Tips for Creating a Block Language with Blockly

Erik Pasternak
Kids Coding
Google
Mountain View, CA, USA
epastern@google.com

Rachel Fenichel
Kids Coding
Google
Mountain View, CA, USA
fenichel@google.com

Andrew N. Marshall
Kids Coding
Google
Mountain View, CA, USA
marshalla@google.com

***Abstract*—Blockly is an open source library that makes it easy to add block based visual programming to an app. It is designed to be flexible and supports a large set of features for different applications. It has been used for programming animated characters on a screen; creating story scripts; controlling robots; and even generating legal documents. But Blockly is not itself a language; developers who use Blockly create their own block languages. When developers create an app using Blockly, they should carefully consider the style, which blocks to use, and what APIs and language features are right for their audience.**

***Index Terms*—Education; visual programming; computer science; developer tools; language design**

I. Introduction

A. Visual Programming Languages

A Visual Programming Language (VPL) is a programming language that allows a user to create programs primarily through graphical manipulation [1]. Some common interaction models in VPLs are:

- Dragging blocks around a screen (e.g. Scratch [2])
- Using flow diagrams, state diagrams, and other component wiring (e.g. Pure Data [3])
- Using icons or non-text representation (e.g. Kodu [4])

Many VPLs still use text, or combine text with visual representations.

Every VPL has a *grammar* and a *vocabulary*. Together they define the set of concepts that can be easily expressed with the language. The *grammar* is the visual metaphor used by the language: blocks, wires, etc. The *vocabulary* is the set of icons, blocks, or other components that allow you to express ideas.

B. The Blockly Library

Blockly is an open source developer library for adding block based coding to an app. It was first released in May 2012 and remains under active development as of 2017. Blockly provides a block editor UI and a framework for generating code in text-based languages. Out of the box it includes generators for JavaScript, Lua, PHP, Dart, and Python; custom generators for other text languages may also be created.

The core library is written in JavaScript and can be used as part of any website or embedded in a WebView. Native Android and iOS versions of Blockly are also available and provide a subset of features for building high performance mobile apps.

As a library, Blockly is neither a full language nor an app ready for end users. It provides a grammar and a representation for programming that developers can use in their apps. Code is represented by blocks, which may be dragged around the screen. The blocks have connection points where they can be attached to other blocks and chained together.

But Blockly does not provide a full vocabulary of blocks or a runtime environment. Developers need to integrate Blockly with some form of output, build their vocabulary, and decide how the generated code will run.

Building a vocabulary for a block language still depends heavily on context. The level of abstraction and resemblance to other languages may vary dramatically, even between apps using Blockly. Blocks that work in one app often won't work with a different app runtime or in different contexts, such as school vs home use.

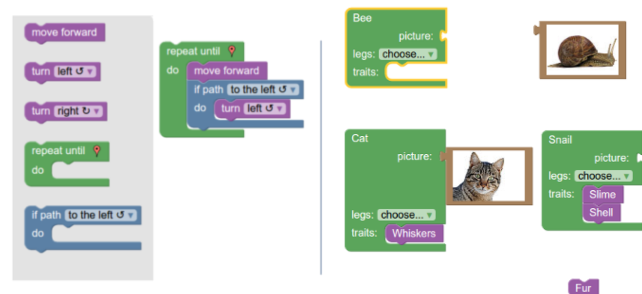

Fig. 1. Blockly Games [5] uses Blockly to teach basic concepts. Left: blocks for solving a maze. Right: blocks for an animal matching game.

In this paper, we will discuss how to think about defining an appropriate set of blocks for an application and audience. We'll focus on blocks that Blockly supports and use existing apps built with Blockly as examples, though much of this discussion is useful for language and app design in general.

II. Questions to Ask Yourself

Before you start writing your app or creating blocks you should take some time to figure out who you are building for and what your overall goals are. There are many different block paradigms, each of which works best under specific conditions.

978-1-5386-2480-7/17/$31.00 ©2017 IEEE

A. Audience

Who are you building for? Blocks for middle school students to control a robot will look very different from blocks for an IT tech to configure a router. Here are a few questions to start with when considering audience:

- What should they get out of using your app?
- How old is your audience?
- What is the reading level of your audience?
- Will they use your app by itself or as part of a lesson plan?
- How much experience do they have with technology?
- Have they seen similar apps before?
- How quickly do they need to learn to use your app?
- Will they use your app once or keep coming back to it?

B. Scope

It's easy to end up with too many blocks and overwhelm your users. In most cases you should keep the set of available blocks small to avoid frustrating your users. Put another way, make things as simple as possible, and then even simpler. Here are a few questions to ask when considering scope:

- What can users do with your app?
- What can't users do with your app?
- What are the user's goals?
- What does each block enable the user to do?
- Can the same action be done with more than one block?
- How long does it take a user to find each block/category when they need it?
- Can a user understand what each block does by looking at it?
- Can a user understand what each block does by running it?

