Categories and Completeness of Visual Programming and Direct Manipulation

Michael J. McGuffin
École de technologie supérieure
Montreal, Canada
michael.mcguffin@etsmtl.ca

Christopher P. Fuhrman
École de technologie supérieure
Montreal, Canada
christopher.fuhrman@etsmtl.ca

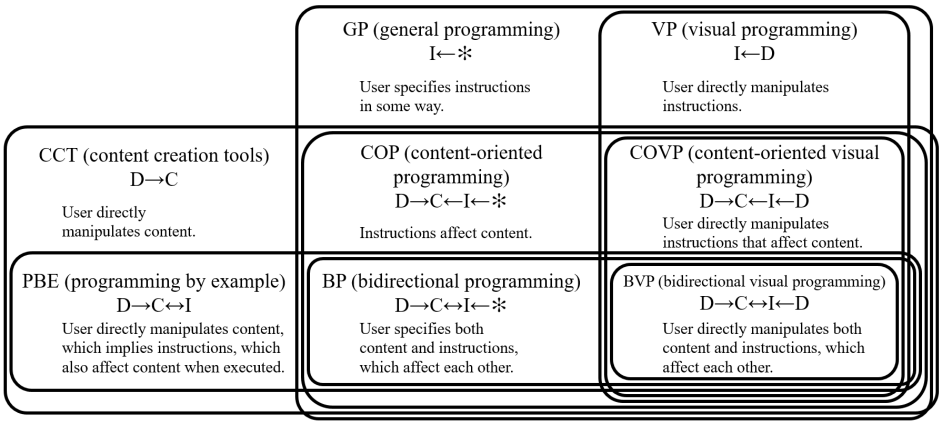

Figure 1: A taxonomy of overlapping categories. Each category contains all systems that have at least the indicated architectural features. The lower-right category (BVP) is the most restrictive, requiring the most features. Input can be Textual (T), Direct Manipulation (D), or either (*), and input is used to define Content (C) or Instructions (I).

ABSTRACT

Recent innovations in visual programming and the use of direct manipulation for programming have demonstrated promise, but also raise questions about how far these approaches can be generalized. To clarify these issues, we present a categorization of systems for visual programming, programming-by-example, and similar systems. By examining each category, we elucidate the advantages, limitations, and ways to extend systems in each category. Our work makes it easier for researchers and designers to understand how visual programming languages (VPLs) and similar systems relate to each other, and how to extend them. We also indicate directions for future research.

CCS CONCEPTS

• **Software and its engineering** → **Visual languages**; Patterns;
• **Human-centered computing** → *Graphical user interfaces.*

KEYWORDS

visual programming languages, direct manipulation, programming by example, output-directed programming, taxonomy, programming by demonstration

ACM Reference Format:
Michael J. McGuffin and Christopher P. Fuhrman. 2020. Categories and Completeness of Visual Programming and Direct Manipulation. In *International Conference on Advanced Visual Interfaces (AVI '20), September 28-October 2, 2020, Salerno, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3399715.3399821

AVI '20, September 28-October 2, 2020, Salerno, Italy

ACM ISBN 978-1-4503-7535-1/20/09…$15.00
https://doi.org/10.1145/3399715.3399821

1 INTRODUCTION

Visual programming languages (VPLs) and end-user programming systems have been in development for decades. Contemporary examples deployed for real-world applications include Scratch [25] and related systems [2, 33, 40], 'visual scripting' with Bolt [24] in the Unity game engine, and PureData [34] for music. VPLs often consist of either *instruction blocks* (that are dragged and snapped together into sequences), or *flow networks* (blocks that are connected together as nodes in a directed graph or dataflow diagram [16]). Recently, other experimental systems have been proposed that allow a user to program through variations of direct manipulation, drawing, or sketching. However, these systems are not based on the more common paradigms of *instruction blocks* nor *flow networks*. Some of these have been described as "programming by example" [39] or "output-directed programming" [11].

These VPLs and related systems often make it easier for beginners to program, and some can also ease the work of experts. VPLs also often appear to have limitations, e.g., only working for the limited domain for which they were designed, with no obvious way to extend them. In this sense, they appear incomplete as languages.

We present a way to categorize and relate these different systems. We start with the dichotomy of a program's *instructions and data*, which can often be understood as *instructions and content*, where content could be a 2D or 3D scene, or a document, etc. We then consider the different ways that content and instructions can each be specified by a user, through direct manipulation or other means, and how content and instructions can affect each other in a given system. This leads to a taxonomy (Figure 1) of 8 overlapping categories of systems. By examining each category, we elucidate the advantages, limitations, and ways to extend systems in each category. Our work makes it easier for researchers and designers to understand how VPLs and related systems pertain to each other, and how to extend them.

2 MOTIVATING EXAMPLES

Figures 2-4 show a few examples of recent work. These motivate some questions that our paper will answer.

Figure 2: In Scratch [25], users create *sprites* on a *stage* (top right). Behaviors are programmed by dragging together instructions resembling jigsaw puzzle pieces (left).

In Scratch (Figure 2), the user populates a *stage* with one or more *sprites*. Sprites can be directly manipulated to give them an initial position. The user also assembles sequences of instructions that are executed in response to *events* (such as keyboard or mouse events). These instructions implement behaviors of the sprites, enabling the user to animate a story or program an interactive game.

