CraftML: 3D Modeling is Web Programming

Tom Yeh **Jeeeun Kim**
Computer Science
University of Colorado Boulder
{tom.yeh, jeeeun.kim}@colorado.edu

ABSTRACT
We explore *web programming* as a new paradigm for programmatic 3D modeling. Most existing approaches subscribe to the imperative programming paradigm. While useful, there exists a gulf of evaluation between procedural steps and the intended structure. We present CraftML, a language providing a declarative syntax where the code is the structure. CraftML offers a rich set of programming features familiar to web developers of all skill levels, such as tags, hyperlinks, document object model, cascade style sheet, JQuery, string interpolation, template engine, data injection, and scalable vector graphics. We develop an online IDE to support CraftML development, with features such as live preview, search, module import, and parameterization. Using examples and case studies, we demonstrate that CraftML offers a *low floor* for beginners to make simple designs, a *high ceiling* for experts to build complex computational models, and *wide walls* to support many application domains such as education, data physicalization, tactile graphics, assistive devices, and mechanical components.

Author Keywords
programming; 3D modeling; 3D printing; creativity support

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (HCI)

INTRODUCTION
Programmatic 3D modeling is an advanced method for designing a 3D model by expressing it as executable code. It offers several advantages over GUI-based 3D modeling tools: parameters can be given to a design to make it easy to customize; precision can be achieved through computation and logic; repeated structures can be defined by mapping data to structures; and sound software engineering approaches such as modularity, abstraction, and version control can be applied.

However, several issues with programmatic 3D modeling have been noted by practitioners. A common issue is its skill requirements. One must already be proficient in a programming language such as Python, C++, or JavaScript. Efforts exist to simplify programmatic 3D modeling by bringing it to a visual programming platform such as Scratch [28]. But such simplification often limits designers to creating simple applications; there is not a clear path to professional applications. Another issue is program interpretability. Even for skilled programmers, there exists a significant gulf of evaluation between the source code and its output. Given a piece of code, to interpret the structure it generates, one must read each step in the code and mentally simulate its effect. As a result of these issues, a great majority of 3D model designers do not use a programmatic modeling tool [7, 25]. While a growing number of parametric models can be found online, they represent only a very small percentage of all shared models.

These issues could stem from the programming paradigm adopted by most current programmatic 3D modeling tools—the imperative paradigm. In this paradigm, a designer writes a sequence of code statements where each statement specifies a modeling operation, for instance: create a sphere, scale it up by 2, or rotate it by 45 degrees. In some tools, functions and classes are available to support higher-level abstractions. While the imperative programming paradigm is adequate for a wide range of computational tasks that involve algorithmic procedures, we question how well it supports design tasks that value expressiveness and interpretability. Are there alternatives?

This paper explores the *web programming paradigm* as a possible alternative. The rise of Web 2.0 and its recent spread to a wide range of application platforms, such as mobile apps, IoT, robotics, and wearables, indicate that the web programming paradigm is becoming a dominant paradigm for developers to build interactive applications. The three "pillars" of the web programming paradigm—HTML, CSS, and Javascript, have proven to be effective for developers to programmatically specify the three aspects of an user interface—structure, aesthetics, and dynamic behaviors. To a web developer, this paradigm presents a *low floor*—easy to start writing a simple web page, while affording a *high ceiling*—possible to grow the page into a full-fledged web app. Moreover, this paradigm features a *wide wall* that supports a large number of application domains. As a result, the web developer community has grown from what used to be a small, exclusive club of elite programmers to an immensely large community of people with varying levels of technical and design skills.

We present CraftML, a new programming paradigm for 3D modeling that is based on the modern web programming paradigm. In designing CraftML, we aimed to integrate as many useful ideas from the web programming paradigm as possible, including declarative tags, hyperlinks, document object model (DOM), cascade style sheet (CSS) selectors, JQuery for DOM manipulation, string interpolation, template engine, data injection, and scalable vector graphic (SVG). In

CHI 2018, April 21–26, 2018, Montreal, QC, Canada
© 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5620-6/18/04 ...$15.00.
http://dx.doi.org/10.1145/3173574.3174101

some cases, we reached the limit of the web design metaphor and needed to introduce extensions, such as modules, STL imports, and transform/layout operators. Later we will give an overview of these programming features and explain how they help resolve the common issues in current programmatic modeling tools.

The overall structure of the paper is as follows: we review related work on current programmatic modeling tools; we describe our design philosophy behind CraftML, explaining who we target, what issues we identified, and how we plan to solve them; we present the major language features of CraftML; we describe an online IDE we built to support developing in CraftML; we provide examples and case studies as validation of CraftML's usefulness; and we discuss controversial issues behind some of our design decisions; finally, we discuss limitations and offer directions for future research.

RELATED WORKS

CraftML is designed to be a (1) low-floor, high-ceiling markup language to provide (2) programmatic 3D modeling capabilities, which can simplify (3) customization of models. Here we review these three areas of related works.

Markup Languages

Markup languages are easy to interpret and write. They have a low floor for many emergent programmers to begin coding. Some also have a high ceiling and provide a pathway for a programmer to gradually learn new features and shift to advanced levels [37]. The most popular markup languages are HTML for web development and XML for data exchange. Relevant to our work is VRML (Virtual Reality Modeling Language) for representing 3D interactive vector graphics for web-based virtual realty applications, such as reviewing architecture plans [24]. VRML's format consists of text with 3D components and associated URLs. A browser can dynamically fetch a web page or another VRML file from the internet. VRML was later succeeded by more advanced formats such as X3D and 3MF, which can encode richer semantic data in a design. More recently, A-frame [1] is a web-based programming language for designing 3D environments for VR and AR applications. It is based on an entity-component framework that provides a declarative, extensible, and composable structures. Using <script> tags, A-frame developers can also design advanced features, for example, layout models in custom pattern.

However, these formats are not designed with 3D printing in mind, where the use of 2D shapes, open curves, and open meshes are paramount [16]. Number of models are not watertight, resulting in non-manifold models that are not slice-able or detached primitives when printed. CraftML mainly targets 3D printable models, guaranteeing to be watertight.

In contrast, CraftML treats 3D solids as first-class objects, making it easy to yield watertight, 3D printable models. In terms of bringing web programming ideas into 3D modeling, MetaMorphe [39] is closely related. It lets a designer decompose an existing mesh into semantic parts (e.g., cup → {body, handle}) and writes a CSS-like expression to modify the mesh (e.g., `body { op: baloon; }`). Like MetaMorphe, CraftML takes advantage of the affordances of web markup languages to support 3D modeling.