There are some cases where you want to intentionally provide a large number of blocks. App Inventor [6] is an example of using a block language to give easier access to a complex problem—in this case building an Android app. If your app falls in this category, look for other ways to help with discoverability like App Inventor's block search.

It is also possible to add blocks and categories dynamically as the scope of your app expands. If your app uses hardware, you can only show blocks for the currently attached hardware or let your user tell you which components they have.

III. Choosing Your Vocabulary

Even among blocks there are many ways to define a language. The three styles we most frequently see being used for block languages are: icons, natural language, and computer language.

A. Icons

Iconic blocks rely on images instead of text to convey what the blocks do. They may also include numbers and small amounts of text. Iconic blocks can be effective for reaching pre-literate users: children who have not learned to read can still use them to write code. And they can be used when local translations are unavailable.

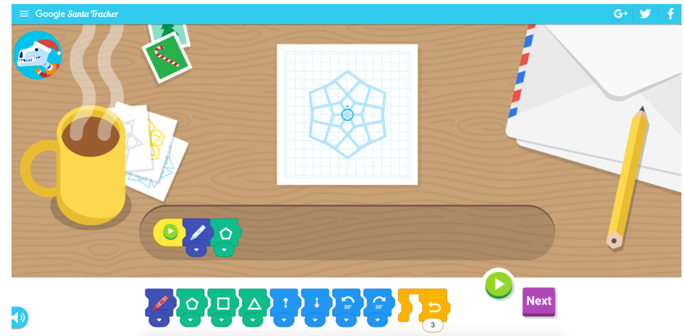

Fig. 2. Code a Snowflake [7] uses iconic blocks with Scratch Blocks [8], a fork of Blockly which includes horizontal and vertical blocks.

While icons can be easier than text for users to understand, icons also have more limitations on what can be expressed. It can be hard to convey abstract concepts with icons, and conditionals can be hard to indicate without text. Even clear icons can be difficult for a user to remember if there are too many of them [9]. For most uses a small set of simple blocks is appropriate.

Iconic languages may be presented horizontally or vertically. Choose based on your screen real estate and the logical flow of your app; bear in mind that horizontal blocks should be rendered right to left (RTL) when your users speak a language that is written RTL.

B. Natural Language

Natural language blocks use standard written sentences for most of their blocks. Blocks that use readable sentences can be used to express more complex concepts than icons, while still feeling familiar and intuitive for users. Sentences are also easier to read and more understandable than code for many users. If well designed, users of any age can find these experiences fun and challenging. Scratch [2], a popular creative coding app for kids and teens, is a great example of a block language in this category with long term appeal.

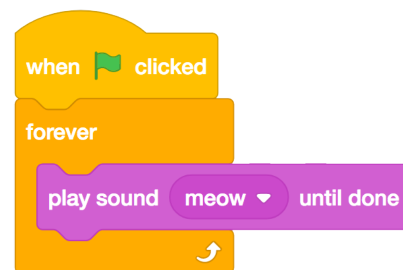

Fig. 3. Blocks from Scratch [2] which read "When flag clicked, forever, play sound meow until done."

Grammar, word choice, and context will all affect how well the user understands your blocks. In general you should use language that is natural for your users and avoid jargon. An exception is if your goal is to transition your users to a text language, in which case some jargon may be appropriate.

Be judicious in your word choice, and include simple and accessible documentation for each block.

C. Computer Language

Another approach is to design blocks that look just like an existing text language. This still removes the most common pain points such as syntax and discoverability [10], [11], and may make the transition to text easier for some students [12], [13].

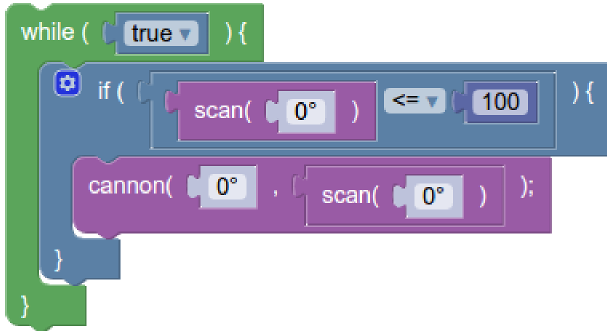

Fig. 4. Blocks from Blockly Games Pond [5]. The text of the blocks is JavaScript, including all of the syntax.

While it can be tempting to provide users with the full power of your target language in block form, doing so can be overwhelming. We recommend carefully choosing which APIs you make accessible. For instance, blocks that look like JavaScript may want to stick to the "good parts" [14].

Avoid APIs that are obscure, hard to understand, or error prone even when used by experienced developers. Providing documentation for all APIs that you use is also a good way to introduce users to online resources for coding.

