OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Master Thesis

Direct Manipulation of Turtle Graphics

Matthias Graf

September 30, 2014

Supervisors:

Dr. Veit Köppen & Prof. Dr. Gunter Saake
Otto-von-Guericke University Magdeburg

Prof. Dr. Marian Dörk
University of Applied Sciences Potsdam

Abstract

This thesis is centred around the question of how dynamic pictures can be created and manipulated directly, analogous to drawing images, in an attempt to overcome traditional abstract textual program representations and interfaces (coding). To explore new ideas, Vogo[1] is presented, an experimental, spatially-oriented, direct manipulation, live programming environment for Logo Turtle Graphics. It allows complex abstract shapes to be created entirely on a canvas. The interplay of several interface design principles is demonstrated to encourage exploration, curiosity and serendipitous discoveries. By reaching out to new programmers, this thesis seeks to question established programming paradigms and expand the view of what programming is.

[1]http://mgrf.de/vogo/

Contents

1 Introduction **5**

1.1 Research Question . . . 6
1.2 Turtle Graphics . . . 6
1.3 Direct Manipulation . . . 8
1.4 Goal . . . 10
1.5 Challenges . . . 12
1.6 Outline . . . 14

2 Related Research **15**

2.1 Sketchpad . . . 15
2.2 Constructivism . . . 16
2.3 Logo . . . 19
2.4 Scratch & Snap . . . 22
2.5 Recursive Drawing . . . 23
2.6 Drawing Dynamic Visualisations . . . 25
2.7 Summary . . . 28

3 Vogo **29**

3.1 Interface Overview . . . 30
3.2 Design Principles . . . 31
3.3 Proxies . . . 32
3.4 Move . . . 33
3.5 Rotate . . . 34
3.6 Loop . . . 36
3.7 Branch . . . 37
3.8 Selection & Editing . . . 40
3.9 Expressions & Dragging . . . 43
3.10 Subpictures . . . 45
3.11 Call . . . 46
3.12 Recursion . . . 47
3.13 Scope . . . 49
3.14 Parametrisation . . . 50
3.15 Flow . . . 50
3.16 Export . . . 52
3.17 Compiler Design . . . 53
3.18 Summary . . . 55

4 Discussion **56**
4.1 Comparison with JSLogo . . . 56
4.2 Exploration . . . 61
4.3 Programming or Drawing? . . . 63
4.4 Environment Analysis . . . 63
4.5 Limitations . . . 65
4.6 Open Questions . . . 66

5 Conclusion **68**

Bibliography **70**

1 Introduction

Bret Victor inspired this thesis project. His guiding principle is that **creators need an immediate connection** to what they are creating (Victor, 2012b). For example, drawing on paper is a very direct creative process. Each stroke is visible immediately and adds to a continuously forming picture. The pen or brush becomes an extension of the artists body to express intent. This allows for a tight feedback loop to establish between the formation of ideas and their externalisation, each driving the other, which is crucial for the creative process, according to constructionist theories of learning (Papert and Harel, 1991) (see section 2.2).

The emergence of the computer paved the way for new forms of creation, many of which are yet to be invented, since computing is still in its infancy (Kay, 2007) (Sussman, 2011). In contrast to drawing, which creates static pictures, and animation, which creates static moving pictures, **dynamic pictures** can behave, be sensitive to context, change based on data, or respond to user input (Victor, 2011a). Victor points out that dynamic pictures are native art in the medium of the computer, because it provides the fundamental ability to simulate, not available in any other medium (Victor, 2013c). An example are computer games, perhaps the most sophisticated dynamic pictures today. But dynamic pictures do not have to be interactive. Data-driven graphics, like histograms or treemaps, are not necessarily interactive, yet dynamic, because these information graphics change depending on input data. Explorable explanations are another important category of dynamic pictures (Victor, 2011c). A simple example is Jason Davies' succinct illustration of how parametric Bézier curves are constructed (Davies, 2012).

Although information graphics and visual explanations are scientifically recognised to facilitate understanding (Card et al., 1999), the tools for their creation are hard to use and learn. Some widely used today include D3[1], Processing[2], R[3] and VTK[4]. All involve the **manipulation of abstract textual representations - coding**, a highly indirect form of creation, requiring a great deal of technical sophistication. An author has to maintain an intricate mental mapping between text and picture. How can this process be simplified? How can dynamic pictures be created by manipulating the picture itself, instead of text? Those questions motivate this thesis, but are to broad too pose as research questions.

[1]Data-driven documents (Bostock et al., 2011) - http://d3js.org/
[2]http://processing.org/
[3]The R Project for Statistical Computing - http://www.r-project.org/
[4]The Visualization Toolkit - http://www.vtk.org/

To make programming approachable by children, Seymour Papert et al. developed the programming language **Logo** (Papert, 1980). It creates **Turtle Graphics**, a form of dynamic picture, that is based around the idea of moving a pen on a canvas, creating a line path based on procedures. Logo's design principles, like its inherently visual and body-syntonic nature, allow children to relate to programming and make learning a motivating, fun and insightful experience that can grow powerful ideas. Papert's insights are the second key motivation for this thesis project.

1.1 Research Question

Taking the insights carried by Logo as a starting point, the central research question is: **How can Turtle Graphics be created through direct manipulation?** An alternative formulation is: How can abstract procedural paths be created and manipulated directly, without the "detour" through textual representations of program structure, but rather be "drawn", that is, be visualised and constructed in a spatial context? Those questions itself raise many questions, most significantly:

- What are Turtle Graphics and abstract procedural paths?
- What are the insights of Logo and what makes it a good starting point?
- What is direct manipulation and why is it important?
- What is meant by "detour", "drawing" and "spatial context"?
- How can a desirable solution be characterised?
- What challenges does the question pose?

The following sections are devoted to explaining and answering them.

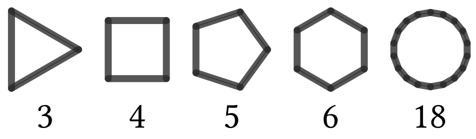
1.2 Turtle Graphics

In Logo a Turtle Graphic is specified by a procedural program that instructs a turtle to draw a path, hence the name. The turtle can also be thought of as a pen, but a turtle is a more anthropomorphic metaphor. The pen has a position and heading on a two-dimensional plane. The most important instructions are *move* forward or backward and *rotate* right or left. To draw an **equilateral triangle** with a side length of 100, the pen would have to do:

```
forward 100
 right  120
forward 100
 right  120
forward 100
```

Each instruction changes the pen's state. The three *forward* commands draw the equilateral sides of length 100, a dimensionless unit. The *right* rotate the pen by a given number of degrees, 180°-60° for the outside angles. Each move of the pen appends a line segment to the path. Note that due to the missing third rotate, then pen is not returned to its original heading. The program can contain iterations and parameterised functions, which allows for the specification of a whole *class of paths*. This is why Turtle Graphics, in their most basic form, are *abstract* procedural paths. A simple example is any **convex regular polygon** (an equiangular, equilateral polygon), written in Logo as:

```
to regularPolygon :numberOfEdges
  repeat :numberOfEdges [
    right 360/:numberOfEdges
    forward 300/:numberOfEdges
  ]
end
```



This function can be called with any number of edges to produce an image. If the number of edges is set to three, the above mentioned equilateral triangle is drawn. The regular polygon is thus a possible abstraction for a equilateral triangle. The higher the number of edges, the more it approximates a circle. A regular polygon is a dynamic image because it changes based on input, the number of edges. It can be thought of as a *parameterised image*.

Logo's syntax and semantics are comparatively **easy to understand**. For example, instead of the common *for-loop*, the simpler *repeat x-times* is available. State is minimised due to its functional style and the state that does remain is trivially visible as the pens position and heading. Furthermore, most commands have a visible effect or pendant in the image. This means that the output (image) still contains much information about the execution flow of the program. This is crucially important for understanding how the program works. But even more important is the use of powerful metaphors. Drawing and navigating in space are two activities everyone is familiar with. Logo enables programmers to reason not only symbolically about their programs but also visually/spatially and even kinaesthetically, because they can project themselves into the turtle. This is called body-syntonic thinking (Papert, 1980, p. 63). Logo is explained in more detail in section 2.3.

In the light of the overarching motivation, how to enable the direct manipulation of dynamic pictures, Logo is **suitable as a research subject** because it 1) can produce a form of dynamic pictures, namely Turtle Graphics, 2) is easy to learn and use, 3) has a strong correspondence between its language constructs (represented symbolically via text) and visuals and 4) is still characterised by indirect manipulation methods, exemplary of most common programming interfaces.

1.3 Direct Manipulation

Programming has a long history. The paradigms and mindsets programming today is based on developed largely as a result of the technological evolution of computers. In his paper *Alternative Programming Interfaces for Alternative Programmers* Toby Schachman conducts a thorough investigation of physical, conceptual and social programming interfaces (Schachman, 2012). One example characteristic of physical interfaces is the **dominance of text manipulation in programming**. Text is a linear medium, well suited for paper formats, punchcards, teletypes, magnetic tapes, keyboards and the like. Although most of these interfaces have become obsolete, traditional ways of programming are still emulated on newer devices, that do not share the same restrictions on format. Video displays allow the manipulation of information in a spatial, non-linear fashion. Yet command line interfaces and consoles are still in common use today. Another important reason for the prevalence of text is the fact that symbolic representations can be made unambiguous and compact through carefully designed formal languages, syntax and semantics. Communicating with the computer in such a language loosely mimics human dialogue in written or verbal form. But **symbolic as opposed to iconic representations are inherently indirect** signifiers, because they are arbitrary and do not resemble the signified (Fiske, 1990, pp. 46-56).

Another implication of textual communication is a **turn-based workflow**. I speak, you speak, I speak, you speak. This is a channel limitation. Humans can not speak and listen at the same time, nor write and read simultaneously. Compare this to the experience of adjusting an audio volume turning knob. The control is continuous, not discrete, the reaction speed instant, not delayed, the communication simultaneously (full-duplex), not turn-based: one can hear the volume change and manipulate it at the same time. Due to those attributes, an *isomorphic relationship* between interface (knob) and object of interest (volume) can develop in the human mind, which is essential for an effortless mastery. For a more detailed study of this subject matter I refer to *The Design of Everyday Things* (Norman, 1988). In contrast, a typical programming workflow (including Logo's) involves **recompile and run cycles**. Between a change in the code and the observable effect of this change a considerable amount of time may pass. This essentially forces a programmer to mentally simulate in advance what the computer would do, because he can not actually see an immediately response. This has multiple negative consequences. For example, **debugging** is impeded, because an error is not revealed in sync with the step that lead to its introduction. This not only delays a correction, which itself makes learning harder, but also complicates the bug-backtracing, because the search room grew bigger. Another negative consequence is that it hampers a quick **exploration** of multiple possible choices, because the up-front investment is increased (in terms of time and cognitive precomputation required). How can we instead design a programming environment that encourages exploration and playful interaction? Much of discovery in the history of mankind would not have been possible without curiosity, open-mindedness and serendipity. Instead of forcing programmers to think like a computer, how can we make programming understandable by people (Victor, 2012a)?

