



Workbench for Creating Block-Based Environments

Mauricio Verano Merino

m.verano.merino@vu.nl

Vrije Universiteit Amsterdam

Amsterdam, The Netherlands

Koen van Wijk

koen.van.wijk@ict.nl

ICT

Eindhoven, The Netherlands

Abstract

Block-based environments are visual-programming environments that allow users to create programs by dragging and dropping blocks that resemble jigsaw puzzle pieces. These environments have proven to lower the entry barrier of programming for end-users. Besides using block-based environments for programming, they can also help edit popular semi-structured data languages such as JSON and YAML. However, creating new block-based environments is still challenging; developers can develop them in an ad-hoc way or using context-free grammars in a language workbench. Given the visual nature of block-based environments, both options are valid; however, developers have some limitations when describing them. In this paper, we present Blocklybench, which is a meta-block-based environment for describing block-based environments for both programming and semi-structured data languages. This tool allows developers to express the specific elements of block-based environments using the blocks notation. To evaluate Blocklybench, we present three case studies. Our results show that Blocklybench allows developers to describe block-based specific aspects of language constructs such as layout, color, block connections, and code generators.

CCS Concepts: • Software and its engineering → Visual languages; Domain specific languages; Graphical user interface languages; Syntax.

Keywords: block-based environments, data languages, visual languages, Projectional editors, IDEs, Blockly

ACM Reference Format:

Mauricio Verano Merino and Koen van Wijk. 2022. Workbench for Creating Block-Based Environments. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22)*, December 06–07, 2022, Auckland, New Zealand. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3567512.3567518>



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

SLE '22, December 06–07, 2022, Auckland, New Zealand

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9919-7/22/12.

<https://doi.org/10.1145/3567512.3567518>

1 Introduction

Computer programming allows people to communicate with computers through a programming language. Computer programming has widespread [27] and the number of people performing any type of programming activity has increased [28]. For instance, in the United States of America more than 12 million people say that they perform some programming at work, and almost 50 million use databases and spreadsheets [32].

Block-based environments are becoming more popular because they help to introduce programming concepts to end-users. This is achieved thanks to their benefits in terms of usability and lower the entry barrier to programming [1] by offering a what-you-see-is-what-you-get (WYSIWYG) experience. Block-based environment are visual programming environments that allow users to create programs by dragging and dropping blocks that resemble jigsaw puzzle pieces; each of these blocks represents a language construct. Moreover, block-based environments allow users to directly manipulate the program's Abstract Syntax Tree (AST) so that creating a block-based program is a form of projectional editing, and therefore by definition, programs are always syntactically correct [30, 37, 48, 50, 51]. In particular, this is useful for semi-structured data languages like JSON or YAML because editing them is error-prone due to the different curly and square braces and the deep nesting between objects.

Figure 1 shows an example of a block-based environment. These programming environments are often divided in three main parts: *palette* (left), *canvas* (center), and *stage* (right). The palette contains all the language constructs so that users can browse and discover the different constructs offered by the language. Then, the *canvas* is where users create programs by dragging and dropping blocks from the palette. Finally, the *stage* is used to render the output of executing programs.

Block-based environments are being adopted in different domains such as Computer Science, Arts, Education, and Robotics [14, 42, 48]. Therefore, the number of block-based environments that are being developed is increasing. When developers want to create new block-based environments they have essentially two options: extending an existing

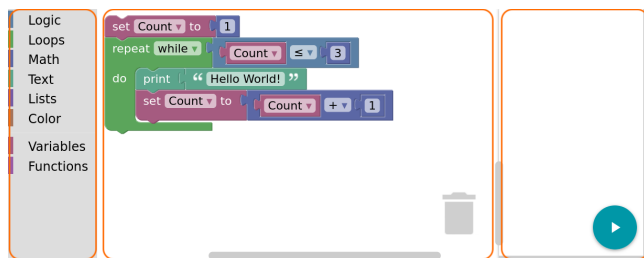


Figure 1. Block-based environment using Google Blockly.

block-based environment or developing it from scratch using a General Purpose Language (GPL). Recently, one research project uses specialized language engineering technology, Language Workbenches, to create block-based environments [47]; however, one of its limitations is that it does not allow developers to define block-based specific features (e.g., blocks' color and layout). Therefore, the tooling for developing these languages relies primarily on ad-hoc implementations through GPLs instead of specialized language engineering technologies [48].

In this paper, we first present a developer's trade-off to support the editing of human-editable semi-structure data languages (Section 2). This is followed by a brief overview of the state of the art for developing block-based environments (Section 3). Then, we present Blocklybench [3], which allows developers to describe and implement block-based environments and their code generators using blocks (Section 4). This enables developers to describe block-based specific elements, such as blocks color, layout, and generators. The generated block-based environments are compiled to Google Blockly [18]. The meta-environment capabilities of the Blocklybench are demonstrated by creating block-based environments for different languages, including a real-life application (Section 5). The implementation along with examples is available on GitHub¹.

We conclude this paper with a discussion of the advantages and limitations of the current approach (Section 6), related work (Section 7), and finally, we draw some conclusions and future directions (Section 8).

2 Developing Editors for Data Languages

When a developer needs an editor for a semi-structured data language (e.g., JSON or YAML) or a Domain-Specific Language (DSL), they have to deal with the trade-off between the usability for the end-user and their productivity (time available to invest in implementing the editor). Table 1 presents a summary of the different levels described above.

Level 1. The developer can expose the semi-structured data file directly. Configuration files for system administrators and developers often use this format. This is a very cheap

¹<https://github.com/block-based-langs/blocklybench>

option as the developer does not need to create anything. The user of this file can just use any text editor. However, it is hard for the user to learn as the syntax and scheme are unfamiliar and error-prone. Therefore, the level is typically not exposed to end-users and is used by developers.

Level 2. The developer can still expose the semi-structured data file; however, it also supplies a scheme. This will help the users of this file get suggestions and check for consistency. This will cost the developer in the order of one day. This level exposes the syntax, but it is still not exposed to end-users and is used by developers.

Level 3. The developer can also choose to make a block-based environment that represents the semi-structured data file. The user does not have to learn the syntax or scheme as the block-based environment is projectional. Besides this, the block-based environment can give tooltips and directly link to the documentation. This paper introduces a meta-environment that reduces the developer costs to a magnitude of days.

Level 4. The developer can also choose to make an editor in Eclipse Modeling Framework (e.g., Sirius, Eugenia, or GMF) or MPS. However, the costs of creating this editor are higher, in the order of a week.

Level 5. The developer makes a dedicated (web) application for the semi-structured data. The costs of creating this dedicated application are in the order of a month.

Although all levels are used, we argue that there is a big gap in terms of usability and development costs between level 2 and level 4. A block-based environment can fill this gap. The following section discusses the ways of developing block-based environments.

