata further contain relevant uniform resource locators (URLs) such as the Websites of their manufacturers and distributors.

USER STUDIES

We conducted two user studies in different stages of developing f3.js. The first study was conducted to validate its potential and to assess its limitations by collecting a

variety of applications that could be made with f3.js. The second study was conducted after revisions to check if the tool could help both interaction designers and end-users to explore variations in devices.

First Study through Two Workshops

We conducted two workshops in a series. The participants were a total of twenty-one people made up of five interaction designers and sixteen university students.

Participants and Equipment

The first workshop was held with five interaction designers working in the same research group as the present authors. Four of them had prior experience with physical computing and one did not. All of them had intense programming experience in JavaScript to create Web-based applications.

The second workshop was held with senior university students who enrolled in a "User Interface" course. Sixteen students with varying levels of prior experience in programming (mean: 4.2 years and standard deviation (SD): 3.14) and physical computing (twelve students had no prior experience) formed nine teams by themselves. Every team had at least one student who had sufficient expertise in programming to develop JavaScript programs.

We provided the participants 1) access to the preliminary version of f3.js, 2) Intel Edison modules, 3) Grove modules compatible with Intel Edison (upon requests,) 4) acrylic panels to create enclosures, 5) screws to attach the modules on the panels, and 6) Grove cables with varied lengths to connect the modules. We also provided tools for them to create devices such as laser cutters and screw drivers.

Workshop Procedure

Each interaction designer and a group of students were asked to create their application with f3.js. First, they were given an introductory lesson on Intel Edison and f3.js for an hour. Then, they were given two weeks for implementation and device assembly. Finally, they submitted the outcomes as an archive of source code and demonstration videos and answered a questionnaire.

Workshop Results

All interaction designers and student groups successfully created their own physical computing devices. The GUI-like APIs were appreciated as they provided programming experience similar to that for GUI applications, enabling the enclosures to be designed by all participants including those without any prior use of CAD tools. They could iterate the prototyping process up to three times in two weeks by printing multiple variations of the enclosures. Multiple versions of the code (234 versions for 27 projects) were created, each of which represented a pair of the layout and program for the physical computing devices.

#	Questions regarding f3.js design tool	Mean	SD	%
1	I would like to use it frequently.	4.62	1.53	12/21
2	I found it unnecessarily complex.	2.38	0.79	20/21
3	I thought it was easy to use.	4.48	1.53	9/21
4	I needed technical support to use it.	3.62	1.68	10/21
5	I thought it was suitable for creating devices.	5.71	1.03	19/21

Table 1. Results from post-workshop questionnaire. (1 = strongly disagree and 7 = strongly agree)

Table 1 summarizes the results of the questionnaire that consisted of the mean, SD, and percentage of positive responses with scores >4 (Q1, 3, and 5) or <4 (Q2 and 4) on a 7-point Likert scale. The mean of every item denotes positive results. Most users considered f3.js to be useful and appropriate for creating devices (Q2 and 5). However, there were mixed feelings on usability (Q1, 3, and 4 with relatively large SDs). Six (Q1), six (Q3), and eight (Q4) out of 21 users awarded negative scores to these questions.

Their answers to the free text question were analyzed to understand the reasons for these low scores. Many of them were found to share the same complaints about insufficient features. Please note that the requested features had already been implemented, were included in the prior explanation of f3.js, and were evaluated in the second study.

First, most of them requested direct manipulation on the interface builder to eliminate the need to be concerned with concrete numbers that specified module locations. It is particularly cumbersome when there are multiple nested containers with transformation matrices. The current version enables drag-and-drop interaction by calculating the inverse matrix to obtain the relative movements of the modules and paths to their parent container.

Second, five of them complained about the difficulty of assembling the laser-cut panels. This motivated us to provide support for the assembly process including the assignment of different joint shapes for each edge and the display of edge labels to identify edge correspondence.

Third, four of them complained that it was tedious to manually download the archive file, extract the archive and transfer the files to the target platform. This is addressed by developing a command-line tool that automated the tasks.