GUI vs Programming for 3D Modeling

Most 3D modeling tools are based on a GUI. These tools are easier to learn (compared to programming) and offer the benefits of direct manipulation. Some let a designer use solids as primitives to combine them into a structure (e.g., TinkerCAD [38], SolidWorks [35]). Some let designers draw 2D sketches and convert them into 3D shapes (e.g., SketchUp [32]). And some let designers work with the mesh directly (e.g., MeshMixer [22]). Programmable 3D modeling tools, on the other hand, have a steeper learning curve [36]. But they enable a skilled designer to use computation and logic to programmatically specify a structure in a precise manner; this would be hard to achieve otherwise. Autodesk Maya [33] and Fusion360 [2] are examples of GUI-based tools offering a scripting system for designers to apply programming principles and abstraction. In terms of pure programmatic 3D modeling tools, OpenSCAD [23] is arguably the most widely adopted. It provides a set of functions to create shape primitives and apply geometric operations to them. Most programmatic 3D models shared online are written in OpenSCAD. One significant drawback of OpenSCAD is that it is not based on an existing programming language and cannot take advantage of the ecosystem of libraries or the user base of it. Thus, wrappers and derivatives of OpenSCAD have been made for Ruby (RubyScad [30]), Python (SolidPython [34]), JavaScript (OpenJScad [26]), and Scratch (BlocksCAD [6]). People already familiar with these high-level languages may find it easy to start coding 3D models and use existing libraries and tools. CraftML follows the same logic by seeking to leverage the existing ecosystem of web programming.

Supporting Customization

As designers think of a 3D printer more as a personal fabrication tool for crafting unique artifacts and less as a mass production tool for churning out identical copies, the need to support designers in remixing, reusing, and customizing models has grown considerably. The most common method is to take a parametric model and provide a GUI for end-users to specify parameters. Thingiverse Customizer [9] exemplifies a platform utilizing this method. It takes an OpenSCAD file and reads its header for parameter definitions. It then generates a GUI dialog and populates it with controls for each defined parameter. A novice user can use the dialog to generate a custom model simply by inputting parameter values, without having to touch the underlying source code. Several other programming-based 3D modeling tools, such as OpenJSCAD, Fusion360, have similar capabilities. While a GUI dialog is convenient, the range of customizability of a design is limited to the parameters the design's author chose to expose. If another user were to add a parameter to a component in the design that is not already parameterized, it would be difficult even for a skilled programmer. It would require procedurally tracing the code to find the exact line or function call responsible for generating that component.

Another customization method is to relax the requirement of being a parametric model and to work directly with a mesh. A user interactively selects a point or a region on a mesh to customize its structure. Some examples are Reprise [8] for adding adaptations such as a handle, Facade [12] for adding accessible

Braille labels, RetroFab [27] for adding an enclosure structure for actuators or sensors, PipeDream [31] for adding internal tubes, and CardBoardiZer [40] for adding articulated features. These systems allow users to interactively select a point or a region on a mesh to customize local properties such as shape and texture or to add attachments such as a handle. CraftML combines both methods. It allows for the automatic generation of a control dialog as Thingiverse Customizer does, but it makes adding new parameters easy. Because of its declarative syntax and the DOM structure it affords, one can interactively select a part in the viewer to highlight the node in the code, thereby locating the insertion point for a new parameter. This benefit will be detailed in later sections.

DESIGN PHILOSOPHY

We present a number of our design philosophy include:

Target Population. We aim to draw more new users into the practice of programmatic 3D modeling, and look to the large web developer community for opportunities. If we provided a tool sufficiently similar to web programming, would it pave an accessible path for some from this large community to enter the marvelous world of programmatic 3D modeling?

We segment web developers into four subgroups according to a typical professional skill development trajectory:

- *Beginner*: who knows the basics of HTML.
- *Template Designer*: who can use HTML and CSS to develop and style a template.
- *Programmer*: who can write Javascript to make a web page interactive and data driven, using a DOM manipulation library such as JQuery [18] or a data visualization library such as D3 [10].
- *Expert*: who can develop Web 2.0 applications, using a framework such as Angular [5].

Later we will refer back to this segmentation whenever a feature or a finding is specific to a particular subgroup.

Usability. There are three dimensions particularly important for assessing the usability of a programming paradigm—floor, ceiling, and wall. We aim to achieve the favorable end in these dimensions:

- *Low floor:* can be learned easily to build simple applications;
- *High ceiling:* can be used to develop advanced applications;
- *Wide wall:* can be used in a variety of applications.

Table 1 below compares several programming tools along these dimensions. The web programming paradigm, as represented by HTML, CSS, and Javascript, is situated at the beneficial end along all dimensions– as prior HTML style 3D modeling platform, A-Frame supports the closest design philosophy to CraftML. Our optimism is that by leveraging the web paradigm, our tool will inherit these useful properties. Some tools also (marked as †) provide programming based development tools for advanced users (i.e. script [4] and APIs [3]) but present a steep learning curve. Later in the evaluation and case studies we will provide evidence to support this optimism.

Tool	Floor	Ceiling	Wall
HTML/CSS/JS	*low	*high	*wide
Scratch [28]	*low	low	*wide
TinkerCAD [38]	*low	low	narrow
Rhino [29]†	high	*high	medium
SolidWorks [35]	high	*high	medium
SketchUp [32]	*low	*high	narrow
OpenSCAD [23]	high	*high	narrow
AutoCAD/Maya [33]†	high	*high	*wide
Fusion360 [2]†	high	*high	medium
A-Frame [1]	*low	*high	*wide

Table 1. Comparison of design tools. Features considered most favorable are starred.

Challenges. By applying the web programming paradigm to 3D modeling, we aim to address issues with procedural 3D model programming. To collect these issues, we surveyed the literature (see Related Works), taught OpenSCAD in an undergraduate computer science class and solicited feedback from students, analyzed posts seeking help in forums such as Stack Overflow, and built research prototypes using OpenSCAD, Sketchup scripting, and Fusion360. Table 2 below summarizes these issues and lists CraftML's features as a potentially viable solution.

Issue	Solution
hard to read code	declarative syntax
hard to align objects	structural tags, layout operators
hard to reuse code	link, part, module
hard to select a part to apply operations	CSS selector, JQuery
hard to refactor	template, string interpolation
hard to 3D print	water-tight solids, CSG operation

Table 2. A number of challenges identified from existing 3D modeling/programming tools, and solutions that CraftML provides.

LANGUAGE

Basic Features

Basic features are suitable for *beginner* web developers to quickly learn and create simple designs.

Primitives

Shape primitives are the most basic building blocks (as in most other 3D modeling tools). CraftML provides 3D primitives such as cubes, spheres, cylinders, prisms, polyhedrons, and 1D/2D primitives such as rectangles, triangles, and points. A primitive is specified as a pair of open and close tags or a self-closing tag. Each primitive type has a set of attributes for customization. The example below defines a cube of the default size and a sphere with a radius of 20.

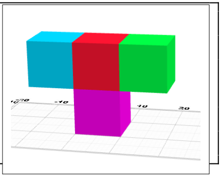
```
<cube/>
<sphere radius="20"/>
```

Structure

Aligning objects in 3D scenes is one of the most difficult tasks for users regardless of their skill level [14]. It is fairly common for manual alignment to introduce small gaps or offsets that are difficult to notice. Professional tools such as SolidWorks and Fusion360 provide four different view-ports for designers to visually inspect whether components are correctly aligned in all axes in order to minimize alignment errors.