Schachman also mentions conceptual programming interfaces, the **metaphors** programmers typically use to think about their programs, like objects, inference, arrays, streams and transitivity. They tend to be *technical or algebraic* in nature. This may be unsuitable in certain domains. If an artist wants to create a dynamic picture, *spatial or geometric* metaphors are more appropriate, like orientation, velocity, perspective and intersection. Again, the use of metaphors in programming is influenced by the predominance of text manipulation. Coding builds on our language capabilities - we think linguistically. A shift in metaphors is likely to co-occur with a shift to visually oriented manipulation, which, in the case of dynamic pictures and Logo in particular, is also a more direct interface, because the objects of interest and the means of manipulating them lie in the same domain. This is precisely what is meant with avoiding a "detour through textual representations" in the research question.

Lastly, social interfaces describe what programming is, **who programs and how it relates to society**. Schachman argues that

> Many profound advances in programming were the result of people reconsidering the question, *who are the programmers?* (Schachman, 2012, p. 3)

He mentions Engelbart's oN-Line System (NLS) as such a revolutionary invention, making him one of the prime conceivers of personal computing and networked collaborative work (Engelbart, 1962). Sutherlands Sketchpad (Sutherland, 1964) had a similar transformative impact. In the early days of computing, programming was considered to be equivalent to calculating. Engelbart and Sutherland, among many others, helped redefine what programming is and who programs. In addition and stark contrast to previous computer interfaces, their programs featured what was 20 years later to be coined **direct manipulation** by Shneiderman:

> The central ideas seemed to be visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the objects of interest [...] (Shneiderman, 1983, p. 57)

Direct manipulation revolutionised the human-computer interface (Hutchins et al., 1985). This also applies to programming programs, which since then evolved into integrated development environments. For example, text manipulation has turned away from mode-based to mode-less editing and writers can use their mouse to point to and select text (Tesler, 2012). But it is important to think about what the **object of interest** is. In the case of programming, it is certainly useful to be able to directly manipulate text, but text itself is not the object of interest, not even the program itself, but rather **whatever people want to use the program for**. An ideal user interface for programming would therefore allow the programmer to manipulate directly whatever is of interest to those who use the program. Shneiderman summarises this *user-centered design* approach by saying:

> The old computing is about what computers can do; the new computing is about what people can do. (Shneiderman, 2002, intro)

1.4 Goal

The goal is to **find means for directly manipulating Turtle Graphics**, proof the concept with the implementation of a prototype and analyse its design ideas in order to further the understanding of how to develop new spatially-oriented, direct manipulation, live programming environments. The **prototype** is to be based on Logo and implement its essential ideas. It has to be possible to create abstract procedural paths in a visual manner, for example: regular polygons, stars, spirals, trees, fractals, histograms, pie charts. All important programming tasks (add, select, delete, copy, paste, adjust, rearrange, insert, abstract, iterate, recurse, branch, call, ...) must *not* rely on code/text manipulation, but be applicable to a graphical program representation, that is iconic of the object of interest: the dynamic picture that is being created. The feedback is immediate and continuous. There are no explicit recompile and run cycles that delay execution. This is related to *live programming* (Tanimoto, 2013). Each step taken in the construction of the Turtle Graphic ideally has to have a visual corollary reflecting its effect on the graphic immediately.
In short, creators need to see what they are doing. A painter wearing blindfolds has to imagine and remember everything he is doing. This is of course unacceptable. Yet as an analogy to programming it illustrates the main concern of this thesis. *Mental focus is sharply restricted* in terms of objects it can hold. Creators need tools that help them externalise their ideas as soon as possible, get them out of their heads, so they can refer to them, reflect on them, refocus without losing track, start to grow and concretise them.

I approach the research question from a **user-centered design perspective**. The goal is for the prototype to be easy to use and understand without loosing versatility and abstraction capabilities. This is a trade-off to the extend that abstraction is inherently hard to grasp and visualise. Logo is a strong basis in that regard because it already targets children and encourages powerful ways of thinking. This should however not be confused with meaning that Logo is unsuitable for "real" programmers or older generations. Papert argues that what helps children learn is not outdated by age (Papert, 1980, pp. 7-11). Furthermore, certain Logo implementations are being used all the way up to university computer science courses, the most prominent example being Brian Harvey (Harvey, 1997b) (Harvey et al., 2014). The prototype presented in this thesis explicitly does not only target today's programmers, nor does it specifically target children. It is however a declared goal to expand the view of who can and should program, whether it is children who want to learn programming, information designers who want to create a data graphic or artists how want to create generative art. The motivation is to **make programming accessible to a wider public** and to democratise visualisation (Viegas et al., 2007). After all, how good is a user-centered design in the domain of programming when most people can not at all relate to programming? Mitchel Resnick, head of the MIT[5] Lifelong Kindergarten group, emphasises this point by arguing that programming is a form of literacy and without programming people can not get fluent in the medium of the computer (Resnick et al., 2009).

[5]Massachusetts Institute of Technology

In order to clarify, I will now point out what is **not within the scope** of this research project. It is *not* the goal to extend, change or reimplement Logo. The prototype builds on the essential ideas behind Turtle Graphics, but I make no claim to be complete in implementing Logo. The command *language itself is not the central concern* but rather its front-end, the programming environment. I shall strictly differentiate between language and environment throughout the thesis. For example, the language is and will remain fundamentally procedural. There is much potential in using *constraint-based* or even *goal-oriented* programming models, but this is far beyond the scope of this thesis. In addition, it is not *programming by example*, because it does not infer or generalise automatically. However, it is *programming with example*, according to Myers taxonomy (Myers, 1986). That means that the programmer is required to provide default values for abstractions in order to work out a program on a concrete example.

Another aspect are the terms *drawing*, *image* and *graphic*, which are used throughout the thesis. They are not meant to imply that the prototype is a drawing tool or that it merely creates conventional images. **Drawing is meant as an analogy** for the process of the program construction, which is to be similar in fashion. But it is not identical and even radically different in certain characteristics because the prototype is not creating an image but a program that can generate a *class of images*. The pictures are not one-offs. For example, they may be data-dependent, which makes them data graphics, or simply customisable via parameters that influence their appearance. So it is not a drawing tool but it is also *not a coding tool*, because it does not emphasise text manipulation.

The last distinction important to draw is that this tool has almost **nothing to do** with what is called **visual programming**. Visual programming has a long history (Myers, 1986) (Sorva et al., 2013). It is generally used to describe means of visualising the program structure with blocks or patches that can be dragged, connected and attached to one another. Scratch is a modern example of this approach (Resnick et al., 2009). It differs substantially from what this project is trying to accomplish in multiple ways: **1)** the goal is not to find graphical representations for the static program structure, but for its "run-time" behaviour, which is not even visible in the code. "Run-time" is quoted because the term is derived from the assumption that there is a distinction between *compile-time* and *run-time*, which the prototype is trying to break. **2)** The spatial layout in visual programming has nothing to do with the object of interest. It is often simply used as a means for organising code elements, much like indentation is used in text, but has no syntactic meaning. Not true here. If a command moves the turtle forward, the new spatial representation of this command is to reflect both its effect and syntactic meaning. **3)** In visual programming, the graphics are separated from the program output. In contrast, the goal of the prototype is to integrate program construction with the result (direct manipulation). **4)** The visuals are not just another symbolic representation but iconic of the run-time behaviour or the object of interest. To my knowledge, nothing that fulfils these requirements exists in the domain of visual programming.

Critically important for defining the requirements of a good solution to the research question are Victors thoughts on how to *design a programming environment for* *understanding programs* (Victor, 2012a). The primary objective is to enable programmers to see and understand the execution of their programs. The key **environment requirements** are:

- The vocabulary is readable; meaning is transparent.
- Program flow is visible and tangible.
- State is either eliminated or visible. There is no hidden state.
- Allows creating by reacting: the toolbox is shown and results are instantly visible.
- Encourages starting concrete, then abstract - see also (Victor, 2011b).

The key **language requirements** are:

- Identity and metaphor - programmers can relate the computer's world to their own.
- Ease of decomposing problems into mind-sized chunks.
- Ease of recomposing solutions.

An evaluation of those is provided in the discussion, section 4.4.

1.5 Challenges

The hardest challenge is the design of a command interface for the **unambiguous creation and manipulation of abstractions**. This requires carefully constraining the pathways for their formulation to eliminate ambiguity but retain flexibility and expressive power. Formal languages are designed to be unambiguous. This is comparatively easy because the meaning of symbols can be defined rather freely. For example, if a formal language designer wants to introduce iteration, he just defines the for-loop construct to be initialised by the reserved keyword *for* followed by brackets (containing 3 special-purpose compartments separated by another ; reserved ; symbol). This construct has no precedent, it is completely arbitrary. People trying to learn programming therefore naturally have no clue what it means. They can not relate it to anything that is already familiar to them. Worse yet, symbols with an established meaning may be redefined in order to suit the need of the language. For example, when trying to make sense of an expression like *c = b % a++* newcomers first have to unlearn what they thought *equals*, *percent* and *addition* means.

Instead it is desirable to **find program representations and interaction methods with well-known connotations** that are in line with their usage/meaning in the programming environment. For example, dragging elements on the screen is an interaction method that is metaphoric of grabbing and moving physical objects in space. If drag was to be used in a programming environment to manipulate an object, the natural and justified assumption is that it somehow changes the object's spatial properties. This is used in all drawing tools (for moving, panning, scaling, etc.) and has been shown to work well. Finding such a good mapping is hard, because in order to draw on analogies, the freedom to define meaning is restricted. If the drag was used to clone the object, the *moving* analogy is useless, even a hurdle, because cloning as a result of dragging is unexpected. This is complicated by the fact that **abstraction is inherently hard** and **well-known analogies for it are scarce**.