Second Study with Interaction Designers and End-users

The second study was conducted after updating f3.js and it was aimed at investigating the iterative prototyping process by interaction designers and checking if the tool was usable by end-users to customize, assemble, and use devices.

Participants and Equipment

We asked three interaction designers who had participated in the first study as well as three end-users who had been newly recruited to use the f3.js design tool. All participations were voluntary who were free to leave at any time if they wanted. As with the previous study, we provided the f3.js design tool as well as all the hardware equipment required throughout the procedure.

Study Procedure

We asked the interaction designers to choose a favorite project, iteratively improve it, and expose parameters so that end-users could customize devices. We then interviewed them to obtain feedback.

After they had made revisions, we asked the end-users to customize, assemble, and use the devices. First, they were given an introductory lesson to use the f3.js Website for 15 min. Then, they were asked to choose a favorite project, change the parameters, and provide us a list of modules and a PDF file of the enclosure layout. We played the role of a shop by providing the modules and laser-cut panels. Next, they followed dynamically-generated instructions. Finally, they assembled the devices and demonstrated their use, and we interviewed them informally about their experience.

Study Results

All interaction designers had no difficulty in revising the projects. We observed iterative cycles of 1) writing the code, 2) customizing the parameters and assembling the device, and 3) installing and debugging the code within the device as their typical workflow.

The first step was already effectively supported in the first user study, and there was a recurring result. The designers appreciated its immediacy for exploring variations in device enclosures with one stating: *"writing the code and providing parameters produced the enclosure layout in no time with no explicit operations."* The layout manager API was effectively used to make the enclosure adaptable to changes in the device size and number of modules.

The second and third steps were carried out more smoothly than in the first study, due to the newly implemented assembly support and the command-line tool. The third step was particularly interesting since its debugging process sometimes involved code edits outside f3.js. Several users edited the code with their favorite text-editor and upload it to the f3.js Website with the `f3-projects` command. This was the only behavior observed outside the f3.js design tool, which reflected requests for a debugger integrated in f3.js.

The interaction designers typically parameterized features (e.g., sound on/off) during the prototyping process and types of similar sensors (e.g., slider or rotary sensors) to compare variations. Finally, they found some parameter values did not work well, and replaced the parameters with effective constant values. The lasting parameters made the devices adaptable to user environments and preferences.

All end-users could customize and create devices with such parameters. They told us that f3.js was easy to use and customization was a delightful experience that gave them greater ownership of the devices. One user built project A (described in the next subsection) with four (three by default) buzzers. Another user built project D with a customized message of "Excellent!" The last user built project C in two-player mode (one-player mode by default).

Example Projects from Studies

Figures 1 and **9** show photographs of five projects selected from the studies to showcase the variety of applications.