Sketch-n-sketch (Figure 3) produces a static vector drawing as output which can be created and modified through direct manipulation. Textual source code is also shown that generates the same output. Modifying either the textual code or the graphical output causes the other to update.

Victor [39] has demonstrated a tool for 'drawing' visualizations through direct manipulation. Each manipulation of the drawing by the user causes an instruction to be generated. The user can thus build up an algorithm, instruction-by-instruction, mostly through direct manipulations (as well as occasional modifications to instructions by dragging in parameters or typing in expressions). The

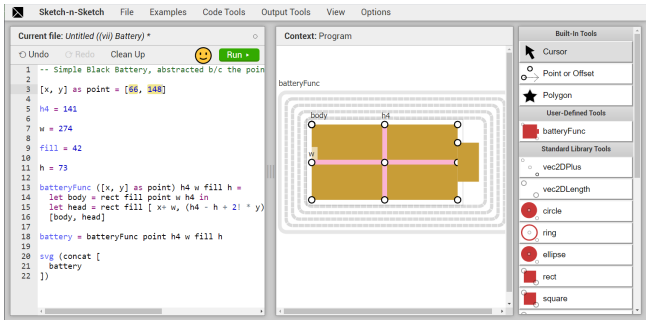

Figure 3: A user of Sketch-n-sketch [5, 11] creates an SVG figure using textual source code (left) or direct manipulation of the SVG output (right). Changes to either representation cause the other to update, maintaining synchronization.

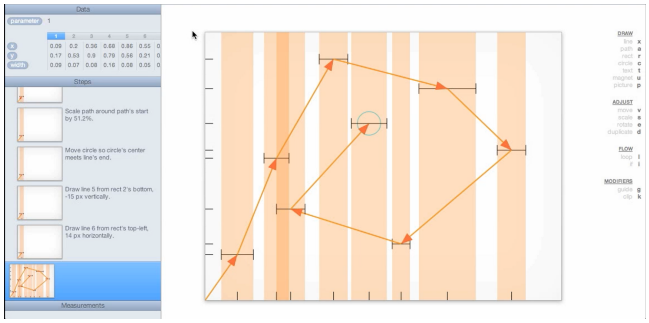

Figure 4: In Victor's system [39], the user directly manipulates the desired output (right). Each manipulation causes a corresponding instruction to be generated (left). Instructions can be enclosed in a loop, causing them to execute on all tuples in a dataset, generating a visualization of the data.

algorithm thus defined can then be executed on different datasets, causing the output visualization to update.

Several observations can be made. First, in each of the preceding three examples, we can see a separation between *instructions* in one part of the window, and *content* in the other. Second, in the case of Scratch, the instructions (blocks) and content (sprites) play different roles: the instructions define the *behavior* of the content. However, in the other two examples, the instructions and content play the *same* role, as they each define the same thing using different representations. Third, once defined, the content can be either dynamic (Scratch) or static (the other two examples). Fourth, programming can be done by editing instructions (either through text editing, as in the case of Sketch-n-sketch, or through direct manipulation, as in Scratch) or by directly manipulating content. In the case of Scratch, a typical programming session involves changing *both* instructions and content; however in the other two systems, it is possible to program almost entirely by editing just content.

Several questions arise. Are each of these systems limited to one application domain? For example, can Sketch-n-sketch or Victor's system be extended to allow a user to program interactive behaviors? How could a user 'draw' interactive behavior? Are any

of these systems "Turing complete"? [1] Can they be extended for general purpose programming? What are the pros and cons of these different approaches to programming? Our paper answers these questions, by classifying systems such as these, and by pointing out the limitations and possibilities within each category of systems.

3 TAXONOMY

VPLs and related systems often involve sequences of instructions (I) as well as content (C).

By **content**, we mean part of the data used by a program. It is sometimes called "resources" or "assets", with examples including 2D figures or images, 3D scenes, and documents. In some cases, people instead speak of the program's "output": for example, in a 3D maze game, the 3D scene could be thought of as output; however, we will refer to it as content. In many cases, content can be created or edited by the programmer through direct manipulation (D) (often via a mouse), and we indicate this with $D \rightarrow C$.

Instructions are often specified through textual (T) input, i.e., primarily through a keyboard, and we can indicate this with the formula $T \rightarrow I$. Some systems (like Scratch) allow instructions to be specified through direct manipulation ($D \rightarrow I$). To indicate the more general idea that instructions are specified through *some* means of input, we can write $* \rightarrow I$, where $*$ indicates T or D. As a matter of convention, we will usually write these formulas in the opposite direction ($I \leftarrow T$, $I \leftarrow D$ or $I \leftarrow *$), because this will make it easier to include them in larger formulas later. However, this does not change their meaning.

These tiny formulas are highly simplified models of the architecture of the systems in question. In some systems, I or C can have effects on each other, which can also be indicated with arrows, as we explain later. We have catalogued such simple architectural models to derive the taxonomy of categories shown in Figure 1. In the following sections, we discuss each category, starting with General Programming (GP) and Content Creation Tools (CCT). The other categories in Figure 1 require additional features, and are incrementally more restrictive than GP or CCT.

3.1 GP (General Programming) and GUI-completeness

GP is the set of all systems allowing the programmer to create or edit instructions ($I \leftarrow *$), including through the use of text input ($I \leftarrow T$). For simplicity, we focus on instructions in imperative programming languages.