3 Developing Block-Based Environments

This section presents developers' most common ways and tools to create block-based environments. One of the most common ways to develop a block-based environment is by implementing everything from scratch through GPLs. This is a popular solution but expensive in terms of productivity since developers must write everything from scratch. Nowadays, software libraries are essential for software development, and today's systems heavily rely on libraries to increase developers' productivity and reduce development time [41]. In the context of block-based environment development the context is not different, as identified by Verano Merino et al. [48].

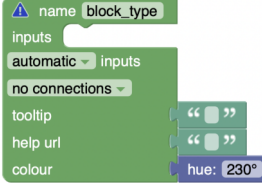
3.1 Libraries

Most block-based environments are implemented using third-party libraries or extending existing block-based environments. In this sense, Google Blockly [18] is by far the most

Table 1. Levels of editors for semi-structured data languages.

Level	Implementation	Example	Development costs	Cost to learn	User
1	Text	Notepad	0	\$\$	Developer
2	Text + scheme	Visual Studio Code	\$	\$\$	Developer
3	Block-based environment	Blockly	\$	\$	End-user
4	Eclipse Graphical Editor / MPS	Capella	\$\$\$	\$	End user
5	(Web) Application	Homey	\$\$\$\$	\$	End user

Table 2. Block definition using Google Blockly.

Textual notation	Blocks
<pre>Blockly.common.defineBlocks([{ "type": "block_type", "message0": "", "colour": 230, "tooltip": "", "helpUrl": "" }]);</pre>	

popular library for developing block-based environments. This library allows developers to create the entire UI of a block-based environment using JavaScript or the block factory interface [19]. In this way, developers can define their languages' toolboxes and blocks. Table 2 shows an example of how to define a block using Blockly (JavaScript). On the left side is the textual notation and on the right side is the equivalent using the block notation offered through the block factory. Additionally, developers must define the toolbox containing different categories; each category groups different blocks (language constructs). Afterward, developers need to define a code generator so that the behavior of the language is compiled into an executable programming language (e.g., JavaScript or Python) or a semi-structured data language (e.g., JSON or YAML).

3.2 Extending Existing Environments

The other popular solution for developing block-based environments is extending existing environments. In this category, most developers rely on extending the most popular and robust platforms such as Scratch [39], Snap! [24], CT-Blocks [49], and App inventor [52]. This approach has some limitations because it relies on the existing infrastructure of the host platform. Therefore, their pertinence depends on the requirements of the new language. It could be effective if the desired language could be seen as an extension of the host language. However, if the desired language requires domain-specific requirements, this option might be more challenging than other options.

3.3 Using Language Workbenches

Specialized language technology has been available for several years, so-called language workbenches. However, as

identified by [48], their usage in the context of developing block-based environments is still limited. Only one project, Kogi [29, 47], uses language workbenches to create block-based environments. This approach relies on the notion of describing computer languages through context-free grammars. Kogi takes as input grammars containing the language definition and processes it to create a block-based counterpart. This approach enables developers to use language workbenches for creating block-based environments. However, this approach requires developers to have expertise in writing grammars, which is not always the case. Another limitation of this approach is that, as mentioned by the authors, the resulting environments have some usability issues. The main reason for this is that block-based environments have some particular characteristics (e.g., layout, colors, and palette) To overcome this issue, [46] applied some heuristics over the input grammars to improve the usability of generated languages based on aesthetic criteria defined by the authors. However, this still imposes some limitations because developers do not have complete control of the look and feel of their generated environments.

In the next section, we present an approach that allows developers to create their block-based environments, including block-specific features (e.g., layout, colors, and connections), using block-based notation.

4 Blocklybench

Blocklybench is a tool for describing and implementing block-based environments using block-based notation. The generated environments use Google Blockly. This section describes how to implement a block-based environment using Blocklybench and all its features.

There are different steps involved when developing a new block-based environment. First, developers have to define the language constructs (blocks). Then, they have the option to create categories to group blocks and ease the discovery of language constructs for end-users. Finally, blocks need to be translated into an executable language or data language; this requires the definition of one or multiple generators depending on the use cases of the language.

Since offering feedback is essential, as stated by Norman [33] “*feedback must be immediate: even a delay of a tenth of a second can be disconcerting*”, Blocklybench offers

immediate feedback during the whole block-based environment development process. In this manner, developers can tweak and see the current status of their implementations in real life.

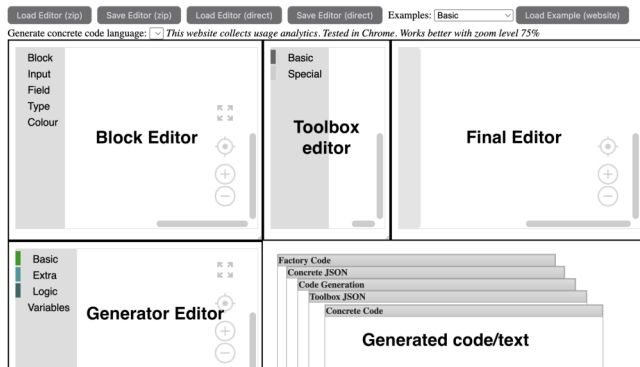


Figure 2. Editors.

4.1 Features

This section presents an initial view of Blocklybench as shown in Figure 2; it is divided into four block-based editors and one panel: *Block Editor* (top-left), *Toolbox Editor* (top-middle), *Final Editor* (top-right), *Generator Editor* (bottom-left), and *Generated Code* panel (bottom-right).

The following paragraphs present a detailed description of each of the four block-based editors and the Generated Code panel that are part of Blocklybench.

Block Editor. The Block Editor is used to create new blocks (language constructs) and it is based on the block factory offered by Blockly [19]. Blocks in the block factory are defined, as shown in Table 2, by a name, set of inputs, connections, tooltip, help url, and colour. The main difference between this and the Block Editor is that the latter allows developers to create multiple blocks in the same workspace, while the first one allows them to create only one block at a time.

A block is defined by different properties such as a name, block connections, possible inputs, tooltip, color, and help URL. The inputs can contain one or several fields. A block can be defined with one of five different connections, namely: top and bottom connections, no connections, left output, top connection, or bottom connection.

Compared to the existing Block factory, Blocklybench's block editor includes four additional field blocks: *self referencing*, *dropdown from URL*, *dropdown from URL split*, and *dropdown with workspaces*.

Self-Referencing Field. This field block is a self-referencing field that creates a drop-down field filled with the names of all the other blocks within the editor. This field has three properties: *name*, *field*, and *from workspace*. The first property for

every field is the name of this field within the block, and one vital feature is that the name has to be unique. The second property and third property refer to all fields of all blocks from the selected workspace that has the name 'FIELD'. The Google Workflows language, as described in section 5, uses this field to create the 'next' block. The 'next' block has a drop-down with all 'NAME's of the steps.

dropdown NAME field: FIELD from workspace: Factory

Dropdown from URL field. This field is also a drop-down menu filled from a URL, which can be used to fill the menu options with a longer or dynamic list of elements.

dropdown url NAME url

This field has two properties: *NAME*, and *url*. The first property is the *NAME* of the field, which must be unique in the block, and the second property is the URL of a file that contains the options to be displayed in the dropdown. An example of such a file is shown below:

```
[
  [
    "Option_1_user_friendly",
    "OPTION1"
  ],
  [
    "Option_2_user_friendly",
    "OPTION2"
  ],
]
```

This file will fill in the dropdown with two options: "Option 1 user friendly" and "Option 2 user friendly".