Ambiguity is present when the programmer's action can be *interpreted or realised* by the environment in multiple ways or vice versa: the environment's feedback is not clear. There is a subtle but important difference here between interpreted and realised: *interpreted* implies that the programmer's intent can not be inferred by the environment from his actions, while *realised* implies that the intention is clear, but it can be realised in multiple ways. A hypothetical example would be the programmer's wish to *create a line!* If he can not unambiguously communicate that wish it is an *interpretation* problem. But if he could, there still remains the *realisation* problem, because multiple programs satisfy this wish. The line could be long, could be short, could be horizontal, could be vertical. This example is an oversimplification, but illustrates the point. The problem of ambiguity is discussed in more detail in section 4.1.

But it is not only the careless use of symbols and inappropriate interfaces that make programming an obscure art, it also has to do with the fact that most programs resemble **black boxes**, even when the source code is available. In order to understand programs it is essential to see inside, see the run-time behaviour. Programmers may ask themselves: *How often was this loop run?* or *Was this condition met?* or *What is the concrete value of this variable in this situation?* In order to answer those questions programmers often resort to printing to the console or setting breakpoints, which is comparable to figuring out the travel route of a salesman by placing a policemen on every junction in the city to report back. This is not good enough. How can program behaviour be made transparent? The particular challenge of this thesis project is to answer this question for the case of Turtle Graphics. How can these dynamic pictures be turned into white boxes?

All of those mentioned are design challenges, but there are also considerable technical challenges. Writing a **live compiler** is the hardest one. The term in fact is an oxymoron. Compiling implies that causality flows originating from the *source* code. But *liveness* requires backwards reasoning: how does a change in the program propagate back to the source? The programmer needs to see and manipulate the program while it is running. Compiling therefore turns from being one step in the chain to a permanently running stateful process, more like an operating system for programming.

Stepping back a bit from the immediate challenges that the research question poses, it is important to keep in mind that the biggest challenge is to **find the right questions to ask** in the first place. Once a better understanding of the core issues is unravelled, the questions again need to be refined or even redefined. Many paradigms and assumptions form our understanding of what programming is and what it is not. Challenging them requires realising their existence first. Marshall McLuhan once humorously noted that fish were probably the last to invent water (McLuhan, 1969, p. 5).

1.6 Outline

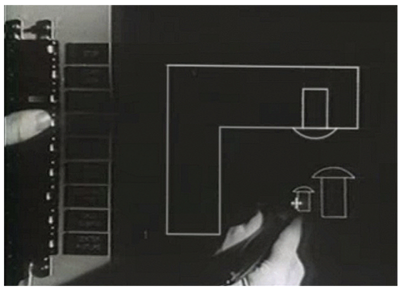
This chapter motivated and explained the research question. I introduced the term dynamic picture and one particular subclass: Turtle Graphics. Direct manipulation was discussed and confronted with today's programming paradigms, followed by a thorough description of the research goals and challenges.

The next chapter will present related research and explain how it contributes to finding an answer to the research question. Afterwards, Vogo is presented, an experimental, spatially-oriented, direct manipulation, live programming environment for Logo Turtle Graphics, which serves as a case study and proof of concept. It follows a discussion of the findings, ideas for future work and the closing conclusion.

2 Related Research

Conceptually, the closest related research projects are Recursive Drawing (Schachman, 2012) and Drawing Dynamic Visualisations (Victor, 2013a). Logo sparked many interesting languages (Boytchev, 2014), notably Squeak/E-Toys (Ingalls et al., 1997), Scratch (Resnick et al., 2009) and most recently Snap (Harvey et al., 2014). Starting with Ivan Sutherland's revolutionary Sketchpad from 1964 the related work is presented in chronological order.

2.1 Sketchpad



Sketchpad[1] pioneered computer graphics which earned Sutherland the Turing Award in 1988 (Sutherland, 1964). It allowed direct manipulation drawing on a computer screen with a light pen. Several modes of operation could be picked with the left hand. Among the many impressive features was a constraint solver that could be operated by **directly applying a set of rules** to screen elements. For example, a set of lines could be set to be mutually perpendicular. The system automatically figured out a solution through numeric approximation. Those **rules were remembered and dynamically maintained** as the drawing changed in subsequent steps. This was the first important step in moving from static to dynamic pictures. The second was **instantiation of subpictures**. This is identical to *cloning groups* in a modern SVG[2] drawing program like Inkscape[3]. A selection of elements could be used as a blueprint for copies. The children of those subpictures could be *repositioned*, *rotated* and *scaled*, but their appearance was bound to the parent. If the parent was manipulated, all children inherited the change. This made Sketchpad a simple object-oriented system with polymorphism.

Vector graphics contain information about how a picture is to be drawn with geometric primitives, like circle, rectangle, line, path, arc, font. This information is used to aid the creative process. For example, elements that are overlapped by others can be brought to the

[1]The picture is taken from the Sketchpad demonstration video.
[2]Scalable Vector Graphics - http://www.w3.org/Graphics/SVG/
[3]http://www.inkscape.org/

foreground. This is not possible in raster graphics or paper drawing. Once applied, actions are irrevocable. **Every creative tool imposes constraints that tend to encourage certain types of working.** Sketchpad's design makes it suitable for technical drawings, while painting a person's face would turn out to be a chore. Likewise, Turtle Graphics are constructed in a way that makes it easy to draw fractals. This is important to keep in mind during the design of dynamic drawing tools.

Sketchpad's **subpictures** all expose the same trivial spatial parameters: *position*, *orientation* and *size*. None can be added, removed or otherwise restrained. For example, a star can not be drawn and then instantiated with a different number of spikes. Also, the graphic can not be made data-dependent. Turtle graphics are far superior in that regard, because the rules that describe the graphic are much more expressive. Although there has been some research in the field of constraint-based drawing, for example, *Briar* improves on Sketchpad's creation, display and editing of constraints (Gleicher and Witkin, 1994), the same essential limitations apply.

Still, subpictures are an important ingredient for the creation of dynamic pictures. They **enable encapsulation** and easy reuse. This is important for problem solving because partial solutions can be packaged and later recombined to solve harder problems. It relates to two language requirements that I brought forward: ease of decomposition of problems and recomposition of (partial) solutions. Sketchpad also serves as a model for the physical interface the prototype will use: a pointing device (the mouse) and buttons (the keyboard).

2.2 Constructivism

This research project is heavily influenced by **Seymour Papert's** work in the fields of developmental psychology, epistemology and computer science. One of his main interests was understanding how humans learn. In his seminal book *Mindstorms: Children, Computers, and Powerful Ideas* Papert proposed radical reforms of education in particular and the learning culture in general (Papert, 1980). Among his proposals was the introduction of Logo, an educational programming language that he developed in 1967, into school education. The prototype developed as part of this thesis is based on Logo. This is why it is in turn important to understand the ideas behind Logo - they equally apply here. I will first take a glance at the epistemological theory of *constructivism* and the related *constructionism.* This is an attempt to **ground the quest for direct manipulation to theories of learning**. I will then take a closer look at Logo and its descendants to analyse to what extent the modalities of their programming environments help to answer the research question.

Papert worked together with the Swiss developmental psychologist **Jean Piaget** from 1958 to 1963. Piaget was the pioneer of the **constructivist theory of knowledge** which argues that knowledge and meaning is constructed in the human mind from an interplay between experiences and ideas (Piaget, 1955, pp. 350–363). This process results in the construction of an individual representation of the world. Learners can be thought of as active "builders of their own intellectual structures" (Papert, 1980, p. 7). Categories and models of sense-making are formed by *assimilation* and *accommodation* of experiences. Assimilation is the process of incorporating experiences into existing mental schemata. Accommodation is the process of adapting mental schemata to fit new experiences. Piaget writes:

> To understand is to discover, or reconstruct by rediscovery, and such conditions must be complied with if in the future individuals are to be formed who are capable of production and creativity and not simply repetition (Piaget, 1973, p. 20).

The constructivist sense-making process can be **compared to scientific experimentation** and model building, but carried out on a personal level. A theory (the mental model) is put to the test which results in an experience: either the theory is *confirmed* or *falsified*. If it is confirmed, the theory is able to *assimilate* the results coherently. If it is falsified, the theory needs to be revised to *accommodate* for inconsistencies. For a theory of science I refer to *Logik der Forschung* (Popper, 1934). Of course, scientific experimentation is a very deliberate and stringent process, in which it radically differs from more casual and even unconcious forms of knowledge creation. But fundamentally **the test is at the heart of the matter** in both science and constructivism because it allows a comparison to be made between a model and experiences. Neither is a test useful without a theory, nor is a theory acceptable without the ability to put it to the test. Constructivism therefore argues that learning works best when the learner can create a theory, devise a test for it, carry it out, obtain results, compare it with the theory and revise it if necessary. The more rapid this cycle can be completed, the faster insights can be generated, the more satisfying is the creative process, irrespective of whether it is scientific knowledge, an engineering feat, a piece of art or fluency in a language that is the object of interest being created/pursued. This is exactly what **direct manipulation** aims for: establishing a rapid feedback loop with the object of interest, which entails immediate control over and visibility of it. But without direct manipulation, the interplay between ideas and experiences is hindered, which is what is needed for the construction of knowledge and meaning. If a scientist can not test a theory, no insights can be gathered. Likewise, an engineer can not learn anything from his construction plans if he lacks the tools to implement them. This tool in both science and engineering is increasingly the computer, which is why communicating with it in a language that is characterised by direct manipulation becomes more and more important.

How do people best learn French? By going to France! They embed themselves into an environment that provides plenty of opportunities to interact with the French language and culture. Surrounded by native speakers, which serve as a reliable and convincing role model, it is easy to engage in a dialogue, get feedback, spot mistakes fast, make corrections, adapt, retry and get a feeling for progression, by seeing what works and what does not. Provided with such an immersive experience, learning happens naturally. Children can learn multiple languages while being raised without the need for formal instruction or reflection. Two important learning mechanism are at work: *observational learning* and *trial and error.* The later obviously depends on the ability to try, get feedback and having a means of evaluating it. Its efficiency among other things depends on the amount, reliability and speed of feedback. This is one of the reasons why young children hunger for attention.