It is well known that for an imperative programming language to be **Turing complete**, i.e., able to express any algorithm, it must support conditional branching (or 'while' loops) and be able to access an arbitrary amount of memory. It is reasonable to expect that any language designed for general use would also support: 'for' loops; a few basic datatypes (numeric and string types, as well as lists or arrays) and operations on them; the ability to dynamically (i.e., at runtime) create, modify, and destroy instances of these datatypes; and some mechanism for defining subroutines to allow the reuse of sequences of instructions. We are furthermore interested in making programs that accept input from common devices (mouse, keyboard, touch ...), and this is often possible in imperative programming by defining *events* of different types (keypress event, mouse motion event, etc.) that each trigger some subset of instructions. Finally, we are interested in making programs that can display arbitrary visual output, either in the form of vector graphics or bitmaps, that can change as a function of time and also as a function of input events. Such time-dependent visual output is normally supported in a language by giving the programmer an API to create or draw graphical primitives, as well as some mechanism to control the timing of output (e.g., via timer events, and/or a way to 'sleep' for a desired time interval). We propose the term **GUI-complete** to describe programming languages that support all these features, since they can be used to implement any GUI (Graphical User Interface). More formally, let F be a computable function that maps a memory state M_{t-1} at time $t-1$ to a new memory state M_t and to visual output V, i.e., $(M_t, V) = F(M_{t-1}, E)$, where E is the set of input events that have occurred over the $(t-1)$th unit of time, and the units of t appear psychologically 'small' to a human user (e.g., milliseconds). We say that a programming language or programming system is GUI-complete if and only if, for any such F, the language allows a program to be defined that captures E and displays V, at each time step. This implies that the program computes F and stores M_t.

Additional features can be imagined to make a programming language more useful and usable, such as: an API for file i/o, or for network communication, or syntax-related features such as object-oriented constructs, etc. The point is that the above set of features already allows a programming system to express *any* interactive front-end, and once these basic features are supported, it is easy to add additional features of the kind just mentioned, either with an API or new syntactical elements.

3.2 CCT (Content-Creation Tools) and Direct Manipulation

CCT contains systems for creating content, such as 2D paint programs, or software for creating animations, 3D scenes, etc. These systems normally allow the user to manipulate content through **direct manipulation** [37, 38], which can be defined as the use of continuous physical motions (of one's hand) to interactively manipulate persistent visual representations of objects, with continuously updated feedback, and with the ability to undo actions by simply reversing physical motions. Examples include using a mouse to "drag and drop" an icon to move it from one place to another, or dragging an object or part of an object to move or resize it. We indicate such systems in CCT with the formula $D \rightarrow C$.

Systems in CCT are not limited to static content, and it is instructive to consider the ways in which *dynamic* content (i.e., animations) can be created with tools in CCT. One method, common in animation software, allows the user to directly manipulate curves representing motion paths, or curves in spacetime, to define movements over time. Such curves can also be implied through interpolation of 'keyframes' at different moments in time where the user has fixed the position and state of each object. Another method is to have objects with built-in dynamic behaviors (e.g., moving under

[1] A researcher and game designer posed this question online in 2015: "Have there been any breakthroughs in Turing complete visual languages for games so gamers can add their own modes, levels, objects, etc.? I am particularly interested in visual languages where the visual program looks a lot like what the game looks like."[28]

the effect of forces, or reacting to collisions, or reacting with pre-programmed behaviors), allowing the user to assemble them and to launch a simulation [1]. These built-in behaviors were defined by some prior programmer, and there may be parameters exposed to the content creator to modify these behaviors; however there is no requirement for a system in CCT to allow the content creator to re-program such behaviors: CCT is merely defined to contain systems that allow a content creator to directly manipulate the content.

Using the methods just considered, any animation (i.e., any fixed sequence of images) can be created, and indeed these methods are routinely used to create animated feature films. However, these methods are *not* GUI-complete. Given any particular content-creation tool T that only supports high-level pre-defined behaviors, it is easy to imagine an interactive program P whose dynamic output depends in a complicated algorithmic way on a user's moment-to-moment input (from a mouse or other device), such that there is no way to define P using T.

As we will see, it turns out that *some* systems in CCT *can* achieve GUI-completeness, if the step-by-step manipulations of content performed by the user imply instructions of a sufficient flexibility. This is explained later in our discussion of the BP category, a subset of CCT.

One theme in research on content-creation systems has been the use of pen input to enable a user to sketch content [1, 41]. This has the advantages of better leveraging the fine motor control afforded by fingers; of encouraging a less formal, more creative way of working; and of allowing the user to fluidly intermix inking and command gestures while avoiding explicit mode changes. Recent systems allow animations [18, 42] and even interactive illustrations [17] to be defined through sketch-based input, but without being GUI-complete.

3.3 VP (Visual Programming)

VP is a subset of GP where the primary means to define instructions is through direct manipulation ($I \leftarrow D$) rather than text input. This includes Blockly [33] which is similar to the way instructions are assembled in Scratch (Figure 2). VP also includes programming systems where the user constructs a control-flow diagram, such as in Lego Mindstorms EV3, and data-flow diagrams [8, 9, 13, 34].

```
Start() {
   int j = 1;
   while ( j*j <= 100 ) {
      print(string.Format(
         "The square of {0} is {1}",
         j, j*j ));
      j = j + 1;
   }
}
```

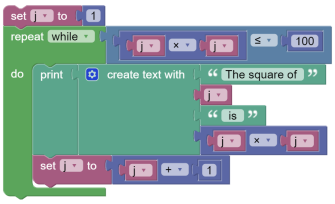

Figure 5: Left: textual instructions. Right: equivalent instructions in Blockly [33].