Dropdown from URL Split. The field, 'dropdown url split' is introduced to handle the list when the number of items in the drop-down list is too long. It helps by grouping the items and making the drop down more manageable.

dropdown url NAME1 NAME2 url split :

This field has even four properties: *NAME1*, *NAME2*, *url*, and *split*. The *url* property works the same as in the previous field. Now the JSON file needs to be a bit different. The user-friendly descriptions are split by the *split* field, in this case ':' . *NAME1* and *NAME2*, are the names of two drop-down lists.

```
[
  [
    "Option:user_friendly_option_1",
    "OPTION:OPT1"
  ],
  [
    "Option:user_friendly_option_2",
    "OPTION:OPT2"
  ],
  [
    "Property:user_friendly_property_1",
    "PROP:PROP1"
  ],
  [
    "Property:user_friendly_property_2",
    "PROP:PROP2"
  ],
]
```

]

The first drop-down will be filled with 'Option' and 'Property'. The second drop-down will change depending on the first. When 'Option' is selected, the second drop-down will give 'user-friendly option 1' and 'user-friendly option 2'. This is how the drop-down is grouped and managed through the URL split.

Drop-down with Workspaces Field. This drop-down field contains the name of the workspaces or editors; it has one property (NAME). The field will give a drop-down with the names of the workspaces; factory (Block editor), toolbox (Toolbox Editor), code (Generator Editor), and concrete (Final Editor). This block is used inside the first field (self-referencing).



As mentioned before, feedback is essential; therefore, the resulting block, including its layout and color is live updated in the toolbox of the Final Editor (top-right Figure 2). In particular, Blocklybench supports the interactive development of block-based environments and their artifacts (block definition, toolbox, and generators) by keeping them in sync. For instance, a common task is renaming a block's name; when this action occurs, Blocklybench will rename all the block occurrences in all places, including the toolbox and generators editor. Also, the resulting JavaScript code that creates the block can be inspected in the Factory Code window in the Generated code panel (bottom-right Figure 2).

Toolbox Editor. This editor supports two types of toolbox layouts *categorized* and *uncategorized*. The first is the traditional Blockly toolbox, where blocks are grouped into categories, while the second does not have any categories; the blocks are directly displayed in the toolbox. In both cases, the blocks shown in the toolbox are selected from a drop-down list containing all the blocks defined in the Blocks editor. In particular, developers can define a name and color for each category for the *categorized* toolbox.

When a toolbox contains nested blocks, it is possible to define some default values for the fields by using the blocks from the *Special* category in the Toolbox Editor. Moreover, it is possible to create getters and setters via the *Variables* category and to define additional labels and gaps in the toolbox.



Finally, Blocklybench allow developers to see the JSON code of the toolbox in the *Toolbox JSON* window in the Generated code panel (bottom-right Figure 2).

Final Editor. This editor (top-right Figure 2) is used to show a preview of the editor that has been described in the Block and Toolbox Editor. Its primary value is to allow developers to see and verify the layout and implementation of the different blocks and categories in the block-based environment. More importantly, this editor is live updated to reflect the most up-to-date information. Adding blocks to the Final Editor canvas will also trigger the code generation as defined by the Generator Editor that is shown in the Concrete Code windows of the Generated code panel.

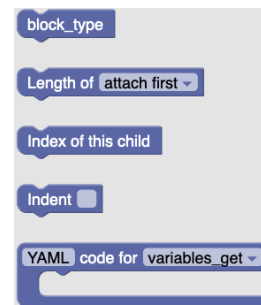
Generator Editor. This editor is used to define a code generator for each block. One generator block is needed per block type and language. The generator per block is built up using string templates combined with placeholders. The placeholders are Field, Statements, Values, and Comments, and each of them contains a drop-down menu with their available values from the block definition.



Statements are indented by default, but developers can change this behavior by unchecking the indent property on the statement block.



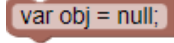
There are five extra placeholders available for code generation.



The Block Type is used for obtaining the current block's type.

When developers are working with statements explicitly with data languages, they are often required to know the number of statements (length) and the index of the current statement; therefore, Blocklybench offers the *length of* and *index of this child* blocks.

In addition, Blocklybench offers a `generate_javascript` block.



This block has a text field in which developers can write JavaScript code, which is directly added to the generated code. Probably good to check the generated code directly as it is straightforward to generate incorrect code. All the generated code can be inspected in the Code Generation window within the Generated code panel (bottom-right Figure 2). This option allows developers to use any Blockly JavaScript code directly in their code generators.

Generated code. This panel (bottom-right Figure 2) contains five windows with the generated code from the different artifacts developed with Blocklybench. This panel's intended use is to give early feedback on the resulting code generation based on the usage of the Final Editor and the different generators. It is essential to mention that Blocklybench's allows developers to define different code generators in different languages. Therefore, the language generator displayed in the Generate code panel can be switched by clicking the drop-down menu in the navigation header; this menu shows the names of all the code generators that have been defined in the Generator Editor.

Factory Code: This window displays the JavaScript code to create the new blocks. This code is executed on every change to create the blocks in the Final Editor.

Concrete JSON: This window displays the JSON representation of the Final Editor.

Code Generation: This window displays the JavaScript of the code generators. This code is also executed on every change to create code in the Concrete Code window.

Toolbox JSON: This window displays the JavaScript to generate the toolbox. This code is also executed on every change to create the Toolbox of the Final Editor.

Concrete Code: This window displays the resulting code created by the selected code generator.

4.2 First Example

This running example is used to illustrate the Blocklybench's features described before, and to explain how they can be used in practice [4].

Figure 3 shows an example of a single block block-based environment implemented in Blocklybench, as it provides an overview of the different editors. In this case the Block Editor contains a single block called `nice_new_block`, and its definition includes a title (TITLE) and a name (NAME). The Toolbox Editor contains the most basic toolbox without categories with a single block listed, and both the toolbox and the block are displayed in the Final Editor. The Generator Editor contains the definition of a code generator for a language called LANG. This generator prints out the content of the NAME field inside the block followed by a colon (:) and

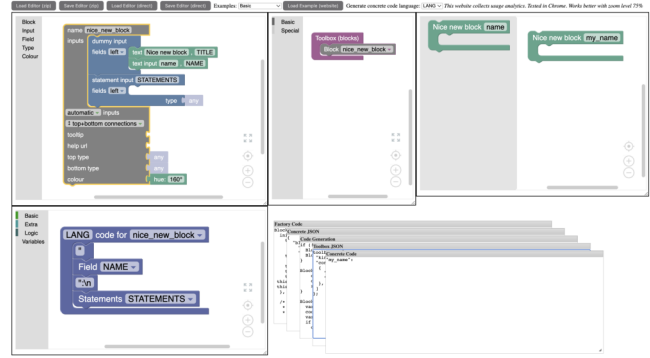


Figure 3. First block.

a newline followed by the statements. The generated code can be inspected in the *Concrete Code* window within the Generated code panel.