Papert takes the example of learning French as an analogy for asking for the equivalent of France for learning mathematics. What would **"Mathland"** look like? He argues that computers can be designed to allow learning to communicate with them in the same way learning French happens in France (Papert, 1980, p. 6). The second central theme is that learning to communicate with computers can change the way other learning proceeds. It can be summarised as follows:

> Papert's Principle: Some of the most crucial steps in mental growth are based not simply on acquiring new skills, but on acquiring new administrative ways to use what one already knows (Minsky, 1986, p. 102).

I described in the *Challenges* section that finding program representations and interaction methods with connotations that people can relate to is essential for the interface design. **Papert's Principle** is the more precise reason for why this is so important. People have to be able to bring their existing knowledge structures to bear in a new environment in order to make sense of it. The analogy of this to experimentation is that having test results is useless without a theory that helps to interpret them - they will just stay meaningless otherwise.

Papert was at variance with Piaget over the matter of what role the surrounding culture and learning environment plays as a source of materials for creative learning. **Constructionism** is inspired by *constructivism* but the central role is attributed to the availability of "construction materials" and explorability of ideas (Papert and Harel, 1991). For example, mathematically sophisticated adults tend to develop arguments and sound logic to underpin their convictions. Their interests may involve the solving of puzzles and riddles, the thinking about paradoxes or the questioning of assumptions in everyday live. Being in close touch with those individuals provides learners with opportunities to "speak mathematics", but not in the sense of solving formal equations, but in forming an intimate relation with a mathematical way of thinking.

Friedrich Fröbel recognised the importance of educational materials and free play when he opened the world's first **kindergarten** in 1837. Each child was to be given one of his gifts (Fröbelgaben) for self-directed activities. Those ranged from marbles to geometric wooden blocks to modelling objects of clay. The *Lifelong Kindergarten* group at MIT took up his ideas:

> Froebel was making for makers – he made objects that enabled children in his kindergarten to do their own making and creating. Froebel's work can be viewed as an early example of Seymour Papert's constructionist approach to education. Papert argued that the activity of making things provides a rich context for learning (Resnick, 2013, p. 50).

2.3 Logo

Logo is Papert's gift to the digital generation. Logo is personified by a *turtle*. Its language is *turtle talk* and its drawings *Turtle Graphics*. Children would not write a new procedure, they would *teach the turtle a new word*. The turtle is either a virtual object on the screen or a robot that actually draws on paper. The robot is more engaging, because children find it easier to relate to a physical object, especially if they can attribute it with properties of living beings, because that is their own most intimate perspective of the world (best fit for assimilation). This allows children to project themselves into the turtle and think as if they where the turtle, which means that they can bring their own body knowledge and sense of orientation to bear in the world of the turtle, a computer simulation. Papert calls this *body-syntonic* thinking (Papert, 1980, p. 63). A fun and insightful experience for children who learn *turtle talk* is to **play turtle**[4]. One child with blindfolds is pretending to be the turtle while others are giving the commands. For this to be successful, children have to start thinking about how they think, incidentally becoming epistemologists in the process. Papert writes:

> Even the simplest Turtle work can open new opportunities for sharpening one's thinking about thinking: Programming the Turtle starts by making one reflect on how one does oneself what one would like the Turtle to do. Thus teaching the Turtle to act or to "think" can lead on to reflect on one's own actions and thinking (Papert, 1980, p. 28).

For example, drawing a **circle** with the turtle can be a challenge. When children ask for help, instead of giving the solution, children would be asked to *play turtle* themselves. And since every child knows how to walk a circular shape, it is easy to come up with a solution: walk a bit and turn a bit, for a complete turn:

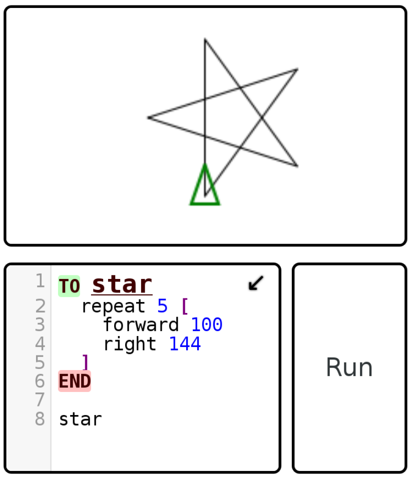
```
repeat 360 [
  forward 1
  right 1
]
```

A circle is a shape with constant curvature, which is particularly simple to express in **differential geometry**, which in turn is why this solution is so elegant. As a contrast, the symbolic representation of a circle in euclidean geometry is $x^2+y^2=r^2$, which requires much more mathematical sophistication to understand, let alone to invent (Kay, 1987, pt. 2, 21:40). This demonstrates that Logo allows the utilisation of body knowledge to help children

[4]The picture is taken from *The Children's Machine* by Seymour Papert.

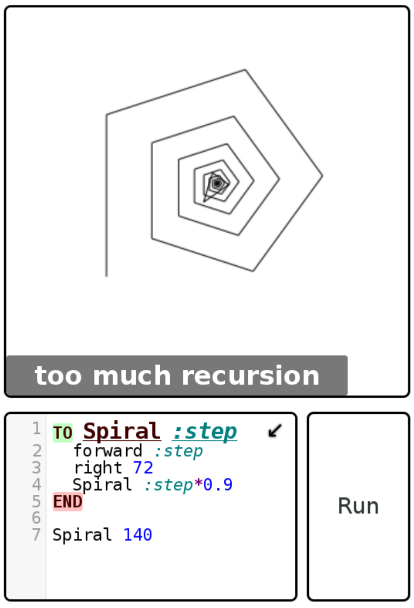
think about differential geometry and become fluent in expressing and exploring their ideas in this context. An excerpt of the things that new programmers can learn about while gradually improving their *turtle talk* is: angles, iteration, state, commands, thinking procedurally, debugging, geometric construction with differential equations, problem decomposition using subprocedures, recursion and fractals, series, state-change operators, scope, functional assignment, variables and abstraction.

Free modern implementations of Logo include **JSLogo**[5], papert[6] and UCBLogo[7]. The figure to the right shows the JSLogo programming environment. JS refers to Javascript; it runs entirely in the browser. The graphics are drawn on a HTML5 Canvas[8]. The turtle is represented by a green triangle, which always shows its current position and heading. *Run* executes the code.



```
TO star
  repeat 5 [
    forward 100
    right 144
  ]
END

star
```

The most important feature of Logo in the context of the thesis is its strong **correspondence between program run-time behaviour and visual output**. What the turtle did can literally be traced. The *Spiral* shown in the second example program on the right shall serve as a demonstration. *Spiral* is a recursive function that does not terminate. JSLogo automatically terminates the execution at a certain call depth. The initial value of the variable *step* is 140, decreasing with increasing recursion depth. Due to the fact that each *forward* and each *right* command left a visual clue in the graphic, the progression of the recursion can be seen, how rotations add up and each individual value that the variable *step* was assigned to and *when* in the program flow. Each Turtle Graphic can be interpreted as a **data graphic of the behaviour of its generating program**. This is tremendously valuable for understanding how *Spiral* and recursion in general works, because the "internal" functioning is not hidden or left obscured in loads of console text output, but beautifully encoded visually and compactly arranged spatially. It is easy do get an overview, pick out details, make comparisons and find patterns. For the importance and design of data graphics I refer to *The Visual Display of Quantitative Information* (Tufte, 1986).



```
too much recursion
```

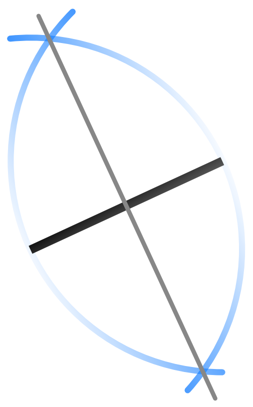
```
TO Spiral :step
  forward :step
  right 72
  Spiral :step*0.9
END

Spiral 140
```

[5]http://www.calormen.com/jslogo/, source: https://github.com/inexorabletash/jslogo
[6]http://logo.twentygototen.org/, source: https://code.google.com/p/papert/
[7]University of Berkeley Logo - often referred to as the standard Logo (Harvey, 1997a).
[8]Hypertext Markup Language - http://www.w3.org/TR/html5/

Logo is well designed for understanding programs because of this double function of its Turtle Graphics. One the one hand, to draw those images is motivating in itself and can in fact be used to generate serious data graphics. On the other hand, the output visually reflects the program itself. It is this property of Logo that lets it be an ideal research subject.

In addition, it is **in line with the research goals** in the following ways: **1)** it targets new programmers, thereby expanding the view of who programs and making programming and visualisation accessible to a wider public, **2)** it uses drawing and orientation in space as metaphors to encourage the application of existing knowledge in a new domain (see Papert's Principle), thus engaging the *kinaesthetic* and *visual* mentalities of thinking, instead of only the *symbolic* one (Kay, 1987, pt. 2, 03:51) (Piaget, 1955), **3)** program state is encapsulated in the turtle, which is always visible (Logo employs multiple programming paradigms, but can be used in a functional style), **4)** like Sketchpad, Logo allows the decomposition of problems into subpictures, **5)** the language vocabulary is readable, syntax and semantic kept comparatively simple.

However, the programming environment also has a number of severe **downsides**. I here refer to JSLogo as a representative of Logo programming environments. If the construction of a Turtle Graphic is seen as the object of interest, it is not possible to directly create and manipulate it. Instead, a textual interface introduces a layer of symbolic abstraction (code) between the programmer and the picture. The canvas in JSLogo, just like any programming console, produces output that can not be altered. Although this output in Logo can be a good representation of the essence of the program, it can not be manipulated directly. **Program flow is visible, but not tangible.** That means that in order to change something in the picture, an angle or the length of a line, the programmer first has to find the corresponding line of code that drew it, change it, rerun the program, find the changed part in the picture again and evaluate whether it had the desired effect. This process may be very complicated. Even the simple fact that programmers have to switch back and forth between two different visual contexts, namely the code and graphics viewport, is strenuous. There also is absolutely no *visual* analogy between the two viewports - text first has to be interpreted. A tiny change in one may result in a huge change in the other. But above all, the switch between picture and code requires a **switch in thinking mentality**. The first may be described as *visual, spatial, geometric* and the second as *symbolic, linguistic, algebraic*. For example, halving the length of a line can be understood in both mentalities. The figure to the right illustrates the **geometric** interpretation. It shows that the black line is halved by the grey line, which can be constructed using the two intersection points of the blue circles, which centre in the endpoints of the black line and have the same radius as the black line is long. The black line can be seen as denoting where the radii of the blue circles "meet", connecting their centre points. The **algebraic** interpretation may be *line.length*=0.5*, where *line* is an identifier, "." a property access operator, *length*



the object property, * scalar multiplication, combined with = a shorthand assignment and 0.5 a rational number. The *algebraic* interpretation represents the problem as an equation to be solved: calculating the product of two numbers and assigning it to an object. The *geometric* interpretation represents the problem as a spatial construction. Both are completely different ways of thinking about halving a line. For a more thorough investigation I refer to *A Mathematician's Lament* (Lockhart, 2009). I do *not* argue that any of the two is superior, though both have their strength and weaknesses, but rather that it is inappropriate to force a person to think about a *geometric* construction in an *algebraic* way. Halving the length of a line in a Turtle Graphic should not require the manipulation of symbolic abstractions, but be possible to perform in a spatial way.