CraftML provides a set of structural tags to achieve precise alignment of elements. They are <row>, <col>, and <stack>, which are containers that can automatically align its children along the *x*, *y*, and *z* axis respectively. Each tag also provides attributes, such as *spacing*, for customizing the alignment behaviors. The example code below uses structure tags to arrange a set of four cubes into a standing *T* formation, by declaring "a row of three cubes stacked on top of another cube."

```
<stack>
  <row>
    <cube/><cube/><cube/>
  </row>
  <cube/>
</stack>
```



Text

CraftML supports a set of common HTML tags for creating textual elements, such as <div>, <h1>, and <h2>. Textual elements, when introduced, are defaulted to 3D shapes with a thickness of 1*mm*. For accessibility support, CraftML provides <braille> for creating braille text. The following example creates "Hello World" in both English and Braille:

```
<div>Hello World</div>
<braille>Hello World</braille>
```

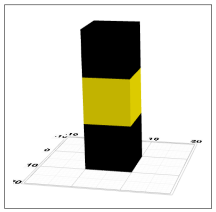
Id/Class

In imperative programmatic 3D modeling (e.g., OpenSCAD), once a structure is generated it is often difficult to then choose a particular piece of the structure on which to perform some operation. CraftML solves this problem by borrowing the concept of CSS selectors, using tag names, id, and class labels. After declaring a structure, one can easily annotate the structure using id and class labels, just like in web development. Later we will explain how CSS selectors can play an important role in many operations, including styling, transformation, layout, and scripting.

Style

CraftML supports using <style> for designers to attach an inline stylesheet to a design. A model is represented as a hierarchy of nodes, just like the Document Object Model (DOM). We can select certain nodes in this hierarchy by their id or class and apply the computed style to them. The example below uses a stylesheet to set the middle cube gold and the others black.

```
<style>
    #middle { color: gold; }
    .other { color: black; }
</style>
<stack>
  <cube class="other"/>
  <cube id="middle"/>
  <cube class="other"/>
</stack>
```



While the majority of 3D prints remain monochrome, styling has utility beyond visual aesthetics. We found a number of practical use cases for styling, such as: (1) *Debugging:* We can change the color or visibility of certain elements to temporarily focus on or ignore them; (2) *Spacing:* We can create spacing elements and set their visibility to *hidden*. Just like in web design, these hidden elements still hold their space in layout calculations; and (3) *Partial Export*: We can select a specific part of the model to export and 3D print, for example, in dual-color 3D printing applications (see Examples).

Transform

CraftML provides a set of transform operators to modify the geometry of a shape. Commonly used operators are:

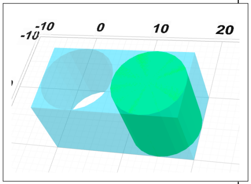
- **scale**, **translate**, and **rotate** for transformations relative to a shape's current dimensions,
- **position** and **size** for setting a shape's position and size.
- **crop** for cropping a shape into a smaller box.
- **cut** ***css-selectors*** for cutting the nodes selected by ***css-selectors*** from the rest of the nodes in the subtree.

The code below creates three unit cubes that respectively: (1) scales by 3; (2) rotates by 45 degrees w.r.t. the *x*-axis; and (3) changes its position to $(10,20,30)$, and rotates by 30 degrees w.r.t. the *z*-axis.

```
<cube t="scale 3"/>
<cube t="rotate x 45"/>
<cube t="position 10 20 30; rotate z 30"/>
```

The following example defines a cube and two cylinders. One cylinder is given an id (**id="hole"**) so that we can select it in the **cut** operator to achieve the desired effect.

```
<g t="cut #hole">
  <cube t="scale x 2"
        style="opacity:0.5"/>
  <row>
    <cylinder id="hole"/>
    <cylinder/>
  </row>
</g>
```



Layout

CraftML provides a set of layout operators to help spatially arrange the children of a node:

- **align**: aligns all children to the max side or the min side of the first child along selected dimensions.
- **center**: centers all children to the first child along selected dimensions.
- **join**: joins all children into an array-like structure where each child is next to the previous child along a selected dimension.

The example below uses layout operators to achieve the same effect as using <row> to arrange three cubes in a row.

```
<g l="join x; center yz"/>
  <cube/><cube/><cube/>
</g>
```

Imports

For beginners, it is easy to start modeling by importing and remixing parts made by others. The example below imports an octopus and a boat from an STL file and another CraftML file respectively. It arranges both objects in a row to depict the

```
<part name="octopus" module="/path/to/octopus.stl"/>
<part name="boat" module="/path/to/boat.craftml"/>
<stack>
  <octopus/><boat/>
</stack>
```

Intermediate Features

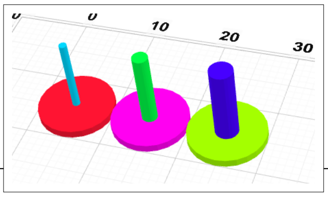
Intermediate features are suitable for web designers who are proficient in HTML and CSS and possess some limited coding skills such as the ability to develop view templates.

Part and Parameters

As a model becomes complicated, it is often desirable to decompose it into parts. CraftML provides a <part> tag for designers to define custom parts that can be reused and also to improve readability. Additionally, a custom part can carry parameters for further customization. To do so, we add a <param> tag for each parameter. Then, we replace hard-coded values using string interpolation (changing **scale 2** to **scale {{ x }}**).

The example code below defines a “pin” part which is a flat disk below a thinner cylinder. It has one parameter to control a pin’s thickness. This parameter is used to specify the *x* and *y* arguments of the *size* transform operator. Then, we can use <pin> as a custom tag to compose a larger structure. Here, we create a column of three pins with varying thickness:

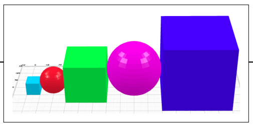
```
<part name="pin">
  <param name="thickness" default="1"/>
  <stack>
    <cylinder t="size xy {{ thickness }}"/>
    <cylinder t="size 10 10 1"/>
  </stack>
</part>
<col>
  <pin thickness="1"/>
  <pin thickness="2"/>
  <pin thickness="3"/>
</col>
```



Logic

Many popular template engines (e.g., EJS, Hogan.js, Jade) support basic programmatic logic that can be specified directly in a tag as an extra attribute. This logical attribute is often referred to as a *directive*. CraftML provides two directives: **repeat** and **if**. The former is useful for repeating a structural pattern, while the latter is useful for controlling whether or not to render a node. This example uses the list comprehension logic to create a row of five shapes with increasing sizes, alternating between a cube and a sphere.

```
<row>
  <g repeat="i in [1,2,3,4,5]" t="scale {{ i }}">
    <cube if="i % 2 === 1"/>
    <sphere if="i % 2 === 0"/>
  </g>
</row>
```



Advanced Features

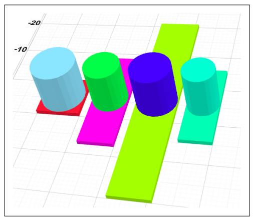
Here we introduce advanced features intended to enable expert developers to build sophisticated models or to develop useful extensions.