Below we present additional features offered in all editors generated with Blocklybench.

4.3 Search

All editors have a search toolbox to quickly highlight its definition across all editors. To achieve this, users should left-click a block in the Final Editor, and Blocklybench will highlight the block type in all the editors and consequently will jump all editors to that block type definition.

4.4 Context Menu

When a user right-clicks a block a context menu with three additional actions is displayed². First, the *Clean Up* action is useful when employing several blocks in an editor as quite a lot of blocks are created the layout will clutter the developer's workspace. Therefore, this action organizes the blocks in the workspace in such a way that they do not overlap each other and the workspace is usable again. Secondly, there is the *Collapse blocks* action, this works when developers want to hide some blocks that are finished in order to have more space to work on unfinished blocks and to free up some space in the editors. Thirdly, a common task is to share the current status of the editor. Here Blocklybench allows developers to take screenshots of their editors and store them as a PNG picture.

4.5 Save and Load

The navigation header (topmost part of Blocklybench) offers several buttons to save and load editors. The save and load buttons on the top save the state of all the editors: Block, Toolbox, Generator, and Final Editor. It also creates an editor.html file containing the Final Editor. There are

²We refer to it as additional actions because Blockly, by default, only activates some actions at the workspace level (e.g., undo and delete blocks)

two options to save a block-based environment: either as a zip file or as a folder with all the files directly to disk ³.

All editors are stored using a flat JSON format [23].

4.6 Load Examples

Saving is essential to make the editor persistent; therefore, the counterpart loading is also necessary. Blocklybench offers a mechanism for loading previously saved definitions. There are four built-in languages (*Basic*, *Google Workflow*, *Smooth Voxel*, and *Fectar*) available to load from the website. The built-in languages include *Basic* that has blocks for languages based on dictionaries, lists, and values, similar to the ones we could find in JSON or YAML formats. The other built-in languages are described in section 5. However, users can also load their own languages by clicking the Load Editor button.

4.7 Advanced: Copy Block

Different blocks (language constructs) might have a similar structure. This is why it becomes handy to offer a functionality that allows developers to reuse and tweak existing block definitions for creating new blocks. Blocklybench allows developers to achieve this by allowing them to make a copy of a block. To copy a block, developers have to right-click a block in the Final Editor and select *Create ... Copy* (Figure 4). This action creates a new block with its own code generators (the same as the block it was copied from), and the block is added to the toolbox below the original block.

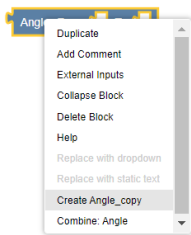


Figure 4. Create block copy.

4.8 Advanced: Tool-Tips on Fields

Tool-tips are essential in a block-based environment to give users in-context feedback. They can be placed both at the block and the field level. To include a tool-tip at the block level, developers can use the tool-tip field in the block definition. While creating a tool-tip at the field level can be achieved by adding a comment on the field in the Block Editor as shown in Figure 5.

4.9 Advanced: Converting Field Types

When converting generic languages to more concrete ones and specific constructs, we often converted text fields into

³This feature uses The File System Access API [31]

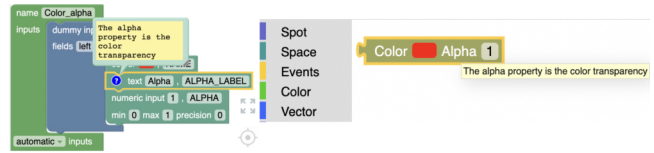


Figure 5. Defining tool-tips at the field level.

static or drop-down fields. Therefore, Blocklybench allows this by right-clicking a block and selecting *replace with drop-down* and *replace with static text*.

5 Case Studies

This section presents three case studies that we implemented using Blocklybench, namely *Fectar*, *Google Workflows*, and *Smooth Voxel*. All languages and the generated editors are publicly available online (see sections below). Beneath, we briefly describe each of the languages and present the resulting block-based editor.

5.1 Fectar Blocks

Fectar [17] is an Augmented Reality (AR), Virtual Reality (VR), and Mixed Reality (MR) content management system that includes a tool to build the Metaverse. In *Fectar Studio* fig. 6 end-users can create 3D spaces with several objects or spots. Basic usage like "click on" to appear or disappear,

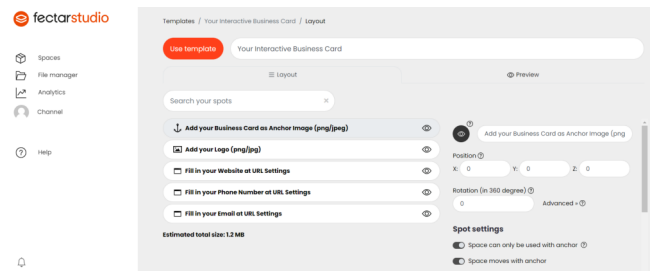


Figure 6. Fectar Studio.

rotation, and location of objects or spots are provided, but sometimes extra behavior is needed in the 3D space. This extra behavior needs to be programmed by developers using JavaScript using *Fectar Code* spot. However, in this scenario, users need to know both JavaScript and the *Fectar* API. Our block-based environment creates a new block-based language that enables these extra behaviors by dragging blocks instead of writing textual JavaScript code. Since the syntax is well-known to be an entry barrier for end-users [35], this block-based environment lowers this barrier level for end-users and is also an easy way to learn the *Fectar* API by exploring the language constructs via the toolbar.

The definition of Fectar blocks using Blocklybench is available online [5], and the resulting block-based editor is available at [6].

Listing 1 shows a JavaScript code snippet that should be typed; what it does is that when a spot is clicked, the position is set to (0, 0.5, 1). Then, the information about the spot is logged. In a traditional (textual) environment the developer needs to understand the syntax of the `onClick` function and that their parameter (`eventParam`) contains a field with the spot. In the block-based world, this process is not required since this information is available directly in the editor. Figure 7 shows the same function using a block-based notation. With a block-based environment it is possible to drag and drop the block to create the corresponding JavaScript.

```
function onClick(eventParams) {
  var spot = eventParams.spot;
  spot.position = new Vector3(0, 0.5, 1);
  log(spot);
};
```

Listing 1. JavaScript code using Fectar API.