A. QuadBuzzer is a musical instrument with one to four buzzers. It is connected to a host PC and creates sound

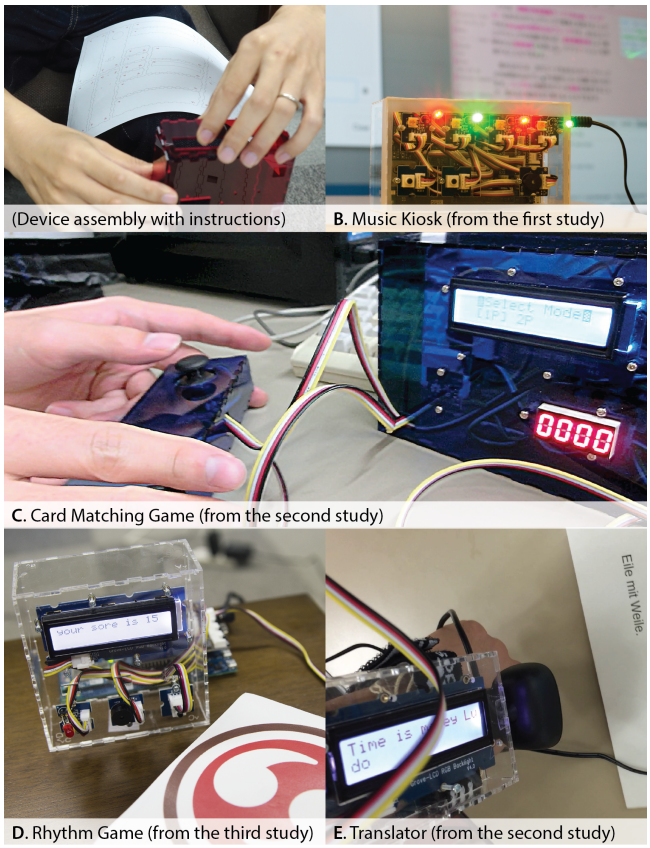

Figure 9. Projects selected from user study results.

according to user operations on the host. The enclosure is centrosymmetrical with curves and the buzzers are placed with the `Circle` layout.

B. Music Kiosk plays musical melodies in synchronicity with another computer that is playing a music video uploaded to YouTube. The device enclosure uses the `Line` layout to place four equally-spaced LEDs in line.

C. Card Matching Game allows a user to choose a number of players from one or two and changes the number of printed controllers, as well as the available game modes.

D. Rhythm Game on a Drum is an interactive game that utilizes a sound sensor to detect a hit on the drum, which is made of a hand-crafted case.

E. Translator is an arm-mounted device that uses a Web-based translation API to translate text captured by a camera and presents the results on its LCD.

DISCUSSION

We will now discuss the validity of our approach, lessons learned, and future work based on the user study results.

Creativity Support for Community of People

While we played the role of shops during the user study for the end-users, our role can be substituted with online stores. Then, f3.js can be an open-source marketplace for physical computing devices with the capability of end-user customization. f3.js, TextAlive [15] (lyrics animations), and

Thingiverse Customizer [17] (3D models) are all Web-based creativity support tools for the community of people, and it would be interesting to investigate how these tools can be extended to foster further user collaborations.

As noted in the related work section, we implemented various ideas from analysis on the Customizer [22]. Among other unimplemented ideas, enabling "remix" of projects (constructing a device with multiple features taken from existing devices) is non-trivial and interesting future work.

Benefits of Code-centric Development of Devices

Adapting to Changes – Changes in the device size and numbers and types of sensor and actuator modules often occurred during the exploration of design alternatives. The GUI-like APIs and layout managers could accommodate the changes without manually re-aligning the modules and paths. *"They made the code clean,"* an interaction designer appreciatively stated, *"combined with the parameter tuning widgets, the APIs certainly helped the parametric design."*

Layout managers in GUI toolkits play a similarly important role in the development process, but their role is slightly different from that in our case of PUI. While GUI benefit from layout managers to handle changes in window size during runtime, PUI do not change their shapes (except for emerging shape-changing interfaces [28]). PUI layout managers are primarily beneficial during the design process.

Encapsulating Details – The APIs handle cumbersome details in the design process. For instance, an interaction designer considered *"extruding support for 3D enclosures that produced multiple panels was helpful and far better than manually creating them."* A student appreciated that *"the module types are automatically detected and opening the holes to hold them was as easy as one API call."*

Standing on Top of Standards – Our APIs were built on top of an existing JavaScript ecosystem (`npm` package manager), and interaction designers and students could greatly benefit from the ecosystem. For instance, various projects used external `npm` modules to connect to the Internet and utilize the Web APIs (e.g., A, B, and E).

Computational Design for Functionalities – The f3.js APIs could be extended to provide more computational design methods. For instance, PrintedOptics [38] utilizes acrylic panels with optical fibers as sensors and displays, and its design patterns can be encapsulated as additional APIs.

Interface Builders that Complement APIs

Bi-directional Relationship – The interface builder with the code editor was *"an essential pair for the enclosure layout design."* Our result replicates the prior work emphasizing importance of the bi-directional editing between the code editor and interface builder [3,13].

3D Visualization – A student complained about the difficulty of imagining the resulting device in its 3D shape. Because the current implementation technically recognizes the 3D shape of the device being developed, its visualization should be feasible and will shortly be included in f3.js.