There is an obvious mapping between text-based source code and VPLs in the style of Blockly (Figure 5). This makes it clear how to make such a VPL GUI-complete, simply by ensuring that all the necessary features (dynamic creation of variables, support for input events, ability to draw arbitrary visual feedback,...) are available to the programmer. GUI-completeness may not be desirable, however, if the VPL is designed for teaching programming and should be kept simple.

The main advantage of a VPL is ease-of-use, especially for beginner programmers. With traditional textual source code, programmers often need to have a precise idea of what to type as they enter each part of the program. In a VPL, however, there is often a menu of possible blocks that programmers can choose from. This ensures *visible affordances*, meaning programmers can depend more on *recognition rather than recall* to determine what block to use next. The shape or color of the blocks can also give some indication of how to use them with other blocks, sometimes through the use of *metaphors* (e.g., jigsaw puzzle pieces with matching contours). In addition, some VPLs (e.g., Figure 6) use curves or line segments to connect blocks, and these connections may be equivalent to using temporary variables in a text-based language to store input and output values. The ability to connect one block's output to another block's input, without choosing an explicit name for a temporary variable, is another advantage with VPLs. Disadvantages of VPLs include being slower to input instructions than typing, requiring more screen space that textual instructions, and the potential for large numbers of criss-crossing lines [36] (however this can be mitigated by allowing the user to define variables and refer to them by name rather than always connecting them with explicit lines, and also allowing for some kind of encapsulation similar to subroutines).

With some VPLs, it is less obvious if, and how, it is possible to achieve GUI-completeness. Purely dataflow-based VPLs operating on a *limited* amount of data, or supporting no conditional branching or loops [29], cannot be Turing complete, and therefore neither GUI-complete. VPLs like Blueprints in Unreal Engine and Bolt [24] in Unity, interestingly, use a mix of control-flow and data-flow diagrams. Figure 6(top) shows one such diagram which looks very similar to the abstract syntax tree of the instructions in Figure 5. Next, Figure 6(bottom) shows how duplicate subtrees in Figure 6(top) can be used multiple times, resulting in a more compact diagram. This kind of VPL nevertheless directly maps back to the instructions in Figure 5, and therefore can be made GUI-complete in the same way.

3.4 PBE (Programming-By-Example)

We define PBE as the subset of CCT containing systems where the user directly manipulates content, and this somehow implies instructions that can later be executed. There is no requirement for the user to have direct access to modify the instructions, so we model PBE with the formula $D \rightarrow C \leftrightarrow I$. The double-headed arrow between C and I indicates that instructions are implied by changes to content ($C \rightarrow I$), and that when instructions are executed they have read/write access to content ($C \leftarrow I$).

Viscuit (Figure 7) is an instructive example of the kind of system found in PBE. One paper on Viscuit [10] describes it as a "visual langauge" in which "programs are written"; however, we do not consider it a member of our VP nor even GP categories, because users have no way to directly edit instructions. Instead, users only interact with content in Viscuit. They do this by drawing sprites, and also by defining "rewriting" rules. When a Viscuit program

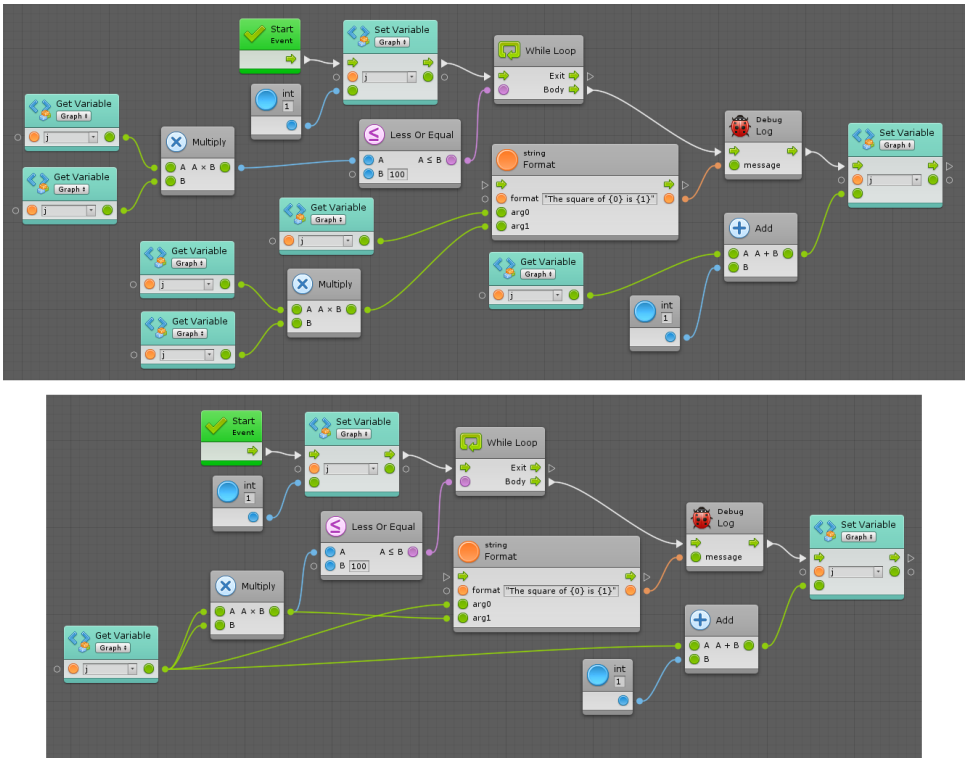

Figure 6: In each of the above Bolt 'visual scripts' [24], the 5 nodes running along the top contour (connected by white arrows) map to the instructions in Figure 5, and the remaining nodes serve to evaluate expressions. Top: the tree structure of this visual script maps to the abstract syntax tree for the instructions in Figure 5. Bottom: this has an equivalent structure, but duplicate expression subtrees (such as j*j) have been collapsed.