At the very least, the **principle of locality** should be honoured (Denning, 2005). The *point of control* and the *point of effect* should be proximal in both time and space. Neither is true in Logo. Spatial proximity is foiled by two distinct viewports that separate cause and effect. Temporal proximity is foiled by the chunking to compile-time and run-time. This is not live programming. Playful interaction and exploration depend on the proximity between cause and effect.

Logo influenced the development of other languages. Among them are Smalltalk, Squeak, Scratch and Snap. A comprehensive list of dialects is provided by the Logo Tree Project (Boytchev, 2014).

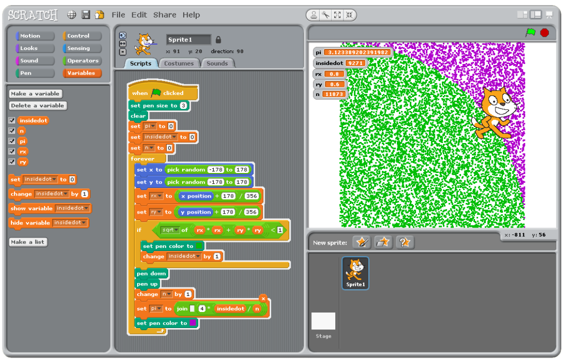
This section took a closer look at several language and environment design characteristics of Logo and JSLogo in particular. Advantages and disadvantages of Logo were discussed in light of the research question, which is aiming to find a direct manipulation interface for Turtle Graphics. Most detrimental is the required switch in thinking mentality between code and graphics view and the violation of the principle of locality. However, notwithstanding interface improvements, it remains true that:

> Logo does not in itself produce good learning any more than paint produces good art (Papert, 1980, p. XIV).

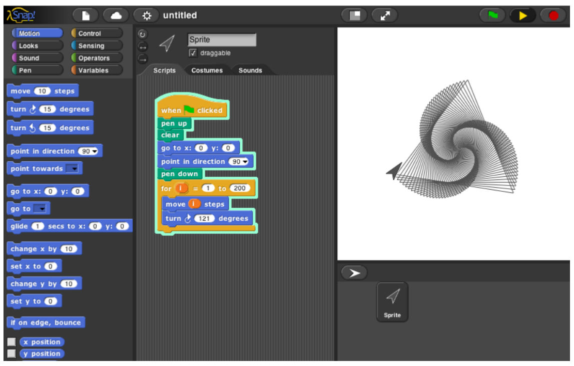
2.4 Scratch & Snap

Two popular modern languages that are inspired by Logo are Scratch[9] and Snap[10], the later being an extension of the former. Both are examples of *visual programming* environments. *Visual programming* means the use of **blocks that represent code elements**. Those blocks can be picked, dragged and snapped together in accordance with the syntactic rules of the language, in effect enforcing them, which reduces syntactic mistakes. Snap was previously called *Build Your Own Blocks* to reflect this. The interface to the program

[9]http://scratch.mit.edu/ (Resnick et al., 2009)
[10]http://snap.berkeley.edu/ (Harvey et al., 2014)



```
when clicked
set pen size to 3
clear
set pi to 0
set insidedot to 0
set n to 0
forever
  set x to pick random -178 to 178
  set y to pick random -178 to 178
  set rx to x position + 178 / 356
  set ry to y position + 178 / 356
  if sqrt of rx * rx + ry * ry < 1
    set pen color to
    change insidedot by 1
  pen down
  pen up
  change n by 1
  set pi to join 4 * insidedot / n
  set pen color to
```



```
when clicked
pen up
clear
go to x: 0 y: 0
point in direction 90
pen down
for i = 1 to 200
  move i steps
  turn 121 degrees
```

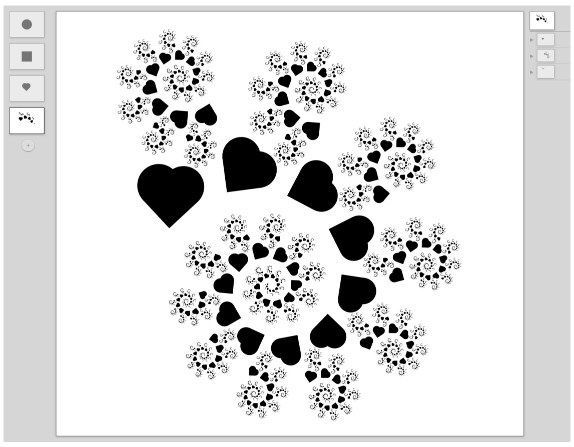
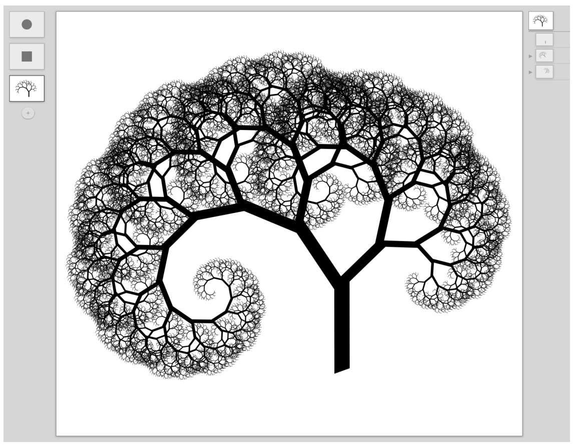
structure is presented in a graphical fashion. Most tasks can be accomplished by selection or drag and drop. Blocks may contain drop down menus for choosing features and other widgets for customisation. This significantly reduces the typing required, which in turn reduces spelling errors. Being able to manipulate the program structure without text manipulation lowers the entry barrier to programming to those not skilled in using the keyboard for writing text. ScratchJr is even more radical - it completely abdicates keyboard control in favour of touch control (Flannery et al., 2013). Another advantage is the display of a toolbox of available commands. Instead of being required to read the language documentation or refer to example programs to find valid program constructs, the toolbox is providing them right next to the program structure.

In spite of the fact that those features ease programming, the **drawbacks of Logo** relevant to the research question **remain**. The code structure can now be manipulated directly with the use of spatial metaphors, like dragging and snapping. But they *do not operate on the object of interest*. They do not break the separation between code view and output. They do not present the code structure in a novel way, which is still an indented list of commands. The graphic itself can not be edited and changing the program still relies on the recompile and run cycle. Visual programming visualises the wrong thing. The object of interest, the Turtle Graphic, remains being intangible.

2.5 Recursive Drawing

The most recent advance in the area of spatially-oriented programming environments is Recursive Drawing[11] (Schachman, 2012). Aside from *Drawing Dynamic Visualisations*, which will be discussed in the next section, Recursive Drawing is the most relevant related research project to this thesis. As the name suggests it allows the creation of recursive shapes in a way akin to drawing vector graphics. A circle and square are the basic building blocks. They can be dragged onto the canvas of a new shape, positioned, resized, rotated and sheared. Multiple existing shapes can be combined to form a new shape. If an existing shape is dragged onto itself it forms a recursive shape. The recursion is parameterised by a

[11]http://recursivedrawing.com/

planar displacement (vector addition) between the original shape and its recursive self. The displacement propagates through the recursive steps, creating new component shapes in a linear spatial progression. The endless recursion is lazily evaluated in the bounds of the screen window, minimum shape size and recursion depth ceiling.

The most interesting feature is that *any* component shape in the recursion can still be repositioned, resized, rotated and sheared *as if it were a normal shape*, without breaking the recursion, which is realised by **constraint solving**: given that the original shape is not altered, how does the recursive call have to be modified in order to comply? This is achieved by a modulation of the recursive displacement (vector component multiplication) and an additional resizing and shearing factor. This approach violates the way programmers typically think about recursion. Instead of reasoning forward from a recursive definition, the programmer can alter the properties of all elements in the recursive call stack and the system then **reasons "backwards" automatically** to derive a program that produces the desired behaviour. It breaks the assumption that the control of the program has to flow from "source" to "output". In addition, the two are not distinguishable in Recursive Drawing.