3D Modeling – While the current implementation is limited to designing the completely planar PUIs, the proliferation of 3D printers and various personal fabrication techniques highlight the importance of modeling freeform 3D objects in the process of PUI design. We plan to add support to 3D printers (producing 3D models instead of the development view for laser cutters) by providing APIs to design freeform 3D models such as Shape.js [32]. Inspired by faBrickation [20] that combines LEGO® blocks with 3D printed objects, combining laser-cut panels with 3D printed objects would also enables cost savings and quick printings.

Domain-Specific Language for Model-View Separation

DSL vs Shallow Embedding – It was technically feasible to use a domain-specific language (DSL) to specify the device layouts. However, we adopted a shallow embedding approach (just providing APIs in the same programming language) because we assumed that it would eliminate learning costs, ease the management of design alternatives, and increase code interoperability.

The results obtained from user studies mostly supported these assumptions. An interaction designer commented on the first two assumptions that *"writing a single codebase to specify every aspect of the device was a very simple user experience."* Editing the code outside the f3.js IDE and copying it back to the IDE in the third step of the workflow required the third assumption of interoperability.

Nevertheless, there was concern if this would scale to larger programs as *"the code for model and view resides in the same place and could be messed up when the codebase increases." "While the variable scope should be shared, it would be better if the code could be separated into two files."* Given that our APIs are used in building the COM, which is very similar to the document object model (DOM) in HTML, we could utilize similar languages for the COM.

View Code for Physical Metrics-aware Applications – f3.js currently uploads the entire source code that contains the view part to microcontrollers. While it could be excluded in the uploading process, we left the code as it was for two reasons. First, the exclusion makes it impossible to recover the original source code, reducing the code interoperability. Second and more importantly, the view code in the microcontroller makes the application aware of its physical metrics. For instance, one can imagine that the application is aware of the distance between two ultrasonic sensors and is capable of estimating the position of an object.

CONCLUSION

We proposed the code-centric development of physical computing devices that enabled them to be parametrically designed – created with code and customized via GUI. Web-based design tool, f3.js (http://f3js.org), was presented and evaluated through two user studies involving people with diverse technical background.

ACKNOWLEDGEMENTS

This work was supported in part by CREST and ACCEL, JST. We thank all the user study participants for using f3.js and providing valuable feedback, which enabled the Web release of the f3.js design tool.

REFERENCES

1. Autodesk. Autodesk 123D Circuits. http://www.123dapp.com/circuits
2. Autodesk. Autodesk Inventor. http://www.autodesk.com/products/inventor
3. Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 341-354. http://dx.doi.org/10.1145/2908080.2908103
4. Cylon.js. https://cylonjs.com
5. Dassault Systems. SolidWorks. http://www.solidworks.com/
6. f3.js | IoT apps with enclosures from single code base. http://f3js.org
7. Grasshopper. http://www.grasshopper3d.com
8. Saul Greenberg and Chester Fitchett. 2001. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology (UIST '01)*. ACM, New York, NY, USA, 209-218. http://dx.doi.org/10.1145/502348.502388
9. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology (UIST '08)*. ACM, New York, NY, USA, 91-100. http://dx.doi.org/10.1145/1449715.1449732
10. Jon Hollander. MakerCase. http://www.makercase.com
11. ImplicitCAD. http://www.implicitcad.org
12. Intel Edison. https://software.intel.com/iot/hardware/edison
13. Jennifer Jacobs and Leah Buechley. 2013. Codeable objects: computational design and digital fabrication for novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1589-1598. http://dx.doi.org/10.1145/2470654.2466211
14. Johnny-Five. http://johnny-five.io
15. Jun Kato, Tomoyasu Nakano, and Masataka Goto. 2015. TextAlive: Integrated Design Environment for Kinetic Typography. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3403-3412. http://dx.doi.org/10.1145/2702123.2702140
16. Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. 2012. Phybots: a toolkit for making robotic things. In *Proceedings of the Designing Interactive Systems Conference (DIS '12)*. ACM, New York, NY, USA, 248-257. http://dx.doi.org/10.1145/2317956.2317996
17. MakerBot. Thingiverse Customizer. http://www.thingiverse.com/apps/customizer
18. Sean McDirmid. 2007. Living it up with a live programming language. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*. ACM, New York, NY, USA, 623-638. http://dx.doi.org/10.1145/1297027.1297073
19. David A. Mellis and Leah Buechley. 2012. Case studies in the personal fabrication of electronic products. In *Proceedings of the Designing Interactive Systems Conference (DIS '12)*. ACM, New York, NY, USA, 268-277. http://dx.doi.org/10.1145/2317956.2317998
20. Stefanie Mueller, Tobias Mohr, Kerstin Guenther, Johannes Frohnhofen, and Patrick Baudisch. 2014. faBrickation: fast 3D printing of functional objects by integrating construction kit building blocks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 3827-3834. http://dx.doi.org/10.1145/2556288.2557005
21. npm, Inc. npm. https://www.npmjs.com
22. Lora Oehlberg, Wesley Willett, and Wendy E. Mackay. 2015. Patterns of Physical Design Remixing in Online Maker Communities. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 639-648. http://dx.doi.org/10.1145/2702123.2702175
23. OpenSCAD. http://www.openscad.org
24. Valkyrie Savage, Sean Follmer, Jingyi Li, and Björn Hartmann. 2015. Makers' Marks: Physical Markup for Designing and Fabricating Functional Objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 103-108. http://dx.doi.org/10.1145/2807442.2807508
25. Valkyrie Savage, Xiaohan Zhang, and Björn Hartmann. 2012. Midas: fabricating custom capacitive touch sensors to prototype interactive objects. In *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST '12)*. ACM, New York,