Scripting

Although CraftML provides a rich vocabulary of tags and transformation/layout commands to meet a wide range of 3D modeling needs, there are situations when experts want to have more control and increase complexity. For instance, an expert may wish to lay out elements in a custom wave pattern, or enlarge any element that is smaller than 20*mm* on the *x*-axis, or generate a computational structure to represent a given data set. CraftML allows custom scripting by writing JavaScript code inside of a <script> block.

Within each script block, a special local variable **$params** is made available for a programmer to manipulate the parameters in the scope. A programmer can add, modify, or delete parameters. The modification will then be visible to all the subsequent sibling nodes and their descendents. In the example below, we define a data array of two objects; each has attributes *a* and *b* and we write a *for* loop to preprocess the data. This array is assigned to **$params** as a new property. Then, the data becomes available to the next sibling (<row>) that defines a row of cubes whose *x* and *y* dimensions are mapped to *a* and *b*. Atop each cube is a cylinder whose diameter is mapped to *b*.

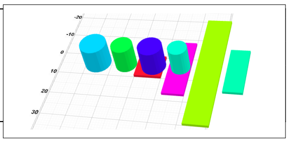
```
<script>
  var data = [{a:5, b:20}, {a:3, b:50}, {a:4, b:100},
              {a:2, b:40}];
  for (var i = 0; i < data.length; i++){
    data[i].a += 5;
    data[i].b /= 2;
  }
  $params.data = data;
</script>
<row spacing="2">
  <stack repeat="d in data">
    <cylinder
        t="size xy {{ d.a }}"/>
    <cube
        t="size {{ d.a }} {{ d.b }} 1"/>
  </stack>
</row>
```

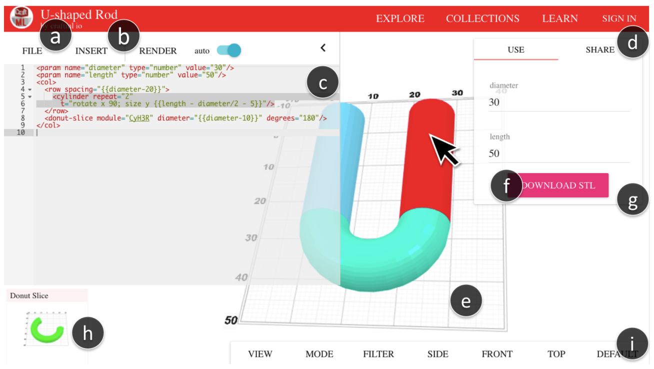


JQuery

JQuery is one of the most popular Javascript libraries for programming dynamic web pages by manipulating DOM. CraftML provides a special variable **$** for selecting and modifying certain nodes in the model hierarchy, just like the typical JQuery symbol in web programming. Continuing from the previous example, suppose we want to move *only* the cubes to the right by 20. The script below accomplishes this by first selecting all the cubes who are the children of a stack (line 2) and evaluating a transformation expression to translate the selected cubes along *x* (line 3).

```
1 <script>
2   $('stack > cube')
3     .t('translate x 20');
4 </script>
```





```
1   <param name="diameter" type="number" value="30"/>
2   <param name="length" type="number" value="50"/>
3   <col>
4     <row spacing="{{diameter-20}}">
5       <cylinder repeat="2"
6         t="rotate x 90; size y {{length - diameter/2 - 5}}"/>
7     </row>
8     <donut-slice module="CyH3R" diameter="{{diameter-10}}" degrees="180"/>
9   </col>
10
```

diameter
30

length
50

Figure 1. Cloud-based IDE for CraftML users to develop, debug, and share their designs

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

To support CraftML users to develop, debug, and share their designs, we developed a cloud-based integrated development environment (IDE). Figure 1 shows a screenshot of this IDE. The design displayed in the IDE is a U-shaped rod composed of three parts: two cylinders and a half donut. We use a scenario to walk thorough the key features of this IDE.

Coding and Live-Previewing

For Bob, making cylinders is easy. He writes a <cylinder> and adds a **repeat** directive to get two copies. He wraps it with a <row> and specifies **spacing** to separate the two cylinders along the *x*-axis. All the while, he sees the rendered model updated automatically each time he makes an edit (Figure 1.e). Bob also has an option to turn off this automatic rendering feature (Figure 1.b). In the literature, the ability to receive live feedback is shown to significantly reduce syntactic and semantic errors [15].

Searching and Importing a Module

Bob needs another half donut to complete his design. He presses the "INSERT" button (Figure 1.a) and a dialog opens. This dialog is a search interface for all of the models which are published on the platform. He enters "donut" as a search term. A search engine retrieves a list of design documents containing this term. In some documents, the term appears in the titles given by the original authors as meta-data. Most search engines for 3D models support similar meta-data search. But here, Bob also sees many other documents where the search term appears as a part of the code. For instance, a user explicitly specified <donut>. This case arises when: (1) a user defined a custom part and called it "donut", or (2) a user imported a design and named it "donut." Either way, Bob learns a few things about how to build a donut structure in the former case and how to use an existing donut design in the latter case. He decides to simply import an existing donut design (Figure 1.h).

Bob's experience here highlights a benefit of taking a declarative approach—we get content-based search *almost* for free. We found when using CraftML, designers are more likely to use meaningful tag names, IDs, and class labels to define local structures while they are designing, as opposed to assigning meta-data after they finished the design as an after thought.

Currently, the search function in our prototype is provided by ElasticSearch [11]. There exists even more potential for content-based search of CraftML models. One future research avenue is to explore structural search wherein user apply commands such as "find me models containing a row of at least three cylinders" in a similar vein as Webzeitgeist [21] for searching web design patterns.

Parametrization / Sharing / Exporting

Bob wants to share his design with others. But instead of a fixed design, he wants to add parameters to make it easy to customize. He adds two <param> tags: one to control the diameter of the rod and the other to control the length. As soon as he defines the parameters in the code, the IDE detects them and automatically generates a form populated with input controls, one for each parameter (Figure 1.g). Bob creates a shared link (Figure 1.d) and emails it to his friend, Matt. Matt opens the link and accesses the design. He finds the form and begins to enter values to customize the design to his imagination. Each time he enters a new value, the model automatically updates. Finally, Matt is satisfied with his variations and presses the big pink download button to get an STL file (Figure 1.f).

Code Highlighting / Learning / Debugging

Bob makes his design public. In a distant part of the world, Jane discovers the design and opens it up in her browser. She uses several viewing features provided by the IDE (Figure 1.i) to move the model for inspection. As her mouse cursor hovers over a part (red dowel in Figure 1), the part is highlighted along with the corresponding lines in the code(Figure 1c, line5-6). Being able to see this correspondence, Jane quickly learns the composition and logic underlying Bob's design. Jane even spots an error. The cylinders are slightly misaligned. With the help of code highlighting , Jane quickly finds the source of the error and fixes it.