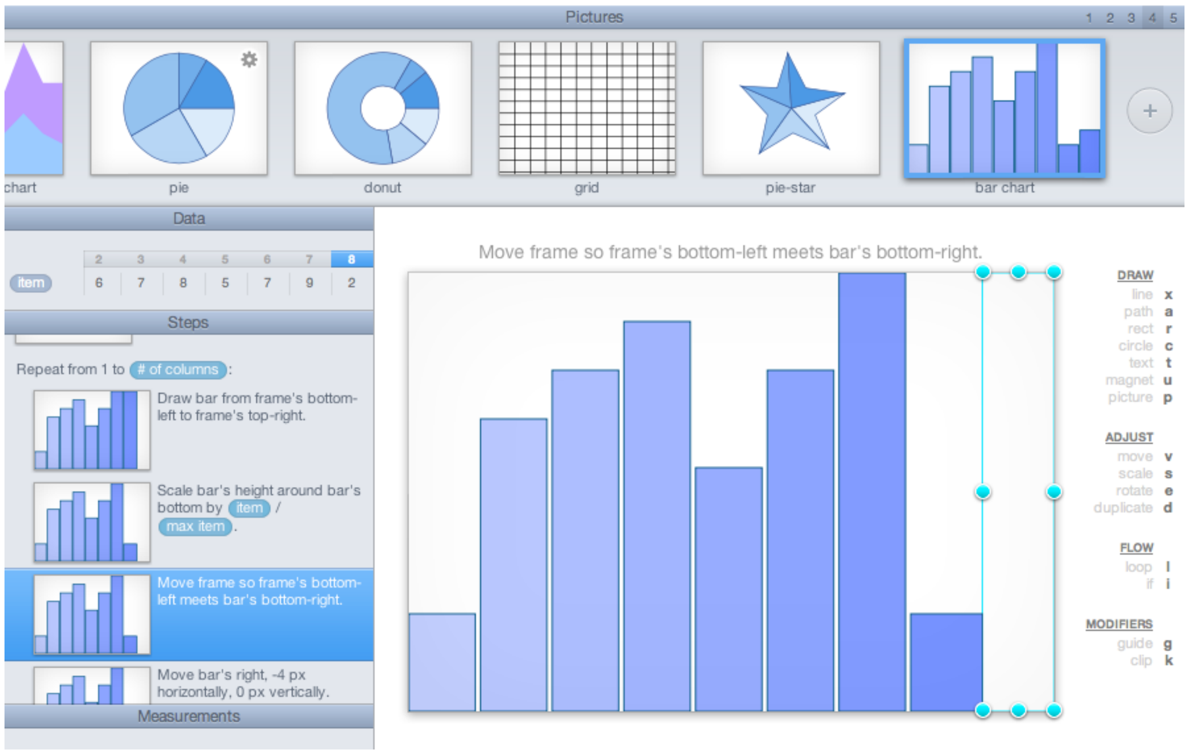
The interface is characterised by **direct spatial manipulation** of the drawing, which means that:

- all shapes are visible
- the recursion is entirely represented spatially
- shapes can be selected by pointing to them
- all properties can be altered with a spatial metaphor: drag
- the shape that is dragged accurately follows the cursor's path
- all changes are continuous
- the update to the drawing is immediate
- the attention is focused on the objects of interest

In such an environment, it is easy to develop an understanding for recursion. Exploration and playful interaction are encouraged, which allows programmers to develop an intuition for the behaviour of recursive shapes. Note that the interface manages *without symbols* *whatsoever*, which is remarkable for a programming environment. This is achieved by carefully constraining the options users are presented with, which also leads to a number of **downsides**. The most severe one is that except for recursion, nothing else can be done. In addition, the flexibility of defining the recursion is limited. For example, the rotation of the shape is coupled to the rotation of the displacement. Furthermore, the assumptions of the constraint solver can not be altered. It is always the root element that is assumed to be fixed. All of those downsides are due to a trade-off that happens between the level of abstraction perceived in the interface and the ability to meticulously control parameters.

In summary, Recursive Drawing is an excellent example of direct manipulation of recursive shapes. However, **recursive drawings are only a tiny subset of Turtle Graphics**. Expressive ability is sharply confined.

2.6 Drawing Dynamic Visualisations



	2	3	4	5	6	7	8
item	6	7	8	5	7	9	2

Repeat from 1 to # of columns:

Draw bar from frame's bottom-left to frame's top-right.

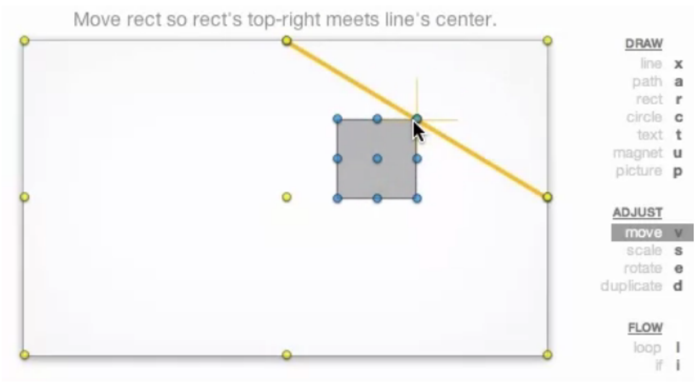
Scale bar's height around bar's bottom by item / max item.

Move frame so frame's bottom-left meets bar's bottom-right.

Move bar's right, -4 px horizontally, 0 px vertically.

Move frame so frame's bottom-left meets bar's bottom-right.

The prime reference research prototype is Drawing Dynamic Visualisations (Victor, 2013a). The tool is similar in terms of interaction style to vector graphics drawing programs, but for creating data-driven graphics, instead of static pictures. The main component is a drawing canvas. A toolbar is provided to the right, the data and program structure to the left. The program is represented as a traditional list of steps, but this viewport is not used to create commands. Instead, **steps are recorded from drawing actions** on the canvas. The steps viewport is only used to add flow control statements (if & loop) around command blocks and for navigating inside the history of the program. Selecting a step shows the picture at that time. Commands are inserted after the selected step. Every step is also enriched by a preview picture. **Each step can be parameterised** by dragging elements from the data panel onto constants.


Move rect so rect's top-right meets line's center.

The core insight is the **use of snapping to establish relationships between elements**. The above figure shows the rectangle's top-right corner snapping to the line's centre point while being dragged. Note that the key *v* is pressed to indicate the type of operation: move. This is different from drawing tools. Elements can not be dragged arbitrarily in order to avoid ambiguity. Artists have to make explicit what kind of operation (move) is performed on which handlers (top-right corner). Overlapping source and target handlers can be cycled through to pick the right one. Snap points are predefined on objects but also automatically generated at intersections. This is useful because geometric constructions often depend on intersection points. In addition, the *glomp* modifier allows snapping to any point along an object (e.g. any point on a line).

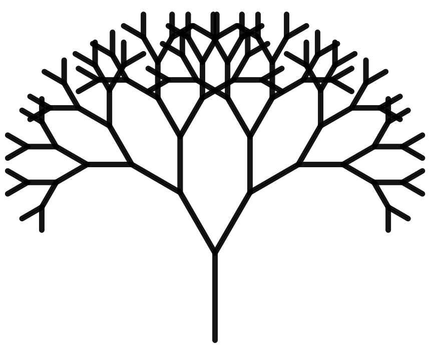
Snapping actions are shown verbalised above the canvas. Established relationships are turned into steps. The command is relative to the target. If the line's position changes, so does the rectangle's. That means that the figure is just one example illustration of the applied command. This is related to **programming by example**, but without automatic "guessing" of intent. Different preconditions lead to different visual results. However, it is possible to explore the dynamics of the program by walking through the steps or by playing with the data. Since everything is live, changes propagate immediately. Combined with parametrisation of geometric operations and flow control, snapping is what turns the static picture into a dynamic one.

The panel at the top contains the **subpictures**, the function analogons. They abstract computation. Their data panel can be thought of as their input. It is currently not possible to import geometry (other subpictures). Output is split into algebraic information (measurements) and geometric information, which consist of the graphic itself and so called *magnets*, exported snap points. Subpictures can be used in recursions.

As mentioned earlier, resolving issues around **ambiguity is challenging**. For example, trying to establish a reference to an object inside a different lexical scope is ambiguous. In case of referencing an object inside a loop, which iteration is meant? In case of a condition, what if the branch switches? In that case the reference is broken. Another example of an ambiguous act is referencing a junction in a path of line segments. Which junction is meant assuming that the path may have any number of junctions? In other words: **operations have to work in the general case**, because they are abstract, data-dependent, driven by parameters. Grounding them to concrete examples can mislead artists to perceive "simple solutions" that are not generalisable.

Drawing Dynamic Visualisations is tailored towards the construction of procedural data-graphics that rely on geometric transformations and relations between object properties. One of the major **differences to Turtle Graphics** is the coordinate system. Logo uses differential geometry. Operations *implicitly* establish a relation towards their successor through state changes. If such an influence is not intended, a subroutine has to reverse its own changes to the state before returning. Consider this binary tree as an example:

```
 1  to tree :size
 2   if :size > 5 [
 3     forward :size
 4      left 30
 5      tree :size *0.8
 6      right 60
 7      tree :size *0.8
 8      left 30
 9     back :size
10  ]
11  end
12
13  tree 20
```



Every line is actually drawn twice. This becomes immediately obvious by considering the fact that the turtle has to walk through the entire tree. Once in a leaf, it has to go back in order to reach the others. This is done by code line 8 and 9. The change to the state done in each subtree is reversed. Drawing Dynamic Visualisations instead operates in euclidean "absolute" geometry. This has the advantage that the state does not have to be reversed explicitly, if it is unwanted, but if it is, the relation instead has to be constructed explicitly. For example, drawing a circle with line segments is therefore much harder. In summary, the methods used to establish relations between objects differs substantially between Logo and Drawing Dynamic Visualisations. They are both procedural, but work with **different systems of reference**.

Drawing Dynamic Visualisations serves as a model for the research prototype. However, due to its different referencing model, not all of its insights can be directly transported to Turtle Graphics.

2.7 Summary

This chapter presented five programming environments that excel at the creation of dynamic pictures. All of them present the picture on a canvas as part of their environment, which hints at their focus. The following table briefly compares them based on the research challenges:

	Direct Manipulation	Expressiveness of Dynamics	Ease of Use	Dominant Metaphor	Presentation of Dynamic Relations
Sketchpad	Yes	Low	Easy	Technical Drawing	Spatial
JSLogo	No	High	Hard	Drawing & Orientation	Code
Scratch	No	High	Moderate	Building Blocks	Code
Recursive Drawing	Yes	Low	Easy	Drag Composition	Spatial
Drawing Dynamic Vis.	Yes	High	Moderate	Geometric Construction	Mix

Direct Manipulation is here reduced to the ability to manipulate the picture. *Ease of Use* refers to novices that are not familiar with programming. A similar set of challenges has recently been proposed in *Constructive Visualisation* (Huron et al., 2014). Coding environments generally have a high expressiveness but are hard to use. Sketchpad has a low expressiveness because of the lack of custom parametrisation and rigid constraint system. Recursive Drawing has a low expressiveness because of the same reasons and it can only create recursive shapes.

The quest for direct manipulation was grounded in constructionist learning theories, which also motivate the development of Logo. Advantages and disadvantages of Logo are explained concerning the use of metaphors, differential geometry, representation of program behaviour by Turtle Graphics, algebraic and geometric thinking mentalities and the principle of locality.