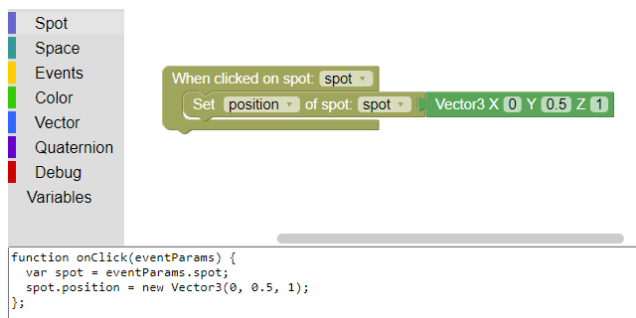


Figure 7. Fectar Blocks Editor.

5.2 Google Workflows

Google Workflows is a service that combines Google Cloud services and APIs that soothe different development tasks such as building reliable applications, processing automation, and machine learning pipelines [22].

Google Workflow relies upon YAML files like the one shown in Listing 2. This example first gets the current time and, depending on the day of the week (Friday, Saturday or Sunday, or weekdays), returns the corresponding string (Almost weekend, Weekend, or Workday) [20].

This example shows several layers of syntax. One is the YAML syntax, which seems easy, but knowing exactly when to use indentation and/or '-' might be challenging. The next layer is the scheme keys (e.g., `call`, `args`, `result`, `next`), and some values in the dictionaries must match other steps, for example `conditionalSwitch`.

The block-based editor Figure 8 helps reduce the complexity of these layers since most syntax and semantics can be

encoded into blocks that will always be syntactically correct. Those blocks only contain the available keys. Also, a drop-down field is provided with the names of the already existing blocks; consequently, the 'conditionalSwitch' value can be selected.

The definition of Google Workflows blocks using Blocklybench is available online [7], and the resulting block-based editor is available at [8].

```
main:
  steps:
    - get_current_time:
      call: http.get
      args:
        url: https://bit.ly/3K353vc
      result: currentTime
      next: conditionalSwitch
    - conditionalSwitch:
      switch:
        - condition: ${currentTime.body.dayOfTheWeek == "Friday"}
          next: Friday
        - condition: ${currentTime.body.dayOfTheWeek == "Saturday"
          OR currentTime.body.dayOfTheWeek == "Sunday"}
          next: Weekend
      next: Workday
    - Friday:
      return: "Almost weekend"
    - Weekend:
      return: "Weekend!"
    - Workday:
      return: "Workday..."
```

Listing 2. Google Cloud Workflows example.

Additional help can be offered for the current blocks. For instance, the `condition` block would benefit from helper blocks because currently this is an input text and users can still make mistakes. This shows that block-based environments can be built up iteratively; First, the basic blocks, with free text (with still requires knowing some syntax), and then developing more detailed blocks. For instance, the `condition` value can be filled with new blocks and the `args` values can have some defaults instead of free text.

5.3 Smooth Voxels

Smooth Voxels allows developers that lack blender skills to create 3D models. It allows developers to transform voxel models into low poly-style looking models. Concrete, Smooth Voxels works by averaging the vertices to obtain a smoother representation.

For instance, the code required for creating a 3D model of an apple is shown in Listing 3. This code snippet is relatively straightforward. First, it defined the properties of the 3D model (e.g., size, scale, and origin). Then, the materials and the colors are defined. In particular, all colors have a corresponding letter between 'A' to 'F'; these letters are used to create the 3D model layer by layer. The dash symbol ('-') is used to represent that no voxel is needed, and note that material 'F' is black and is only used in the top part, while material 'E' represents the brown color in the stem of the



Figure 8. Google Workflows block-based editor.

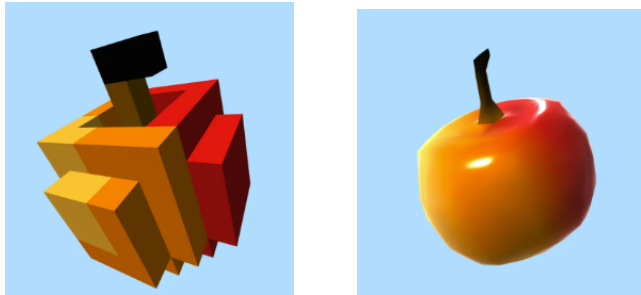


Figure 9. Apple Voxel and Smooth.

apple, the remaining materials ('A', 'B', 'C', and 'D') give the nice yellow-red appearance to the apple.

Similar to the Google Workflow language Section 5.2, users need to be familiar with the scheme shown in Listing 3; they need to know the different properties and valid values for each of them. Also, to define the colors of the different materials, users need to know the hexadecimal code of the colors used. Therefore, the schema is simple, but it is error-prone. To mitigate this, we believe that using a block-based environment will help users to define their 3D models because they can choose the properties directly from the toolbox. Also, they can set their values from only valid ranges or using drop-down menus, and the block-based editor offers help to the users via the tooltips and the links to the documentation.

Figure 10 shows the definition of an apple using the same properties but now with block notation.

The definition of Smooth Voxels blocks using Blockly is available online [9], and the resulting block-based editor is available at [10].

Our block-based editor offers two options for defining voxels. The first option uses plain text (using the same ASCII-art

syntax as the textual definition), and the second option uses drop-down menus that solely display available and valid color letters. The first and second options are less ideal due to the font used, and that for every letter a dropdown needs to be selected, respectively. For instance, Figure 11 shows an example of the definition of the voxels for the apple. The number before a voxel line represents the number of repetitions of that segment.

```

size = 7
scale = 0.15 0.17 0.15
origin = -y
ao = 5 3
material lighting=smooth, roughness=0.25, fade=true,
  deform=3 colors=A:#C11 B:#F60 D:#000 C:#F93
material lighting=smooth, roughness=1, fade=true,
  deform=3 colors=E:#840 F:#000
voxels
----- --AAA-- --AAA-- --AAA-- -----
--AAA-- -BAAAA- -BAAAA- -BAAAA- -BAAAA- -----
-BADAA- BAAAAA BAAAAA BAAAAA -B---A- -----
-BDDDA- BAAAAA CAAAAA CABAAA -B-E-A- ---E--- --FF--
-BBDBA- BAAAAA CAAAAA CABBAA -C---A- -----
--BBB-- -BAAAB- -CAAAB- -CAAAB- -CCBBB- -----
----- --BBB-- --CCB-- --CCB-- -----
    
```

Listing 3. Textual definition of an apple [36].