EXAMPLES

Floor to Ceiling

CraftML offers a development path from simple to sophisticated as users become more experienced with its features. We present a five step scenario to exemplify this path, wherein each step builds upon the previous one. Figure 2 shows the evolving design along the way.

Step 1: Lucy, a *beginner* web developer, writes her first "Hello World" program:

```
Hello World
```

Step 2: Lucy learns to use basic primitives, structural tags, and transformation operators. She writes the simple code below to create her name tag. She downloads an STL file and 3D prints it.

```
<stack>
  <div>Hello Lucy</div>
  <cube t="size 50 10 2"/>
</stack>
```

Step 3: As Lucy increases her web development experiences, she now considers herself to be a competent web *template*

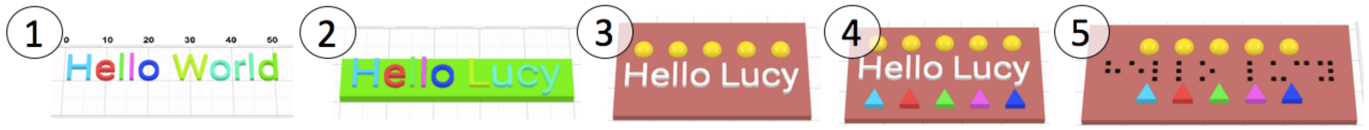


Figure 2. Example of building from the simplest "Hello World" design to more complex designs.

designer. She has learned how to write a stylesheet and can design templates using mustache expressions (i.e., {{ }}). Also, she has increased competency with simple logical directives such as **repeat**. She now adds contrasting colors and a stripe of five domes to decorate her name tag. She adds a parameter for others to customize the name tag with their own name. The source code of her new design now reads as follows:

```
1  <param name="name" value="Lucy"/>
2  <style>
3    div { color: white; }
4    #board { color: brown; }
5    dome { color: gold; }
6  </style>
7  <stack>
8    <col spacing="2">
9      <row spacing="4">
10       <g repeat="5" t="size 5 5 2">
11         <dome/>
12       </g>
13     </row>
14     <div>Hello {{ name }}</div>
15   </col>
16   <cube id="board" t="size 50 30 2"/>
17 </stack>
```

Step 4: Lucy's skills have grown rapidly and she has begun to identify as a *programmer*. She has learned about refactoring and can design complex templates. She is also getting accustomed with JQuery. She updates her design to:

```
1  ...
2  <part name="stripe">
3    <row spacing="4">
4      <g repeat="5" t="size 5 5 2">
5        <content/>
6      </g>
7    </row>
8  </part>
9
10 <stack>
11   <col spacing="2" id="aboutme">
12     <stripe> <dome/> </stripe>
13     <div>Hello {{ name }}</div>
14     <stripe> <prism/> </stripe>
15   </col>
16   <cube id="board"/>
17   <script>
18     $params.w  = $('#aboutme').size().x
19     $('#board').t("size {{ w + 10 }} 30 2")
20   </script>
21 </stack>
```

Intending to create two stripes instead of one, Lucy defines a new part called "stripe" to encapsulate the logic for creating a row of shapes (lines 2-8). She uses <stripe> twice (lines 12, 14), each contains a different shape to repeat—a dome and a prism. The previous design has a bug; if someone's name is too long, the board is not wide enough. She resolves this bug by writing a script that calculates the width of the content (line 18) and dynamically sets the board's width to fit it (line 19).

Step 5: Lucy wants to make a name tag for her friend who is blind. She changes line 13 to the following:

```
1    <braille>Hello {{ name }}</braille>
```

Now she has a Braille version of the design. She 3D prints a copy and gifts it to her friend.

Wide Walls

CraftML offers wide walls for a range of modeling applications. To demonstrate this, we present eight distinct application categories with selected examples [1]

Characters: Even with simple primitives as building blocks, it is possible to create artistic characters. Figure 3 shows an Angry Bird composed of spheres and cones; a Minion who wears a pair of goggles, which is an imported STL file; a Charmander composed of a pixel-based design; the Linux Tux with organic curves made by smoothing between primitives; and Homer Simpson whose hair is actually a flipped 'W.'

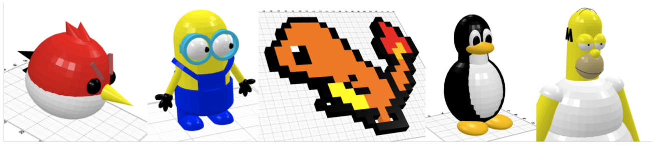


Figure 3. Characters: AngryBird, Minion, Charmander, Linux Tux, Homer Simpson.

Assistive Devices: 3D printing is a promising method to accommodate physical constraints and limitations by bridging gaps between digital designs and physical objects. For example, if a door handle is too small for a user's grip, a user can 3D print a lever to augment the handle to make it easier to open. However, assistive augmentation designs on Thingiverse are mostly static and non-parametric [7]. Figure 4 shows an assistive door handle lever with customizable handle length and loop radius; a zipper pull with customizable width and thickness; a cup holder with a customizable size; a graspable bottle opener with a variable number of flanges; and a customizable corner cover to protect a toddler from sharp furniture corners. Researchers also used CraftML to turn non-parametric off-the-shelf assistive device models from online to be parametric, by implanting modular adjustors [20].

Fractals: A fractal is an abstract object used to describe and simulate iteratively occurring objects. Implementing a fractal typically involves recursion, which is difficult to do in a GUI-based modeling tool. CraftML allows a part to recursively

[1]Hyperlinks in figure captions refer to the CraftML source code.

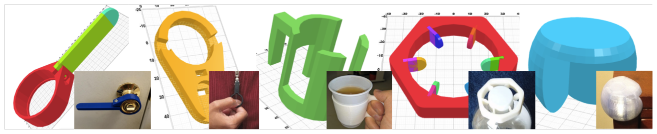


Figure 4. Assistive Devices: lever, zipper pull, cup holder, grasable bottle opener, furniture corner cover

include itself and provides the **if** logic directive to check for a terminating condition—two necessary ingredients for coding a fractal. Figure 5 shows an H-tree, a heart emblem, a Menger sponge, a randomly generated snowflake, and a candle tower.

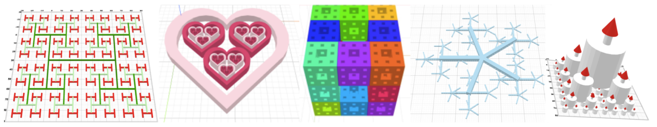


Figure 5. Fractals: h-tree, heart emblem, Menger sponge, snowflake, candle tower.

Mechanical Components: Parametric and reusable mechanical components are good candidates for modeling in CraftML. Figure 6 shows examples of parametric gearboxes written in CraftML. They can be remixed with other 3D models. These gearboxes are exposed to users as modules just like any other CraftML modules. Users can customize aspects such as the size of a gear or the length of a shaft by specifying attribute values.