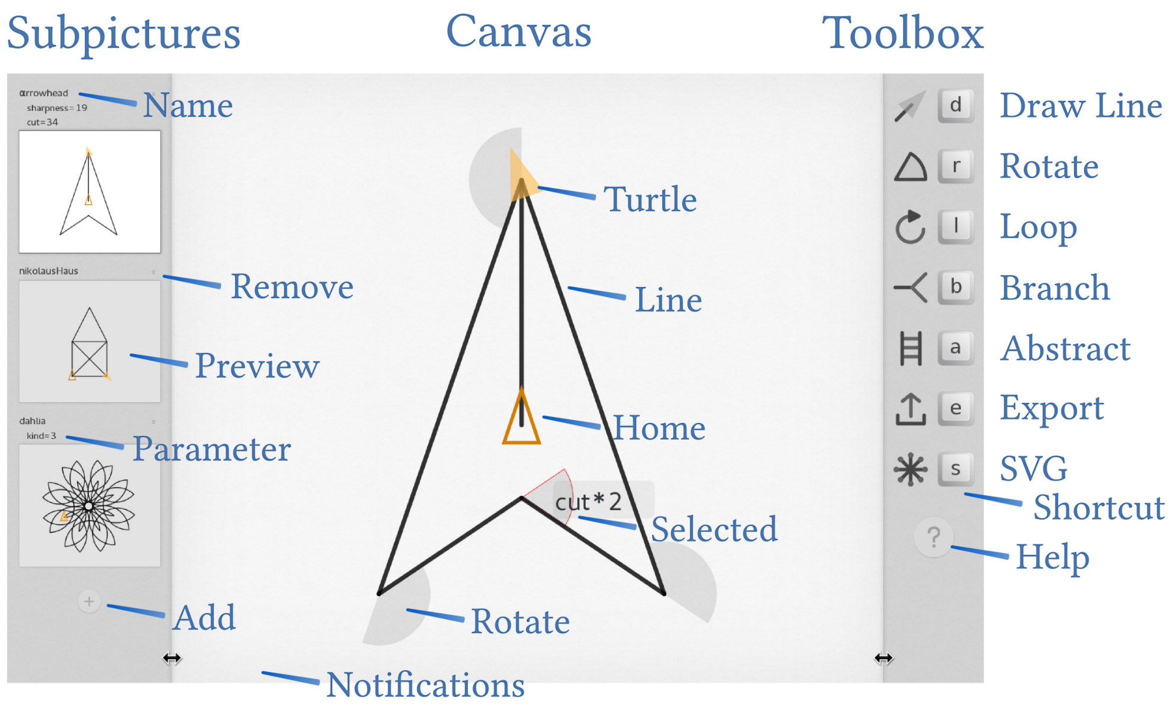
3 Vogo

The prototype is called Vogo, a portmanteau word of *Visual Logo*. Vogo is an experimental, web-based, spatially-oriented, direct manipulation, live programming environment for Turtle Graphics. Vogo serves as a demonstration of new user interface ideas. This chapter will elaborate on its usage, design and implementation.

Vogo is free software[1] and available online[2]. The source code is available on GitHub[3]. The reader is encouraged to try Vogo for himself. Many aspects of Vogo are better understood by seeing them in action then by being described. Vogo works entirely in the browser and has been verified to work reliably in Firefox 31[4] and Chromium 36[5]. It may not work in other browsers. Vogo requires state-of-the-art HTML 5[6], CSS 3[7], ECMAScript 5.1[8] and SVG 1.1[9] compliance. The next section will introduce Vogo's graphical user interface.

[1]GNU Affero General Public License 3, http://www.gnu.org/licenses/agpl-3.0.html
[2]**http://mgrf.de/vogo/**
[3]https://github.com/rbyte/Vogo
[4]https://www.mozilla.org/de/firefox/new/
[5]http://www.chromium.org/
[6]http://www.w3.org/TR/html5/
[7]http://www.w3.org/TR/CSS/
[8]http://www.ecma-international.org/ecma-262/5.1/
[9]http://www.w3.org/TR/SVG/

3.1 Interface Overview



The interface and interaction is modelled after a typical vector graphics drawing tool. Inkscape has been an inspiration. The general composition has similarities to Recursive Drawing and Drawing Dynamic Visualisations.

Vogo's central interface component (literally) is the **canvas**. It can be panned and zoomed. Compared to JSLogo and *papert*, this is a new feature. Panning works by dragging the background with the middle mouse. Left mouse drag is reserved for rectangular selection. Zooming works by scrolling with the mouse wheel. Zoom transitions smoothly towards or away from the cursor position. Zooming can therefore be used to emulate panning.

The canvas' texture and shadows are designed to **mimic a sheet of paper** on a desktop. The pen is visualised by a simple pointed orange triangle. The idea of a turtle from Logo is abandoned in favour of emphasising the metaphor of drawing.

The **home** symbol is new. It is visualised by a darker outline of the pen with transparent fill area. When a new subpicture is started, the pen "is home". Being able to spot home makes it easier to trace the path of the pen. Panning can be interpreted as "moving home".

The **toolbox** shows icons, keyboard shortcuts and provides tooltips for their meaning. Keyboard usage is encouraged for fast access but not required.

The **subpictures** panel is located on the left. Only the selected/focused subpicture is shown on the canvas. Clicking on a subpicture opens it. Each subpicture has a name, a preview and a list of parameters, which is empty by default. New subpictures can be added and removed, but one subpicture is always open. Subpictures can be renamed. Their name defaults to a Greek letter, chosen due to brevity and clarity. Greek letters are reserved for subpictures. Subpictures are analogous to functions. The figure shows 3 examples that were created with Vogo.

The interface is fully **responsive**, scales relative to the window size and without pixelation. The panels can be resized within certain boundaries. The toolbox can be hidden completely.

3.2 Design Principles

The most radical decision was to **abandon the code editor** altogether. As discussed in section 2.3, Turtle Graphics provide many clues about the program behaviour. As a consequence, reverse engineering a program from a Turtle Graphics without the source code is often easy. However, execution still reduces information. For example, *repeat 3 [forward 10]* is indistinguishable from *forward 30.* How can this loss of information be avoided by reintegrating it into the canvas? The first design principle is hence to **integrate the "unfolded" program structure with the Turtle Graphic** to break the separation between code and picture and to merge the compile-time into the run-time. This is in pursuit of the *principle of locality.*

The second consequence follows trivially: **visualise every step**. Program flow control needs to be visible and tangible inside the canvas, as well as commands and parameters. Code structures programs *horizontally* with indentation into logical blocks and *vertically* into sequential steps (execution direction top to bottom). Both scope and sequence need to be made visible spatially.

But how can **abstract functionality be visualised?** Two general observations help inform an answer. Section 2.2 mentioned how assimilation is used to categorise experiences. The human brain is innately gifted at pattern matching. It is easy to extract common patterns from examples and make generalisations based on similarities. The second observation is that everything abstract has concrete incarnations, however indirect the connection. In dialogue, when trying to explain an abstract idea to someone that is not familiar with it, people often resort to giving a concrete, illustrative example and let the other person do the inferencing. For an in depth examination I refer to *Moving Up and Down the Ladder of Abstraction* (Victor, 2011b). Based on those two observations, a **tight coupling with examples** is proposed as a design principle. Abstract functionality has to be constructed in terms of concrete examples. Default values have to be given for parameters. Drawing Dynamic Visualisations follows the same approach. It is in line with the environment requirement: *start concrete, then generalise.*

In addition, abstraction is to be made explorable along the dimensions of parameters by providing interactive control over them. This leads to the next principle: **minimise indirection** in the chain from *intention* to *control* to *effect*. This applies to all editing tasks like adding functions, renaming parameters, inserting commands, selecting steps, manipulating constants and editing expressions. Feedback must be immediate and control be interactive.

Adopting a **functional programming style** is a design guideline. State is to be minimised and variables eliminated. By default, everything is in a function. Encapsulation is encouraged. All functions are in global scope.

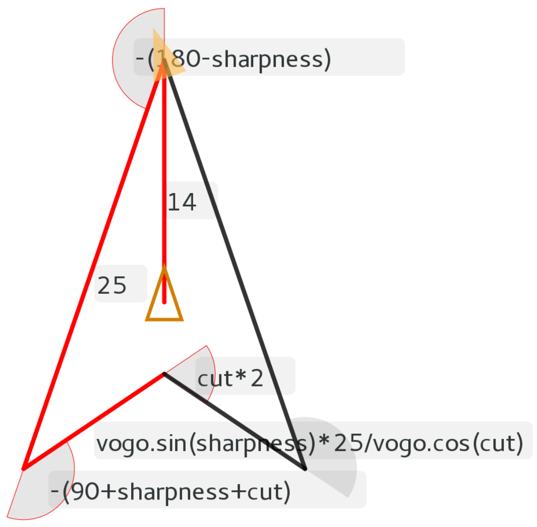
Logo has hundreds of commands. What are the essential ones? What is a set of **minimal yet flexible** commands? I conducted an informal online survey of Logo programs and found that the commands most commonly used concern: movement, orientation, repetition and calling functions. Conditions are not widely used but added in favour of flexibility. Vogo concentrates on those essentials. It is a radically simplified version of Logo.

The design principles are in summary:

- provide immediate feedback
- honour the principle of locality
- abandon textual in favour of direct manipulation
- couple program structure and behaviour
- visualise every step
- ground abstractions to concrete examples
- make state transparent - follow a functional programming style
- reduce tools to a minimal yet flexible set

The following sections will explain how they are implemented in Vogo.

3.3 Proxies



Proxies are "unfolded" commands. *repeat 3 [rotate 10]* unfolds to *repeat 3 [rotate 10 rotate 10 rotate 10]*. Programs traditionally unfold at run-time. In Vogo, **the unfolded program is the working representation**. Proxies serve a number of functions, but the most important one is to ground abstractions to concrete examples. The figure to the left shows an arrowhead, but due to parametrisation, the *sharpness* of the tip and the *cut* at its base can be controlled. Unfolding this program allows the display of one concrete incarnation with default parameters.

The code on the left is the Logo equivalent. On the right is the unfolded form.

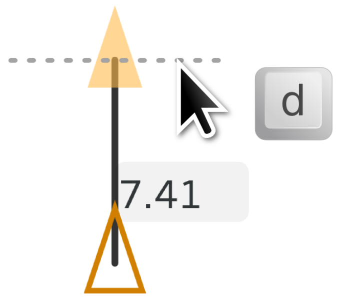
```
to  arrowhead :sharpness  :cut
  forward 14
   left  180−:sharpness
  forward 25
   left  90+:sharpness+:cut
  fd  (sin  :sharpness)*25/cos  :cut
   right  :cut*2
  fd  (sin  :sharpness)*25/cos  :cut
   left  90+:sharpness+:cut
  forward 25
end

arrowhead 19 34
```

$\rightarrow$

```

forward 14
 left  161
forward 25
 left  143
forward 10
 right  68
forward 10
 left  143
forward 25
```

Note that Vogo represents the abstract in terms of this unfolded program, but does not replace the expressions in the interface. This allows programmers to construct and understand a program with an example. The proxy system is explained in more detail in section 3.17 Compiler Design.