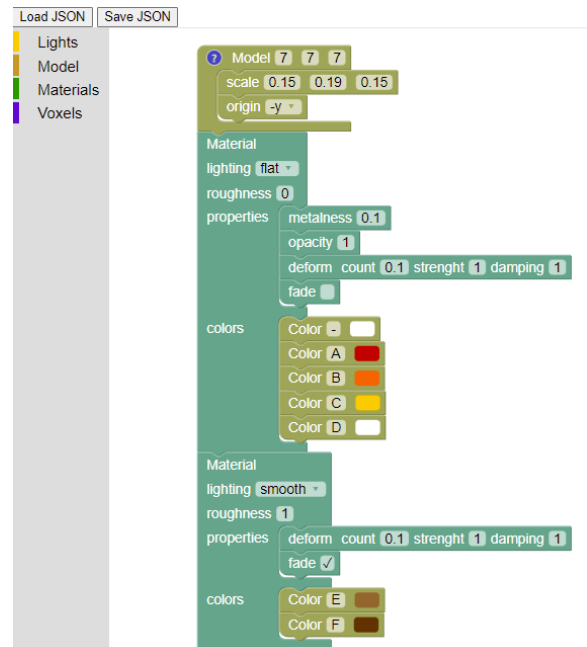


Figure 10. Smooth Voxels with Blockly.

5.4 Effort

To have a better understanding of the size of the different case studies (Section 5), we measured the number of blocks required to define the language’s toolbox and language constructs (blocks), the resulting number of categories in the toolbox and the number blocks, and the number of SLOCs of

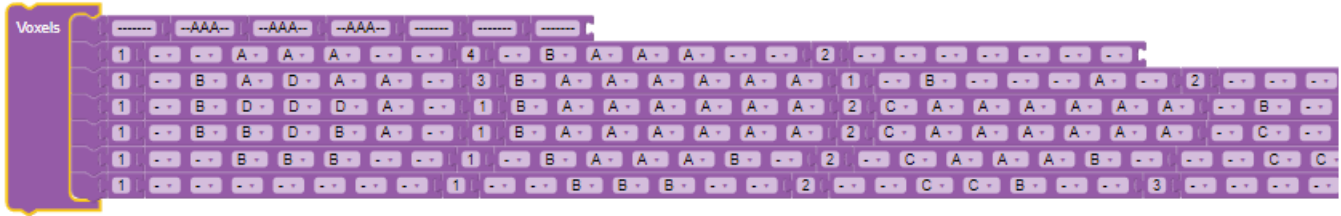


Figure 11. Voxel definitions using block-based notation.

the resulting block-based editor. We manually measured the number of blocks required for implementing the different cases studies, and for measuring the SLOCs of the generated block-based editors we used SonarQube [40].

Table 3 presents a summary of quantitative data we measured for each case study. The first column (Language) contains the name of the language. Then, the following three columns contain information related to the language definition. This includes the number of Blocklybench’s blocks required to define the toolbox (Toolbox), the language constructs (Blocks), and their code generators (Code Gen.) Finally, the remaining columns display the information related to the generated editor (Editor). This part is divided into two; the first two columns, Categories and Blocks, contain the number of categories and blocks of the language, respectively. Finally, the last three columns show the number of generated Lines of Code (SLOC) of each block-based editor divided into three parts, the HTML, the JS, and the sum of the two.

For all the case studies, the number of HTML SLOCs is identical because Blocklybench generates the same sample web application; what differs from each block-based environment are the JavaScript files. The JavaScript (JS) code represents the block’s definition using Blockly’s API. The *Total* column in Table 3 represents the sum of the SLOCs of the HTML and the JS columns. On average each generated block-based environment contains around 3.9k SLOCs. As we observe, the size of the generated editors is consistent. Most of the code is related to the definition of the blocks, then the code generator and finally the toolbox.

6 Discussion

As explained throughout the paper, Blocklybench offers an interesting block-based mechanism for creating block-based environments and their code generators. In the following paragraphs, we present some of the limitations of the current approach and possible ways in which further research projects can address them. Blocklybench offers a great way to develop block-based environments and gives options to define all block-specific characteristics, including color and layout. However, language developers must keep in sync the link

between the blocks and code generation of the text languages, which is something that other approaches like Kogi [47] and S/Kogi [46] offer for free. One way to mitigate this could be to integrate Blocklybench’s approach with Kogi or S/Kogi, so that the input grammars used in these approaches could contain annotations to describe block-specific aspects and that the grammar can be edited using Blocklybench’s notation.

Blocklybench allows developers to create different types of languages, including programming and data languages. However, in the current evaluation (Section 5) we have implemented only one programming language Section 5.1; while we have implemented several data languages Sections 5.2 and 5.3. Based on our results and experience, and the results from other research projects [46, 47], the block-based metaphor works best for Domain-Specific Languages (DSLs) and data languages. Additional research is required to determine whether this metaphor is also beneficial for programming languages beyond the realm of programming education.

Also, in this direction, when data languages are created with Blocklybench there is a need for parsing existing data files (e.g., JSON or YAML files) so that they can be edited using the newly created block-based editor. Currently, only the *Basic language* [2] supports parsing lists and dictionaries, which are common constructs in data languages. As mentioned, developing block-based environments iteratively is one of Blocklybench’s key strengths. Therefore, maintaining an abstract parser for data languages (JSON and YAML) derived from the *Basic language* with newly created blocks is an open question for further research. One possible solution could be that the editor keeps track of the language changes and assists developers in this transition.

Even though Blocklybench seems relatively easy to use, only a few users have tested it, and only around 15 languages have been developed. Based on their usage, they have reported some usability issues. For instance, some users indicated that the number of windows on the screen is overwhelming and sometimes intimidating. To mitigate this, we plan to develop a Visual Studio Code extension to control and manage the different windows better. Moreover, some fields within a block come with validators (type checkers) for

Table 3. Number of blocks required for implementing the case studies using Blocklybench, and the number of Lines of Code (SLOC) of the generated block-based editors.

Language	Lang. Def.			Editor				
	Toolbox	Blocks	Code Gen.	Categories	Blocks	HTML**	JS**	Total**
Fectar	34	232	131	8*	26*	19	3721	3740
Google Workflow	17	332	116	2	14	19	4163	4182
Smooth Voxel	27	228	145	4	22	19	3889	3908

*An additional block is available for defining variables.

**This measurements are in SLOCs.

common data types (e.g., number range and precision) that improve the end-users overall experience. However, Blocklybench does not support the definition of new validators or mutators. Supporting this might be beneficial for the creation and usage of new blocks.

Currently, as shown in Section 5.4, Blocklybench generated block-based environments are one (simple) static HTML page and a JavaScript file that contains the whole definition of the block-based editor, including code generators and toolbox. This can be turned into a more robust web application, including support for NPM, UNPKG, React, or as a Visual Studio Code extension. Likewise, Blocklybench’s generated languages do not support collaboration features. This means that several users cannot work on the same application at the same time. This is an interesting and valuable feature to collaboratively create block-based applications in real time between various users.

Finally, Blocklybench supports blocks localization using string tables, in the same way that Blockly [21]. However, when a new user’s language is added, Blocklybench does not create nor identify an editor that keeps track of the newly created messages. Existing Blockly localization mechanisms offer support mainly for programming languages but lack support for strings, which are essential for block-based data languages.