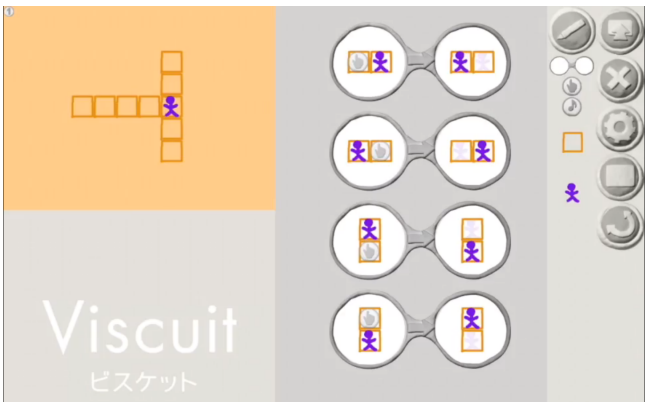

Figure 7: In Viscuit [10], the user defines rewriting rules for sprites. In this example, the rewriting rules react to touch events on each side of a sprite, causing the sprite to move.

is run, at each time step, the rewriting rules determine how the simulation advances, causing sprites to move, rotate, change, or respond to touch input. Animations and games of a surprising variety can be created with Viscuit, with the user only ever manipulating representations that resemble pieces of the program's final output.

However, Viscuit, and systems like it, cannot be GUI-complete. The limitation stems from users having no (direct nor indirect) way to create instructions corresponding to loops, nor instructions for reading/writing arbitrary amounts of memory, nor ways to draw arbitrary visual feedback. The functionality that *can* be accessed through interaction with the content is similar to the functionality available with a fixed high-level API: although the API makes it easy to define certain animations and games, because there is no way to access lower-level primitives, the user can never break out of the limits of the higher-level functionality that is "baked-in" (predefined).

One advantage of PBE is that programming by manipulating content reduces the *representational gap* between the target application domain and the input. Other examples in PBE include systems for defining prototypes of GUIs [23] and of websites [32], and for defining visualizations [31, 35].

3.5 COP (Content-Oriented Programming) and COVP (Content-Oriented Visual Programming)

It is now common for programming systems to involve both instructions and content of some kind, each of which can be edited separately. We define COP as the set of such systems, which we model with $D \rightarrow C \leftarrow I \leftarrow *$. The $C \leftarrow I$ means that, when

instructions are executed, they can read and write to the content, modifying it, inserting/deleting parts, etc.

COP includes IDEs (Integrated Development Environments) where images (such as icons) can be edited through direct manipulation, possibly in another program and then imported (as "assets" or "resources") into the IDE, and then the source code instructions can load this content and use it, render it, or transform it at runtime. Another example in COP is web programming, where an HTML document can be created through a direct manipulation tool, and then JavaScript instructions are written to modify the HTML document. A third example is found in the Unity game engine: a 3D scene is created through direct manipulation, and C# instructions can modify this scene at runtime or add behavior.

COVP is the more restrictive category of systems where *both* content and instructions are modified through direct manipulation ($D \rightarrow C \leftarrow I \leftarrow D$). Examples include Scratch, AppInventor [40] (for mobile apps), Alice [7] (for 3D scenes), and MaggLite [15].

Systems in COP and COVP can be made GUI-complete simply by ensuring their instruction set covers all the necessary functionality for this.

3.6 BP (Bidirectional Programming)

The set BP is the intersection of COP and PBE, combining aspects of each. In BP, instructions can be edited directly, or implied by the editing of content. In addition, instructions act on content when they are executed. This is expressed by $D \rightarrow C \leftrightarrow I \leftarrow *$. Our use of the term "bidirectional programming" is inspired by its use to describe Sketch-n-sketch [4]; however, here we mean it in a larger sense, as will become clear with the examples we now give.

A first example of systems in BP are "GUI builders" or "GUI designers" that allow a programmer to build a front-end through direct manipulation, e.g., dragging-and-dropping widgets into place on a window. Some of these tools then generate source code that can be executed to instantiate the same window and widgets. If we view the GUI as content, then direct manipulation of the content implies instructions ($D \rightarrow C \rightarrow I$), and these instructions (1) recreate the same content, and (2) the instructions can be further edited by the programmer to modify the content or add behavior (event-triggered subroutines) to the content ($C \leftarrow I \leftarrow *$).

A second example: Autodesk Maya is a highly customizable 3D modeling and animation software package. 3D content can be created through direct manipulation, and also created programmatically through a scripting language. Maya has a feature called "echo all commands". When activated, every direct manipulation of the 3D scene causes an equivalent script command to be printed to a terminal. These can then be saved as part of a script, and optionally further edited. When this script is executed, it can recreate the same modifications to the 3D scene.