IV. Other Considerations

A. Consistency

Users will have an easier time with your block language if you are consistent in your use of language and design patterns. We suggest that you:

- Use color to reinforce similarities between blocks.
- Use the same sentence structure across blocks.
- Use the same input order when different blocks use the same inputs.
- Use the same word everywhere when referring to the same thing.

B. Defaults

Many text languages will fail to run if no value is provided (e.g. "var i = ;"). For most apps it is better to avoid these errors by providing default values, either explicitly or implicitly.

Explicit defaults include a visible default value for the input. They are easy for users to edit and show what kind of input is expected. In Blockly these explicit defaults are called shadow blocks. Users can replace them with other blocks; when a covering block is removed, the shadow block reappears. For example, Blockly's repeat block uses a default block to make it easy to loop a set number of times and shows that a number is expected:

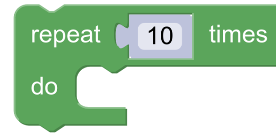

Fig. 5. A repeat 10 times block from Blockly. The 10 can be edited directly or replaced with another block.

You can also provide an implicit default, by leaving an input blank and handling the default when generating or running the code. Implicit defaults may be better when the default doesn't do anything useful, such as in the block below. The empty space prompts the user to put something there. Scratch has several more examples of defaults [15].

Fig. 6. A "for each" block from Blockly. The block input will default to an empty list, but since that won't do anything, the input is left empty to encourage the user to add a block.

C. Testing

Even if you carefully consider all of the trade-offs for your language, there is no substitute for putting it in front of users and seeing what they do. Starting early with paper or other low-fidelity prototypes can catch most problems [16] and it's easier to make changes when you're less invested in a specific solution [17]. And A-B testing alternative blocks or subsets of blocks can help find the vocabulary that makes the most sense to your users.

V. Closing

Blockly lets you easily create a drag and drop programming app, but designing a block language requires significant consideration. In this paper we presented some questions to guide your design, and some best practices to follow. Each application has a different audience and a different set of goals; allow that to drive the design of your blocks.

There is a lot more to designing a block language than discussed here. There are plenty of other resources for learning about API design, many of which apply to block languages as well.

Other developers can also be great resources. Don't hesitate to ask a question on Blockly's mailing list [18]!

References

[1] Visual Programming Language. Retrieved July 17, 2017 from https://en.wikipedia.org/wiki/Visual_programming_language

[2] Scratch - Imagine, Program, Share. Retrieved July 16, 2017, from https://scratch.mit.edu/

[3] Pure Data. Retrieved July 17, 2017, from https://puredata.info/

[4] Kodu. (2010, April 5). Retrieved July 17, 2017, from https://www.microsoft.com/en-us/research/project/kodu/
[5] Blockly Games, Retrieved July 16, 2017, from https://blockly-games.appspot.com/
[6] MIT App Inventor. Retrieved July 16, 2017, from http://appinventor.mit.edu/explore/
[7] Code a Snowflake. (2016, December 4). Retrieved July 16, 2017, from https://santatracker.google.com/snowflake.html
[8] LLK/scratch-blocks. Retrieved July 16, 2017, from https://github.com/llk/scratch-blocks
[9] G.A. Miller, ”The magical number seven, plus or minus two: some limits on our capacity for processing information.” Psychological Review. 1956;63(2):81-97. doi:10.1037/h0043158.
[10] M.C. Jadud, ”A first look at novice compilation behaviour using bluej,” Computer Science Education, vol. 15, no. 1, pp. 2540, 2005.
[11] A.J. Ko, B.A. Myers, and H.H. Aung, ”Six learning barriers in end-user programming systems,” in 2004 IEEE Symposium on Visual Languages and Human Centric Computing, 2004, pp. 199 206.
[12] C. Kelleher and R. Pausch, ”Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers,” ACM Comput. Surv., vol. 37, no. 2, pp. 83137, 2005.
[13] Weintrop, D. Minding the gap between blocks-based and text-based programming. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (2015), ACM, 720.
[14] D. Crockford, JavaScript : The Good Parts. Sebastopol :O’Reilly, 2008. Print.
[15] Default Value. Retrieved July 16, 2017, from https://wiki.scratch.mit.edu/wiki/Default_Value
[16] N. Heaton, ”What’s wrong with the user interface: how rapid prototyping can help,” IEE Colloquium on Software Prototyping and Evolutionary Development, London, 1992, pp. 7/1-7/5.
[17] E. Gerber and M. Carroll, (2012). ”The psychological experience of prototyping,” Design Studies, 33(1), 2012, pp. 64-84.
[18] blockly@googlegroups.com Retrieved July 10, 2017, from https://groups.google.com/forum/#!forum/blockly