7 Related Work

Block-based environments are part of the graphical editors or visual languages family. Currently, there is a lack of development tools for creating block-based environments [13]. One of the strengths of visual languages is that they make programming easier for beginners than textual languages [26]. Most developers have to make their block-based implementations using general-purpose programming languages with little or no support for language definitions. Blocklybench contributed to the research line on programming environment generation [12, 15, 16, 25, 34, 38, 45]. In this research line, there are a couple of approaches to using language workbenches for implementing block-based environments. For instance, Kogi [47] takes new or existing textual languages and generates a block-based interface from this definition.

Likewise, S/Kogi [46] follows a similar approach, but it includes a set of heuristics that improve the usability of generated block-based environments. Within the block-based world, Blockly is the most common tool and it only offers the block factory [19] interface, which allows users to create custom blocks and their toolbox. However, this tool does not allow developers to directly create code generators for these languages using the blocks notation nor to keep the blocks, toolbox and code generation in sync during development.

Model-Driven Software Engineering (MDSE) focuses on creating domain models for software development. Eclipse Modeling Framework (EMF) [11] is one of the most popular frameworks for MDSE, and it relies on a meta-model editor, which is a visual editor for defining domain models using a graphical notation. This is a similar approach to the one proposed in Section 4, but we focused on offering a block-based notation with block-based-specific features.

8 Conclusions & Future Work

Block-based environments are great tools that help end-users achieve programming tasks by lowering the entry barrier and guiding them in creating their programs. However, their usage has been mostly for teaching programming to children, and their implementation is mostly in an ad-hoc fashion. In this paper, we present Blocklybench, which is a tool that helps developers to create their block-based environments using blocks. Therefore, it is possible to create new block-based environments with a block-based environment. However, this does not guarantee a correct and successful block-based language because careful language design is required [44].

Nevertheless, block-based environments development still requires effort, particularly experience and knowledge of language design. However, Blocklybench’s approach offers a high-level language description of these languages that also use the block-based metaphor. Also, its immediate feedback gives developers ways of verifying and quickly testing with final users whether the environment fits the business needs.

There are several points to be addressed in further research. For instance, adding support for fine-grained features

when defining blocks (e.g., set moveable, deletable and editable properties at the block level [43]). In the same direction, supporting field verification for non-basic data types would benefit end-users. Also, it is helpful for existing block-based environment that their definitions or definitions from existing JSON/YAML schemes can be imported into Blocklybench for further development.

Specialized language engineering technology like language workbenches is powerful for creating and implementing software languages. However, they lack support for block-specific matters like the ones supported by Blocklybench. Therefore, bridging language engineering tools like Kogi with Blocklybench, would allow developers to take the benefit of both worlds. On the one hand, the full implementation of a language and all its components, and on the other hand, tweaking and defining block-specific properties.

Finally, one of the main motivations and benefits of block-based environments are the benefits for end-users; therefore, we plan to conduct a user study to evaluate the usability of Blocklybench and their generated editors.

References

- [1] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (2017), 72–80. <https://doi.org/10.1145/3015455>
- [2] Blocklybench. 2022. Basic Language. <https://motar-242711.ew.r.appspot.com/?editor=basic&load=1>. <https://motar-242711.ew.r.appspot.com/?editor=basic&load=1> [Online, accessed 28 July 2022].
- [3] Blocklybench. 2022. Blocklybench. <https://motar-242711.ew.r.appspot.com/>. <https://motar-242711.ew.r.appspot.com/> [Online, accessed 28 July 2022].
- [4] Blocklybench. 2022. Blocklybench. <https://motar-242711.ew.r.appspot.com/?editor=first&load=1>. <https://motar-242711.ew.r.appspot.com/?editor=first&load=1> [Online, accessed 28 July 2022].
- [5] Blocklybench. 2022. Fectar Code Blocks. <https://motar-242711.ew.r.appspot.com/?editor=fectar&load=1>. <https://motar-242711.ew.r.appspot.com/?editor=fectar&load=1> [Online, accessed 28 July 2022].
- [6] Blocklybench. 2022. Fectar Code Blocks. <https://motar-242711.ew.r.appspot.com/editors/fectar/editor.html>. <https://motar-242711.ew.r.appspot.com/editors/fectar/editor.html> [Online, accessed 28 July 2022].
- [7] Blocklybench. 2022. Fectar Code Blocks. https://motar-242711.ew.r.appspot.com/?editor=google_workflow&load=1. https://motar-242711.ew.r.appspot.com/?editor=google_workflow&load=1 [Online, accessed 28 July 2022].
- [8] Blocklybench. 2022. Google Workflows Editor. https://motar-242711.ew.r.appspot.com/editors/google_workflow/editor.html. https://motar-242711.ew.r.appspot.com/editors/google_workflow/editor.html [Online, accessed 28 July 2022].
- [9] Blocklybench. 2022. Smooth Voxels Blocks. <https://motar-242711.ew.r.appspot.com/?editor=svox&load=1>. <https://motar-242711.ew.r.appspot.com/?editor=svox&load=1> [Online, accessed 28 July 2022].
- [10] Blocklybench. 2022. Smooth Voxels Blocks Editor. <https://motar-242711.ew.r.appspot.com/editors/svox/editor.html>. <https://motar-242711.ew.r.appspot.com/editors/svox/editor.html> [Online, accessed 28 July 2022].
- [11] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. 2003. *Eclipse Modeling Framework*. Pearson Education.
- [12] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton, Evelyn Duesterwald, and Jurgen Vinju. 2009. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. 44, 10 (2009), 191–206. <https://doi.org/10.1145/1639949.1640104>
- [13] Enrique Coronado, Fulvio Mastrogiovanni, Bipin Indurkha, and Gentiane Venture. 2020. Visual Programming Environments for End-User Development of intelligent and social robots, a systematic review. *Journal of Computer Languages* 58 (2020), 100970. <https://doi.org/10.1016/j.cola.2020.100970>
- [14] Shruti Dhariwal. 2019. BlockArt: Visualizing the ‘Hundred Languages’ of Code in Children’s Creations. In *Proceedings of the 2019 on Creativity and Cognition* (San Diego, CA, USA) (C&C ’19). ACM, 633–639. <https://doi.org/10.1145/3325480.3326585>
- [15] Söderberg Emma and Hedin Görel. 2011. Building Semantic Editors Using JastAdd: Tool Demonstration. (2011), 6 pages. <https://doi.org/10.1145/1988783.1988794>
- [16] S. Erdweg, T. v. d. Storm, M. Volter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. v. d. Vlist, G. Wachsmuth, and J. v. d. Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. <https://doi.org/10.1016/j.cl.2015.08.007>
- [17] Fectar. 2022. Fectar. <https://www.fectar.com>. <https://www.fectar.com> [Online, accessed 25 July 2022].
- [18] Google. 2020. Blockly. <https://developers.google.com/blockly>. <https://developers.google.com/blockly> [Online, accessed 25 July 2022].
- [19] Google. 2020. Blockly block factory. <https://blockly-demo.appspot.com/static/demos/blockfactory/index.html>. <https://blockly-demo.appspot.com/static/demos/blockfactory/index.html> [Online, accessed 28 July 2022].
- [20] Google. 2020. Google Cloud Platform. https://github.com/GoogleCloudPlatform/workflows-samples/blob/main/src/step_conditional_weekend.workflows.yaml. https://github.com/GoogleCloudPlatform/workflows-samples/blob/main/src/step_conditional_weekend.workflows.yaml [Online, accessed 28 July 2022].
- [21] Google. 2020. Localize Blocks. <https://developers.google.com/blockly/guides/create-custom-blocks/localize-blocks>. <https://developers.google.com/blockly/guides/create-custom-blocks/localize-blocks> [Online, accessed 17 August 2022].
- [22] Google. 2022. Google Workflows. <https://cloud.google.com/workflows>. <https://cloud.google.com/workflows> [Online, accessed 28 July 2022].
- [23] Blockly Group. 2022. Blockly json serialization and merging. https://groups.google.com/g/blockly/c/6lfkH-mSWdl/m/K25gHd_oAAAJ. https://groups.google.com/g/blockly/c/6lfkH-mSWdl/m/K25gHd_oAAAJ [Online, accessed 28 July 2022].
- [24] Brian Harvey and Jens Monig. 2020. Snap! 4.1 Reference Manual. <https://github.com/cwi-swat/rascal-minijava>. <https://snap.berkeley.edu/snap/help/SnapManual.pdf> [Online, accessed 12 July 2021].
- [25] Heering Jan and Klint Paul. 2000. Semantics of Programming Languages: A Tool-oriented Approach. *SIGPLAN Not.* 35, 3 (2000), 39–48. <https://doi.org/10.1145/351159.351173>
- [26] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37, 2 (2005), 83–137. <https://doi.org/10.1145/1089733.1089734>
- [27] Andrew J. Ko, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, Susan Wiedenbeck, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, and Henry Lieberman. 2011. The state of the art in end-user software engineering. *Comput. Surveys* 43, 3 (2011), 1–44. <https://doi.org/10.1145/1922649.1922658>
- [28] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. *End-User Development: An Emerging Paradigm*. Springer Netherlands, Dordrecht, 1–8. https://doi.org/10.1007/1-4020-5386-X_1