As a third example, consider Victor's work [39] (Figure 4), where each direct manipulation of 2D content generates another instruction, and these instructions set the position, size, or other attributes of 2D shapes. Some instructions contain parameters which the programmer can change by typing in expressions or dragging-and-dropping arguments into the parameters. To create a loop, the programmer selects a contiguous set of instructions and presses a keyboard shortcut ("L"). The system also maintains a sequence of thumbnails, in chronological order, showing the result of each direct manipulation. This allows the user to go back to any previous step and change it or insert new steps.

Sketch-n-sketch is a fourth example in the BP category, and it maintains a fairly strict synchronization between the instructions and 2D content, when either are edited.

One advantage of these bidirectional programming systems is allowing the programmer to switch between two different representations of the program, according to whichever representation is easier to work with.

In the four examples given above, notice that the content is either static (Victor's work and Sketch-n-sketch) or else interactive behaviors can only be defined with instructions (GUI builders, Maya). It is not obvious how direct manipulation of content can be used to generate instructions that define interactive behavior, or how *abstract* data can be directly manipulated. However, by extending the scheme in Victor's system, we can outline a strategy for this.

Figure 8 shows the strategy. If the objects that the programmer needs to manipulate are visible as part of the content, the programmer can use these as handles to access commands (i.e., verbs) associated with them. For some applications, it makes sense for the position or size of the object's visible form to map to data values in the program, in which case directly manipulating the position or size can generate corresponding instructions (this happens in Victor's work, in Sketch-n-sketch, and in Figure 8, upper right). However, for objects and/or verbs that are abstract, simply making the object visible in the content, and listing verbs in a menu, is sufficient to generate corresponding instructions. Generating instructions involving multiple objects could be done with drag-and-drop actions, lasso/multiple selections, or with additional menus (Figure 8, lower left).

This strategy allows a programmer to generate sequences of instructions, involving arbitrary sets of objects and verbs, just by manipulating content. Different sequences of instructions could be triggered by different input events, to allow the programmer to define interactive behavior. If, in addition, some mechanism is provided to define loops, conditionals, and subroutines (such as the selection of multiple instructions followed by hitting a keyboard shortcut in Victor's work), then this strategy, of programming (mostly) through direct manipulation of content, *can be made GUI-complete.*

Other examples of previous work approach the strategy just outlined. Greenfoot [12] is a Java development environment that facilitates the creation of visual *microworlds*. It makes use of the same live interaction of Java objects seen in BlueJ [22]. Both Greenfoot and BlueJ allow methods to be invoked on objects through popup menus on the visual representation of objects. BlueJ also has a feature to facilitate creating JUnit tests, by recording the interactions with Java objects, similar to how macros record steps in other environments. They can be replayed later, as regression tests. This could also be viewed as a way to create instructions by visually interacting with live objects.

Generating instructions via content ($D \rightarrow C \rightarrow I$) would be even more useful if there were better ways to show data structures (arrays, lists, trees, networks, ...) as visual content that could be directly manipulated. Imagine if the programmer could indicate, in a visual representation of a list, where to move or insert a new

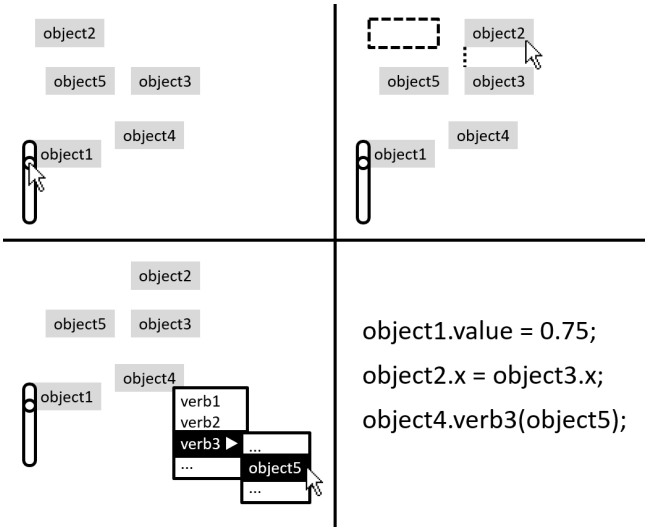

Figure 8: Direct manipulation of content generating instructions, step-by-step. Upper left: dragging the slider sets object1's value. Upper right: dragging object2 to the right aligns it with object3. Lower left: right-clicking on object4 pops up a menu of verbs on that object, with a submenu to select an argument to the verb. Lower right: the instructions generated by these actions.

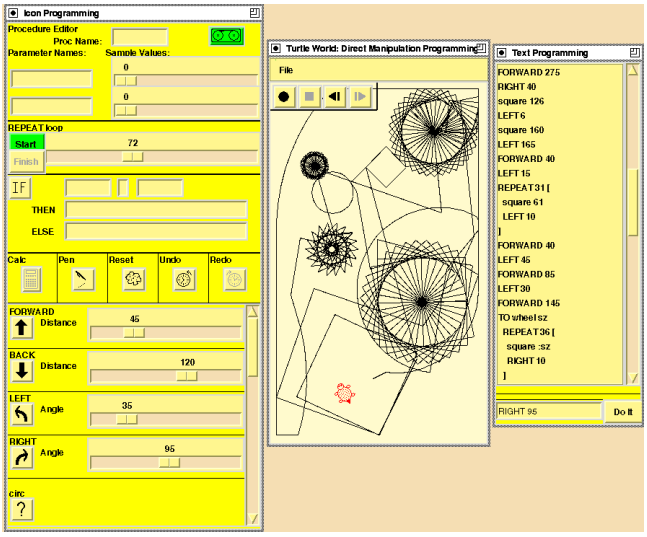

Figure 9: Leogo [6] supports three kinds of input: an "iconic language for graphical programming", direct manipulation of content, and textual input of instructions.

item, without the danger of one-off indexing errors that are common when editing textual source code. Some previous work has demonstrated direct manipulation of data structures that generates instructions, for binary trees [27], for arrays [14] and for red-black trees [26], but a more general approach for a larger variety of data structures has not been demonstrated to our knowledge.