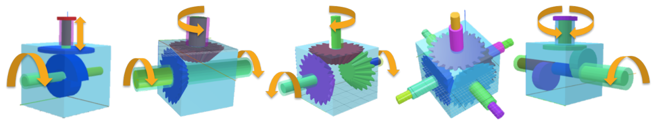


Figure 6. Mechanical Components: crank, bevel gear, friction gear, multi gears, double cam. Arrows were added to indicate the mobility of each unit.

Education Manipulatives: 3D printing has been used in STEM education as a way to turn abstract, invisible concepts into a tangible representation one can see and touch [19]. Figure 7 shows models of a hydrogen emission spectrum, a methane molecule, an oxizen molecule, a cell membrance, and a plant cell. .

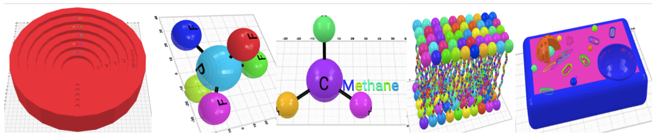


Figure 7. Education Manipulatives: hydrogen emission spectrum, methane, oxizen (generated from the same parametric design as the previous), cell membrane, plant cell.

Data Physicalization: Physical representations of data produced by digital fabrication are shown to help people explore, understand, and communicate data in a tangible and accessible manner [17, 19]. CraftML makes programming data physicalization designs similar to programming web-based data visualizations using libraries such as D3 [10]. A programmer imports a data array and defines how to map each element in the array to a structural element. In D3, a digital bar chart is a result of mapping data values to the heights of a series of rectangles. In CraftML, a physical bar chart can be similarly defined by mapping data values to the heights of a row of cubes. Figure 8 shows examples of 3D printable versions of common chart types developed using CraftML. All the designs are parametric and customizable with new data.

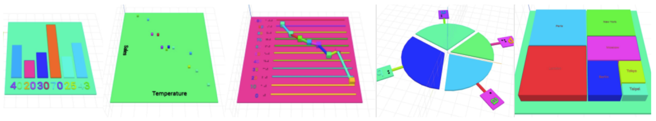


Figure 8. Data Physicalization: barchart, scatter plot, line chart, pie chart, tree map.

Dual-material printing: Designing a model for dual-materials or colors is challenging; most slicing tools for multiple extruders require the exact same number of separate input files in order to assign them to each extruder head. CraftML provides the capability to download individual segments of a model simply using a CSS selector. Figure 9 lists five examples, which were successfully printed. For example, in the red-white twisted thread example, a user may assign class labels ‘white’ and ‘red’ to each color region. In turn, the user can select one segment using ‘.white‘ and the other segment using ‘.red’ to obtain two separate STL files, which can then be sent to a 3D printer.

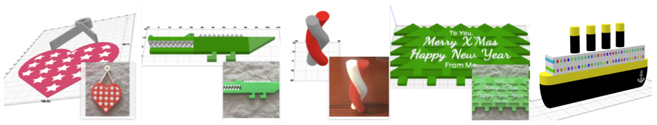


Figure 9. Dual-material Printing: stars in a heart, alligator clip, twisted thread, Christmas card, titanic.

CASE STUDIES

Youth Summer Camp

Instructors of a summer camp experimented with adding CraftML into their existing 3D printing lesson plan. It was added on the 4th day of a week-long program. In a 3 hour session, a group of 9 to 12-year-old children ($N = 8$) learned to use CraftML to make a “homepage” by writing simple tags to define a page and then importing existing models to include in the page (Figure 10). They also learned to design simple models by first building a LEGO representation and then breaking it down into a hierarchy of primitives. They learned how to describe the hierarchy using various tags. One particularly effective example was a “snowman.” Most children could instinctively relate a snowman to a geometric abstraction of a stack with three cubes. Given partially completed code of a stack with two cubes, most children could correctly interpret the code and write another correctly placed `<cube>` to make a snowman. However, some required staff assistance to fix

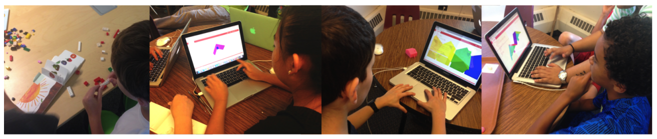


Figure 10. CraftML was taught in a youth summer camp.

syntax errors such as to properly open and close tags. Thus, the result of this experiment was mixed. While young designers were definitely exposed to the benefits of "coding" a design, we consider whether being also exposed to the perils of real-world coding (syntax errors) is beneficial.

Tactile Picture Book Project

Tactile Picture Book Project (TPBP) is an initiative to design and give accessible 3D printed story books to children with visual impairments [36]. The TPBP team works with families across the world to create a custom book for their children. They take specifications from families and make changes to individual pages, such as what braille text to include on each page or the minimum margins between elements. Initially, the team used existing tools to make changes but found the process rather difficult and time consuming. For example, to change the Braille text in TinkerCAD, they needed to manually move "dots" and do that for every page. After transitioning to CraftML, the team made a common template shared across pages and books. Each template has places to put Braille and tactile graphical contents. Also, they added parameters for several aspects of their designs, such as the number of animals, the number of stars in the sky, and the spacing along the borders. Customization became easier; they simply entered parameter values via a form.

While CraftML continues to augment TPBP's workflow, there exist limitations. We observed that the TPBP team did not use CraftML to model all elements. For textual and structural elements, such as Braille, rainbows, and repeated waves, they wrote CraftML code. But for natural elements such as giraffes, they opted to find an existing STL file on Thingiverse or to use a GUI-based tool to first sculpt it. Then they imported the premade model into CraftML and applied various transformation and layout operators to customize it. This design practice is similar to web design. A designer prepares images in another tool and embeds them in an HTML document.

Designing Modular Prosthetic

A researcher on asssitive technology wanted to contribute to the eNable [13] initiative by designing a modular 3D printable prosthetic hand. One particular issue this researcher intended to tackle was the difficulty in customizing the end-effector of a prosthetic hand. The researcher took inspiration from the design of the Python Utility Hand as seen in Figure 11 (left). This design consists of a gauntlet and a number of interchangeable effectors, such as a cup holder and a card holder. But this existing design is not parametric. One needs to load the file into a CAD software and manually edits the model. Using CraftML, the researcher reverse-engineered this design and introduced parameters to control various dimensions of the design. Also, a set of compatible and customizable effectors were developed. The researcher provided a module that encapsulates all the logic of attaching an effector to a gauntlet. Others can create a custom hand by writing the code below:

```
<attach module="...">
    <cup-holder modulde="..." diameter="100"/>
    <gauntlet module="..." length="350"/>
</attach>
```

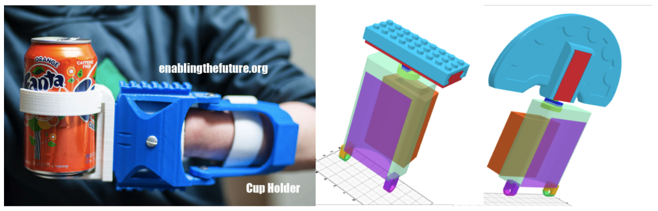


Figure 11. CraftML was used to reverse engineer the python utility hand (left) to a modular, parametric design (right).