- [29] Mauricio Verano Merino and Tijs van der Storm. 2020. *cwi-swat/kogi: Kogi 0.1.0*. <https://doi.org/10.5281/zenodo.4033220>
- [30] Luke Moors and Robert Sheehan. 2017. Aiding the Transition from Novice to Traditional Programming Environments. In *Proceedings of the 2017 Conference on Interaction Design and Children (Stanford, California, USA) (IDC '17)*. ACM, 509–514. <https://doi.org/10.1145/3078072.3084317>
- [31] Mozilla. 2022. File System Access API. https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API. https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API [Online, accessed 28 July 2022].
- [32] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. 2006. Invited Research Overview: End-User Programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (Montréal, Québec, Canada) (CHI EA '06)*. ACM, 75–80. <https://doi.org/10.1145/1125451.1125472>
- [33] Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books, Inc.
- [34] Klint Paul. 1993. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 2 (1993), 176–201. <https://doi.org/10.1145/151257.151260>
- [35] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennesen, Marie Devlin, and James Paterson. 2007. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (Dundee, Scotland) (ITiCSE-WGR '07)*. Association for Computing Machinery, New York, NY, USA, 204–223. <https://doi.org/10.1145/1345443.1345441>
- [36] Smooth Voxel Playground. 2022. Smooth Voxel Playground. <https://svox.glitch.me/playground.html>. <https://svox.glitch.me/playground.html> [Online, accessed 28 July 2022].
- [37] Thomas W. Price and Tiffany Barnes. 2015. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (Omaha, Nebraska, USA) (ICER '15)*. ACM, 91–99. <https://doi.org/10.1145/2787622.2787712>
- [38] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. (1984), 42–48. <https://doi.org/10.1145/800020.808247>
- [39] Mitchel et al. Resnick. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (2009), 60–67.
- [40] SonarSource SA. 2008. SonarQube. <https://www.sonarqube.org>. [Online, accessed 16 August 2022].
- [41] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software* 145 (2018), 164–179. <https://doi.org/10.1016/j.jss.2018.08.032>
- [42] Andrew Stratton, Chris Bates, and Andy Dearden. 2017. Quando: Enabling Museum and Art Gallery Practitioners to Develop Interactive Digital Exhibits. In *End-User Development*. Springer, 100–107.
- [43] Blockly team. 2022. Create custom blocks: Per-block configuration. https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks?hl=en#per-block_configuration. https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks?hl=en#per-block_configuration [Online, accessed 25 July 2022].
- [44] Blockly team. 2022. Custom Blocks: Style Guide. <https://developers.google.com/blockly/guides/create-custom-blocks/style-guide?hl=en>. <https://developers.google.com/blockly/guides/create-custom-blocks/style-guide?hl=en> [Online, accessed 25 July 2022].
- [45] Mark G.J. van den Brand, Arie van Deursen, Jan Heering, Hayco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 3–8. [https://doi.org/10.1016/S1571-0661\(04\)80917-4](https://doi.org/10.1016/S1571-0661(04)80917-4) LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).
- [46] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (Virtual, USA) (SLE 2021)*. ACM, 12 pages. <https://doi.org/10.1145/3486608.3486908>
- [47] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (Virtual, USA) (SLE 2020)*. ACM, 283–295. <https://doi.org/10.1145/3426425.3426948>
- [48] Mauricio Verano Merino, Jurgen Vinju, and Mark van den Brand. 2021. What you always wanted to know but could not find about block-based environments. (2021). <https://arxiv.org/abs/2110.03073> [Under review at ACM Computing Surveys].
- [49] R. Vinayakumar, K. Soman, and P. Menon. 2018. CT-Blocks: Learning Computational Thinking by Snapping Blocks. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 1–7. <https://doi.org/10.1109/ICCCNT.2018.8493669>
- [50] David Weintrop, Afsoon Afzal, Jean Salac, Patrick Francis, Boyang Li, David C. Shepherd, and Diana Franklin. 2018. Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (Montreal QC, Canada) (CHI '18)*. ACM, 366:1–366:12. <https://doi.org/10.1145/3173574.3173940>
- [51] David Weintrop and Uri Wilensky. 2018. How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction* 17 (2018), 83–92. <https://doi.org/10.1016/j.ijcci.2018.04.005>
- [52] David Wolber, Harold Abelson, and Mark Friedman. 2015. Democratizing Computing with App Inventor. *GetMobile: Mobile Comp. and Comm.* 18, 4 (Jan. 2015), 53–58. <https://doi.org/10.1145/2721914.2721935>