Taking a step back, we can broadly distinguish 3 ways of programming: (1) textual input of instructions ($I \leftarrow T$), (2) VPL-style direct manipulation of instructions ($I \leftarrow D$), and (3) direct manipulation of content implying instructions with perhaps additional editing directly of instructions ($D \rightarrow C \leftrightarrow I \leftarrow *$). Each of these approaches can be made GUI-complete, and each has its advantages. The first approach can work well for expert programmers, as they can type quickly *if* they know what they should type, and also gives them access to a large scope of possible objects and verbs. An IDE can help by popping up a menu of possible text auto-completions as the user types. The second approach can make programming easier for a beginner, by offering menus of possible verbs and objects, enabling the programmer to rely more on recognizing what they need rather than recalling. In the third approach (Figure 8), any object that is visible in the content can be manipulated, which can be easier to understand, especially if the content is presented in a way that makes it easier to understand the relationships between objects. However, this approach may not scale as well to a large number of objects.

3.7 BVP (Bidirectional Visual Programming)

The category BVP is a straightforward specialization of BP where instructions are directly manipulated ($D \rightarrow C \leftrightarrow I \leftarrow D$). Such systems could offer two kinds of advantages to beginner programmers: easier manipulation of instructions ($I \leftarrow D$), as well as direct manipulation of visual representations of data ($D \rightarrow C \rightarrow I$). The only example we found of a system in BVP is Leogo [6] (Figure 9).

4 PREVIOUS WORK

Among previous work on VPLs, the most recent taxonomies we could find are at least 15 years old [3, 19, 30]. Ko et al.'s survey [21] is more recent but does not present a taxonomy of VPLs based on their syntax or visual form. We found no previous taxonomies based on the architecture of the programming system as our paper presents. The only work we found that discusses Turing completeness of VPLs is Kiper et al. [20], but their work does not give strategies for achieving completeness; instead they propose completeness as one of several criteria for evaluating VPLs.

5 FUTURE DIRECTIONS

The tradeoffs between different representations and approaches to programming suggests that there is no single best way to program. An intriguing vision for the future is to have an IDE where the programmer is free to define multiple portions of the source code that are each related (with bidirectional updating) to alternative visual representations that are most appropriate for each portion of code. The programmer would be free to switch between representations, or define new representations, according to their needs. This could combine the advantages of projectional editors (such as JetBrains MPS[2]) with bidirectional programming.

[2]https://youtu.be/iN2PflvXUqQ

ACKNOWLEDGMENTS

This research was funded by NSERC. Many thanks to Brian Hempel for insightful email discussion of certain issues and for pointers to related work.

REFERENCES

[1] Christine Alvarado and Randall Davis. 2007. Resolving Ambiguities to Create a Natural Computer-Based Sketching Environment. In *ACM SIGGRAPH 2007 Courses*. ACM.

[2] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Communications of the ACM (CACM)* 60, 6 (2017), 72–80.

[3] Margaret M. Burnett and Marla J. Baker. 1994. A classification system for visual programming languages. *Journal of Visual Languages and Computing* 5, 3 (1994), 287–300.

[4] Ravi Chugh. 2016. Prodirect manipulation: bidirectional programming for the masses. In *Proc. International Conference on Software Engineering Companion*. 781–784.

[5] Ravi Chugh. 2016. Sketch-n-Sketch: Interactive SVG Programming with Direct Manipulation. Strange Loop Talk https://youtu.be/YuGVC8VqXz0.

[6] Andy Cockburn and Andrew Bryant. 1997. Leogo: An equal opportunity user interface for programming. *Journal of Visual Languages & Computing* 8, 5-6 (1997), 601–619.

[7] Stephen Cooper, Wanda Dann, and Randy Pausch. 2003. Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin* 35, 1 (2003), 191–195. http://www.alice.org/.

[8] Pierre Dragicevic and Jean-Daniel Fekete. 2004. Support for input adaptability in the ICON toolkit. In *Proc. Int. Conf. Multimodal Interfaces (ICMI)*. ACM, 212–219.

[9] Barrett Ens, Fraser Anderson, Tovi Grossman, Michelle Annett, Pourang Irani, and George Fitzmaurice. 2017. Ivy: Exploring spatially situated visual programming for authoring and understanding intelligent environments. In *Proc. Graphics Interface (GI)*. https://youtu.be/Gn8sBvKHHYk.

[10] Yasunori Harada and Richard Potter. 2003. Fuzzy rewriting: Soft program semantics for children. In *IEEE Symp. Human Centric Computing Languages and Environments*. 39–46. https://www.viscuit.com/asobikata/.