NY, USA, 579-588. http://dx.doi.org/10.1145/2380116.2380189

26. Raf Ramakers, Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2016. RetroFab: A Design Tool for Retrofitting Physical Interfaces using Actuators, Sensors and 3D Printing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 409-419. https://doi.org/10.1145/2858036.2858485
27. Raf Ramakers, Kashyap Todi, and Kris Luyten. 2015. PaperPulse: An Integrated Approach for Embedding Electronics in Paper Designs. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 2457-2466. http://dx.doi.org/10.1145/2702123.2702487
28. Majken K. Rasmussen, Esben W. Pedersen, Marianne G. Petersen, and Kasper Hornbæk. 2012. Shape-changing interfaces: a review of the design space and open research questions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 735-744. http://dx.doi.org/10.1145/2207676.2207781
29. Raspberry Pi. https://www.raspberrypi.org
30. Daniel Saakes, Thomas Cambazard, Jun Mitani, and Takeo Igarashi. 2013. PacCAM: material capture and interactive 2D packing for efficient material usage on CNC cutting machines. In *Proceedings of the 26th annual ACM symposium on User interface software and technology (UIST '13)*. ACM, New York, NY, USA, 441-446. http://dx.doi.org/10.1145/2501988.2501990
31. SeeedStudio. Grove System. http://www.seeedstudio.com/wiki/GROVE_System
32. Shapeways. ShapeJS. http://shapejs.shapeways.com
33. Studio Ludens. Magic Box. http://www.studioludens.com/magicbox
34. Tessel.io. https://tessel.io
35. Unity. http://unity3d.com
36. Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. 2012. .NET gadgeteer: a platform for custom devices. In *Proceedings of the 10th international conference on Pervasive Computing (Pervasive '12)*. Springer, Berlin, Heidelberg, 216-233. http://dx.doi.org/10.1007/978-3-642-31205-2_14
37. Christian Weichel, Manfred Lau, and Hans Gellersen. 2013. Enclosed: a component-centric interface for designing prototype enclosures. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction (TEI '13)*. ACM, New York, NY, USA, 215-218. http://dx.doi.org/10.1145/2460625.2460659
38. Karl Willis, Eric Brockmeyer, Scott Hudson, and Ivan Poupyrev. 2012. Printed optics: 3D printing of embedded optical elements for interactive devices. In *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST '12)*. ACM, New York, NY, USA, 589-598. http://dx.doi.org/10.1145/2380116.2380190