Examples of custom gauntlet designs generated by this code are shown in Figure 11 (right). This case study illustrates the potential of CraftML for designers to revisit certain useful designs originally created in a CAD tool and reverse-engineer it to obtain a parametric, customizable version.

Teaching Web Programming in an HCI Class

Instructors of an introductory HCI course wanted their students to gain exposure to basic web programming (HTML/CSS). One issue they encountered was inequality in students' technical background; some were already familiar with HTML/CSS, while others had no experience. Typical tutorials for basic web programming would be beneficial to some students but boring to the others. The instructors experimented with using CraftML in the class. Two homework assignments were designed as an equalizer for all students to understand nested tags and basic CSS styling. Regardless of prior background, students had an opportunity to learn something novel and useful. The first homework was for each student to create a name tag. The second homework was to create a landmark in each student's favorite city. Almost all students were able to complete their assignments (176/177). Many students exceeded the requirements. Figure 12 shows a selection of students' works, including high quality and average submissions. This case study suggests knowledge and skill can be transferred bi-directionally between CraftML and web programming.

DISCUSSION

We discuss a number of design decisions behind CraftML that could potentially generate debate.

Declarative Alternative to Extrusion. Extrusion is a key procedure in many programmatic 3D modeling tools to support the conversion of a 2D shape into a 3D shape. For instance, we take a 2D circle lying on the *x*-*y* plan, move it along the *z*-axis, and "extrude out" a 3D cylinder. However, in developing CraftML, we resisted supporting extrusion because of its strong association with the imperative programming paradigm. We challenged ourselves to find declarative alternatives that are functionally equivalent to extrusion. After experimenting with several options, we decided on the following declarative semantic: "A 3D solid is a set of [walls] built around a [layout] of [2D shapes]." For example, a sloped cylinder can be expressed declaratively as:

```
<stack t="wall">
  <circle/><circle t="translate x 10"/>
</stack>
```

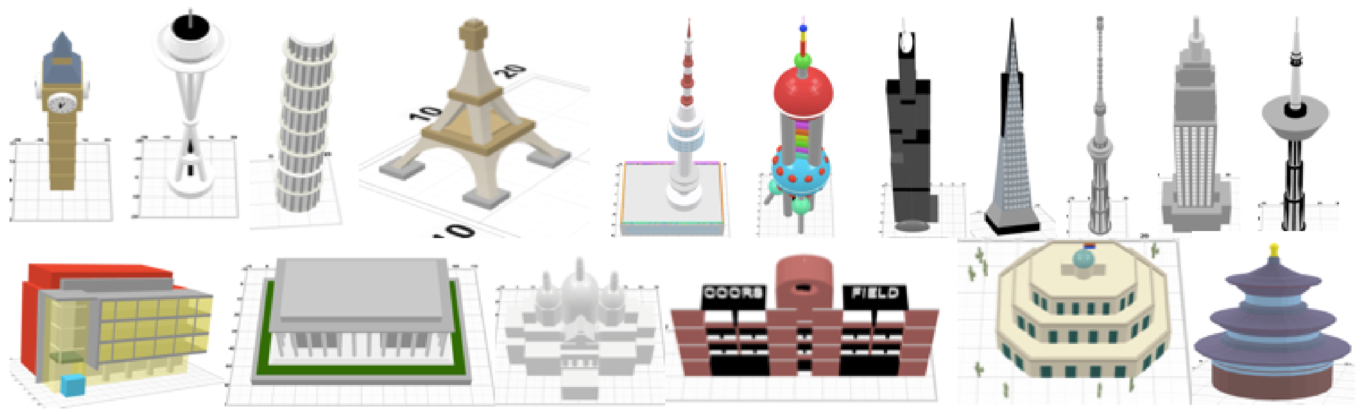


Figure 12. CraftML was taught to a large CS course as a hook for and a bridge to web programming. Selected examples of students' works are shown.

A donut, often accomplished imperatively by a "circular extrude" procedure, can be written as:

```
<part name="donut" t="wall">
  <sun-layout module="uHUlw" radius="30">
    <circle repeat="25" t="rotate x 90"/>
  </sun-layout>
</part>
```

This declares a donut as "walls around a sun layout of 25 circles." So far, we have not yet encountered situations that cannot be handled by our declarative alternative. We have seen early evidence to suggest this declarative construct is easy for web developers at all skill levels to grasp.

Inverted Y-Axis. In web design, we use a screen coordinate system where the *y*-axis is inverted. As new content nodes are added to a page, they move toward the bottom of the page and their *y*-positions increase (rather than decreasing). In developing CraftML, we aimed to offer a conceptual model consistent with web design regarding spatial relationships. Thus, we decided to adopt the screen coordinate system for the *x*-*y* plane in CraftML's design environment. This choice has some desirable effects, such as: a) when creating a 2D shape, its upper-left corner is (0, 0); and b) when writing text elements, the natural reading order (downward and right) is positive in both x and y directions. This consistency enables web developers to bring their spatial reasoning from web design directly into 3D modeling without the burden of mental inversion. Also, certain code written for the web can be copied directly into CraftML without any coordinate flipping (e.g., `<div>`, `<path>`). However, the decision to adopt inverted *y*-axis did not come easily. We knew we would break tradition with nearly every 3D modeling tool where the *y*-axis is not inverted. In the spirit of our research initiative, we ultimately adhered to the convention followed by nearly every web design tool.

LIMITATIONS & FUTURE WORK

While CraftML inherits several benefits from the web programming paradigm, it also inherits some issues:

Broken Links. In order to encourage content sharing and remixing, CraftML makes it easy to link to other design files with limited restrictions. When a design file is rendered, our rendering engine pulls in the latest version of all upstream dependencies dynamically. One benefit is that if an upstream design is improved, such improvement is automatically propagated downstream to every design that links to it. However, there exists a consequence. If an update to a design introduces an error, breaks certain assumptions, or a design is deleted, all downstream designs would be similarly broken. To deal with this issue, we need to create a robust versioning system, such as: (a) allowing users to take a snapshot (sacrificing automatic update); or (b) implementing a semantic versioning system popularized (increasing the workload for export/import modules).

Attributions. The health of a creative platform depends not only on an easy mechanism to share and reuse but also on a proper way to handle attributions. In web design, attributions to linked resources operate like an honor system. Designers are not forced to include attributions and may do so out of courtesy. Currently, CraftML's online platform has a limited attribution mechanism. It assumes the most generous creative-common license. and keeps track of a history of cloning and forking. When an external design is linked, the online editor prominently displays its source and author. However, more can be done, such as: requiring users to credit each linked source, allowing designers to pick a license, and adding constraints to module linking to respect such license.

CONCLUSION

In this work, we set out to explore the web programming paradigm as a viable alternative to the imperative programming paradigm for programmatic 3D modeling. We developed a new language, CraftML, that offers a rich set of programming constructs familiar to web developers, such as declarative syntax, semantic tags, CSS, Javascript, JQuery, string interpolation, and templates. We showed examples and case studies as supporting evidence for concluding that the web programming paradigm is indeed a viable and beneficial alternative.