[11] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *ACM Symp. User Interface Software and Technology (UIST)*. 281–292. https://youtu.be/i48KQqPwaAA.

[12] Poul Henriksen and Michael Kölling. 2004. greenfoot: combining object visualisation with interaction. In *Companion to ACM SIGPLAN Conf. Object-oriented programming systems, languages, and applications*.

[13] Valentin Heun, James Hobin, and Pattie Maes. 2013. Reality editor: programming smarter objects. In *Proc. ACM Conf. Pervasive and Ubiquitous Computing (UbiComp) Adjunct Publication*. 307–310. http://realityeditor.org/.

[14] Christopher D Hundhausen and Jonathan L Brown. 2007. What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing* 18, 1 (2007), 22–47. https://youtu.be/_zihXBIjofs.

[15] Stéphane Huot, Cédric Dumas, Pierre Dragicevic, Jean-Daniel Fekete, and Gérard Hégron. 2004. The MaggLite post-WIMP toolkit: draw it, connect it and run it. In *ACM Symp. User Interface Software and Technology (UIST)*. 257–266. https://web.imt-atlantique.fr/x-info/magglite/.

[16] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. 2004. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)* 36, 1 (2004), 1–34.

[17] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: sketching dynamic and interactive illustrations. In *ACM Symp. User Interface Software and Technology (UIST)*. 395–405. https://youtu.be/mqQO-IoG4qw.

[18] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: bringing life to illustrations with kinetic textures. In *ACM Conf. Human Factors in Computing Systems (CHI)*. 351–360. https://youtu.be/l84YK1_ytks.

[19] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)* 37, 2 (2005), 83–137.

[20] James D Kiper, Elizabeth Howard, and Chuck Ames. 1997. Criteria for evaluation of visual programming languages. *Journal of Visual Languages & Computing* 8, 2 (1997), 175–192.

[21] Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011).

[22] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ system and its pedagogy. *Computer Science Education* 13, 4 (2003), 249–268. https://youtu.be/3dCVERWX9Z8.

[23] James A Landay and Brad A Myers. 2001. Sketching interfaces: Toward more human interface design. *IEEE Computer* 34, 3 (2001), 56–64.

[24] Ludiq. 2020. Bolt: Visual Scripting for Unity. https://ludiq.io/bolt.

[25] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15. https://scratch.mit.edu/.

[26] Sean McDirmid. 2019. rbRebalance. https://youtu.be/O2-8uEEtvso.

[27] Amir Michail. 1996. Teaching binary tree algorithms through visual programming. In *IEEE Symp. Visual Languages (VL)*. 38–45.

[28] Amir Michail. 2015. https://www.reddit.com/r/compsci/comments/3bvw1t/have_there_been_any_breakthroughs_in_turing/.

[29] Mauro Mosconi and Marco Porta. 2000. Iteration constructs in data-flow visual programming languages. *Computer Languages* 26, 2-4 (2000), 67–104.

[30] Brad A Myers. 1992. Demonstrational interfaces: A step beyond direct manipulation. *Computer* 25, 8 (1992), 61–73.

[31] Brad A Myers, Jade Goldstein, and Matthew A Goldberg. 1994. Creating charts by demonstration. In *ACM Conf. Human Factors in Computing Systems (CHI)*. 106–111. https://youtu.be/TDK5PI6WJtw.

[32] Mark W Newman, James Lin, Jason I Hong, and James A Landay. 2003. DENIM: An informal web site design tool inspired by observations of practice. *Human-Computer Interaction* 18, 3 (2003), 259–324.

[33] E. Pasternak, R. Fenichel, and A. N. Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*. 21–24. https://doi.org/10.1109/BLOCKS.2017.8120404

[34] Miller S Puckette. 1996. Pure Data: another integrated computer music environment. *Proc. Second Intercollege Computer Music Concerts* (1996), 37–41. https://puredata.info/.

[35] Bahador Saket, Hannah Kim, Eli T Brown, and Alex Endert. 2016. Visualization by demonstration: An interaction paradigm for visual data exploration. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* 23, 1 (2016), 331–340.

[36] Robert Schaefer. 2011. On the limits of visual programming languages. *ACM SIGSOFT Software Engineering Notes* 36, 2 (2011), 7–8.

[37] Ben Shneiderman. 1982. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology* 1 (1982), 237–256.

[38] Ben Shneiderman. 1983. Direct manipulation: a step beyond programming languages. *IEEE Computer* 16, 8 (August 1983), 57–69.

[39] Bret Victor. 2013. Drawing Dynamic Visualizations. https://vimeo.com/66085662, http://worrydream.com/DrawingDynamicVisualizationsTalkAddendum/.

[40] David Wolber. 2011. App Inventor and Real-World Motivation. In *Proc. ACM Technical Symposium on Computer Science Education*. 601–606. https://appinventor.mit.edu/.

[41] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. 2018. DataInk: Direct and creative data-oriented drawing. In *ACM Conf. Human Factors in Computing Systems (CHI)*. 1–13. https://youtu.be/xlVZKGClcC0.

[42] Jun Xing, Rubaiat Habib Kazi, Tovi Grossman, Li-Yi Wei, Jos Stam, and George Fitzmaurice. 2016. Energy-brushes: Interactive tools for illustrating stylized elemental dynamics. In *ACM Symp. User Interface Software and Technology (UIST)*. 755–766. https://youtu.be/qj2XxB2dsco.