ACKNOWLEDGMENTS

We thank Tactile Picture Books Team from University of Colorado Boulder, for their creative endeavors to enhance CraftML. This research is supported by NSF Grant No. IIS-1453771.

REFERENCES

1. A-Frame. https://aframe.io/.
2. Autodesk Fusion360. https://www.autodesk.com/products/fusion-360/overview.
3. Autodesk Fusion360 API. https://autodeskfusion360.github.io/.
4. Rhino and Grasshopper Developer Documentation. http://developer.rhino3d.com/.
5. AngularJS. https://angularjs.org/.
6. BlocksCAD. https://www.blockscad3d.com/.
7. Erin Buehler, Stacy Branham, Abdullah Ali, Jeremy J. Chang, Megan Kelly Hofmann, Amy Hurst, and Shaun K. Kane. 2015. Sharing is Caring: Assistive Technology Designs on Thingiverse. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 525–534. DOI: http://dx.doi.org/10.1145/2702123.2702525
8. Xiang 'Anthony' Chen, Jeeeun Kim, Jennifer Mankoff, Tovi Grossman, Stelian Coros, and Scott E. Hudson. 2016. Reprise: A Design Tool for Specifying, Generating, and Customizing 3D Printable Adaptations on Everyday Objects. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 29–39. DOI: http://dx.doi.org/10.1145/2984511.2984512
9. Thingiverse Customizer. https://customizer.makerbot.com/.
10. D3. https://d3js.org/.
11. ElasticSearch. https://www.elastic.co/.
12. Anhong Guo, Jeeeun Kim, Xiang 'Anthony' Chen, Tom Yeh, Scott E. Hudson, Jennifer Mankoff, and Jeffrey P. Bigham. 2017. Facade: Auto-generating Tactile Interfaces to Appliances. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 5826–5838. DOI: http://dx.doi.org/10.1145/3025453.3025845
13. Enabling The Future âĂŞ A Global Network Of Passionate Volunteers Using 3D Printing To Give The World A 'Helping Hand.'. 2017. http://enablingthefuture.org/.
14. Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. 2016. Understanding Newcomers to 3D Printing: Motivations, Workflows, and Barriers of Casual Makers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 384–396. DOI: http://dx.doi.org/10.1145/2858036.2858266
15. Christopher D. Hundhausen and Jonathan Lee Brown. 2007. An experimental study of the impact of visual semantic feedback on novice programming. *Journal of Visual Languages Computing* 18, 6 (2007), 537 – 559. DOI:http://dx.doi.org/https://doi.org/10.1016/j.jvlc.2006.09.001
16. Jacek Jankowski, Izabela Irzynska, Bill McDaniel, and Stefan Decker. 2009. 2LIPGarden: 3D Hypermedia for Everyone. In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia (HT '09)*. ACM, New York, NY, USA, 129–134. DOI: http://dx.doi.org/10.1145/1557914.1557938
17. Yvonne Jansen, Pierre Dragicevic, Petra Isenberg, Jason Alexander, Abhijit Karnik, Johan Kildal, Sriram Subramanian, and Kasper Hornbæk. 2015. Opportunities and Challenges for Data Physicalization. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3227–3236. DOI: http://dx.doi.org/10.1145/2702123.2702180
18. JQuery. https://jquery.com/.
19. Shaun K. Kane and Jeffrey P. Bigham. 2014. Tracking @Stemxcomet: Teaching Programming to Blind Students via 3D Printing, Crisis Management, and Twitter. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 247–252. DOI: http://dx.doi.org/10.1145/2538862.2538975
20. Jeeeun Kim, Anhong Guo, Tom Yeh, Scott E. Hudson, and Jennifer Mankoff. 2017. Understanding Uncertainty in Measurement and Accommodating Its Impact in 3D Modeling and Printing. In *Proceedings of the 2017 Conference on Designing Interactive Systems (DIS '17)*. ACM, New York, NY, USA, 1067–1078. DOI: http://dx.doi.org/10.1145/3064663.3064690
21. Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3083–3092. DOI: http://dx.doi.org/10.1145/2470654.2466420
22. Meshmixer. http://www.meshmixer.com/.
23. OpenSCAD The Programmers Solid 3D CAD Modeller. http://www.openscad.org/.
24. Pietro Murano and Dino Mackey. 2007. Usefulness of VRML building models in a direction finding context. *Interacting with Computers* 19, 3 (2007), 305–313. DOI: http://dx.doi.org/10.1016/j.intcom.2007.01.005
25. 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. DOI:http://dx.doi.org/10.1145/2702123.2702175
26. OopenJSCAD. https://openjscad.org/.
27. 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. DOI: http://dx.doi.org/10.1145/2858036.2858485

28. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. DOI:http://dx.doi.org/10.1145/1592761.1592779

29. Rhinoceros. https://www.rhino3d.com/.

30. RubyScad. https://github.com/cjbissonnette/RubyScad.

31. Valkyrie Savage, Ryan Schmidt, Tovi Grossman, George Fitzmaurice, and Björn Hartmann. 2014. A Series of Tubes: Adding Interactivity to 3D Prints Using Internal Pipes. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 3–12. DOI: http://dx.doi.org/10.1145/2642918.2647374

32. SketchUp. https://www.sketchup.com/.

33. Maya: Computer Animation & Modeling Software., Autodesk. https://www.autodesk.com/products/maya/overview.

34. SolidPython. https://github.com/SolidCode/SolidPython.

35. SolidWorks. http://www.solidworks.com/.

36. Abigale Stangl, Chia-Lo Hsu, and Tom Yeh. 2015. Transcribing Across the Senses: Community Efforts to Create 3D Printable Accessible Tactile Pictures for Young Children with Visual Impairments. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility (ASSETS '15)*. ACM, New York, NY, USA, 127–137. DOI: http://dx.doi.org/10.1145/2700648.2809854

37. Gilbert Tekli, Richard Chbeir, and Jacques Fayolle. 2013. A visual programming language for XML manipulation. *Journal of Visual Languages Computing* 24, 2 (2013), 110 – 135. DOI:http://dx.doi.org/https://doi.org/10.1016/j.jvlc.2012.11.001

38. TinkerCAD. https://www.tinkercad.com/.

39. Cesar Torres and Eric Paulos. 2015. MetaMorphe: Designing Expressive 3D Models for Digital Fabrication. In *Proceedings of the 2015 ACM SIGCHI Conference on Creativity and Cognition (C&#38;C '15)*. ACM, New York, NY, USA, 73–82. DOI: http://dx.doi.org/10.1145/2757226.2757235

40. Yunbo Zhang, Wei Gao, Luis Paredes, and Karthik Ramani. 2016. CardBoardiZer: Creatively Customize, Articulate and Fold 3D Mesh Models. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 897–907. DOI: http://dx.doi.org/10.1145/2858036.2858362