3.4 Move

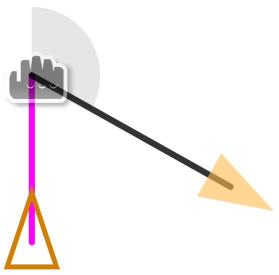


The two most basis commands Vogo supports are *Move* and *Rotate*. The default *Move* direction is forward from the current pen state. Move has one parameter: the distance travelled. When it is negative the direction is reversed. Logo instead differentiates between *forward* and *backward*, in addition to the short synonyms *fd* and *back*. Vogo reduces those four commands to one. It is the first in the toolbar and can be triggered by hitting the key *d* on the keyboard, which is the initial letter of *draw line*. Once hit, Vogo reacts by creating a **preview line** on the canvas. Its direction (forward or backward) and length can be controlled by moving the mouse on the canvas. The length is determined by dropping a perpendicular from the cursor position. The preview is finished by clicking with the left mouse on the canvas. The preview is discarded by hitting the *escape* key.

The **angle of the line** is determined by the heading of the pen and can not be altered by *Move*. This behaviour may at first be unexpected because drawing a line or path is typically realised in a vector graphics program by moving the preview line's end point directly to the current cursor position. However, in Vogo this can only be achieved by a combination of *Rotate* followed by *Move*. This on the other hand would mean that *Move* can not be created independent of *Rotate*, which precludes other constructions that require such precise control.

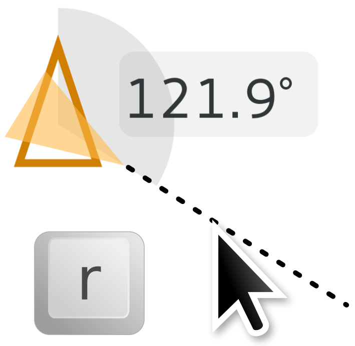
The preview line provides immediate feedback and visualises the command. It can be **altered spatially** by positioning the cursor. No text input is required to create a line, neither for typing the command nor for typing its parameter. No spelling or syntactic rules have to be known and met. *Move* is also not represented by a word but by an icon in the toolbar. The icon is a visual representations. It portraits recognisable characteristics of the object of interest (a line). In fact, programmers do not even know that the command is called *Move*. Yet it is understandable and usable without any knowledge of language and the meaning of words. The length is also shown next to the line by a continuously updated number in an input field. It can be used for precise control, but can also be ignored. Vogo does not require the programmer to think about length in terms of numbers or of lines in terms of commands. He can just see the line and see its length.

Any line's length can be altered after it has been created by simply **dragging** it. Dragged objects are coloured in violet. Note that successive commands are affected by such an action. They implicitly depend on one another through propagating pen state changes that "flow from home". Again, changes to the whole program are visible immediately. *Run* is implicit.



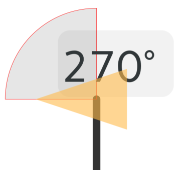
3.5 Rotate

While *Move* is explicitly visible in Turtle Graphics, *Rotate* is not. It can only be guessed from the changes in the heading of lines. This guessing is ambiguous. *right 90* can not be distinguished from *repeat 2 [left 135]* and *forward 10 right 90 back 10* not from *forward 10 left 90 forward 10*. Vogo introduces a new representation for *Rotate* which is the **geometric angle**. It is visualised by a pale grey **circular sector**. It has a centre and two sides. The centre is positioned to wherever the current pen is. One side points to its current heading while the other points to where it will be after the applied *Rotate*. It is the second item in the toolbar. The key *r* creates a preview, *click* finishes, *escape* aborts. The heading is adjusted to be in one line with the cursor's position. Adjusting an existing angle via **drag** works in the same manner.

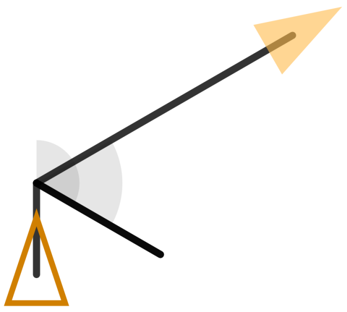


Angles are given in **degrees**. The *Rotate* parameter in the input field is followed by a degree sign ° to clarify that. Degrees tend to be used in *geometric* contexts while radians tend to be used in *algebraic* contexts. Logo uses degrees but is an exception. Most programming languages use radians. Vogo is implemented in Javascript which also uses radians. Parameters can also be expressions, which are in Javascript syntax. *Math* is Javascript's library for mathematic operations. This is why using *Math.sin(90°)* in Vogo will lead to errors. Vogo provides its own interface to trigonometric functions: *vogo.sin, .cos, .tan, .asin, .acos & .atan* which take or return degrees respectively. They have been used in the *arrowhead* example. Degrees are used in Vogo for the same reason they are used in Logo: they are preferable to radians in geometric contexts because significant headings (360°, 270°, 180°, 90°, 45°, ...) are integers, instead of irrational numbers.

Logo's *right/rt* and *left/lt* are again reduced into *Rotate*. Its **default orientation is clockwise** rotation. Negative angles rotate counter-clockwise. The circular sector is never larger than 180° or smaller than -180° even if the angle is. 270° is equivalent to -90°, 400° to 40°.

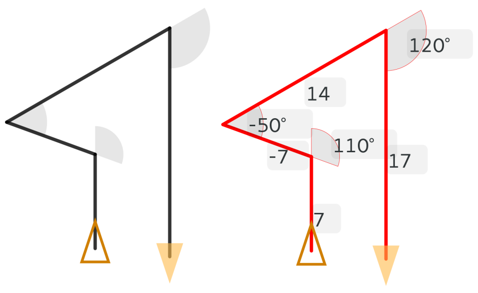


The **radius** of the circular sector can be influenced by the length of the following line. This has two advantages. **1)** the radius of the circular sector does not exceed the following line in length. **2)** Overlapping *Rotate* can easier be distinguished. This is aided by the fact that circular sectors are semi-transparent. The structure of the graphic to the right can be determined unambiguously:

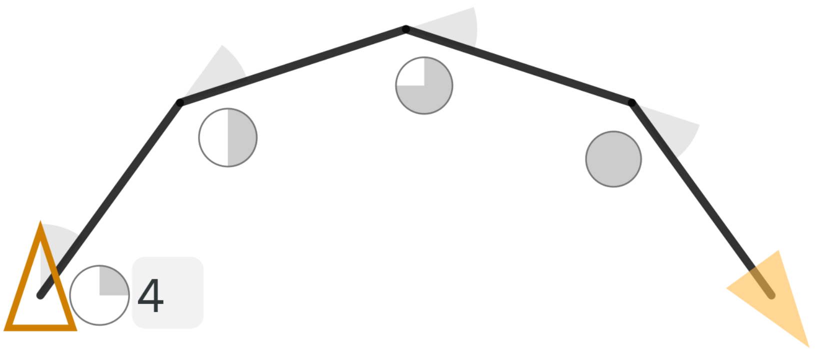


```
Move +
Rotate +
Move +x
Move −x
Rotate −
Move +
```

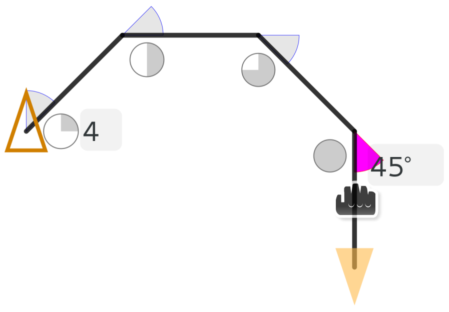
The exact values are not visible by default to avoid clutter in the interface. However, steps can be selected. Selected steps turn red and input fields become visible again. The figure to the right has three *types of angles*: **cornered**, **free-standing** and **sided**. Those give clues about the direction of the attached lines. Cornered angles occur when the incoming *Move* is negative, but the outgoing is positive. Angles are free-standing in the opposite case: when the incoming *Move* is positive, but the outgoing is negative. Sided angles occur in any other case. Those distinct visual clues make it possible to reliably determine the program structure from the spatial representations.



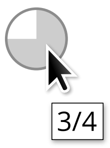
3.6 Loop



Loop is the third command in the toolbar, shortcut *l*. It is the first command that has its own scope. It contains others commands. Vogo's *Loop* is equivalent to Logo's *repeat*. It is the simplest form of iteration: do whatever is in the scope x-times. It therefore has two parameters: x and the commands it contains. The commands are determined by the current selection and x initially defaults to two. That means the programmer is **forced to first create what he wants to loop**. Loops can not be created without a body. If this is attempted, a warning is shown in the notification area. This decision was made for two reasons: 1) it is easier to visualise a non-empty loop and 2) it is in line with the principle *start concrete, than generalise.*



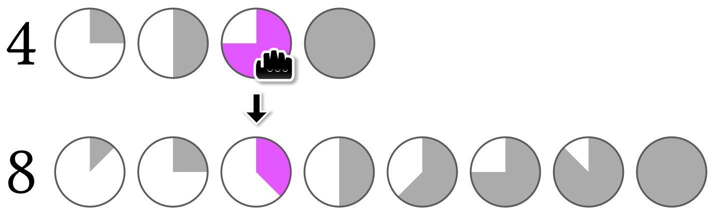
A loop creates **dependencies between elements** because it replicates commands. Copies are not unique. They are bound to their root. The figure at the top shows *Loop 4 [Move 7 Rotate 36°]*. If any one of the path segments or angles change, so do the others. They are proxies to their root command. The figure to the right shown what happens when the last *Rotate* is dragged to 45°. The proxy delegates the change to the root which then propagates the change back to all proxies. At 90° a square emerges, a triangle at 120° and so on. This drag provides immediate feedback by instantly updating the entire program. Note that the dragged element is violet and all of its **proxies have a blue outline**.



The *Loop* is visualised in the toolbar by an icon that shows a clockwise turn. Each step in the turn is one iteration within one full revolution. This is **metaphoric of a clock or a pie**. On the canvas, each progression in the iteration adds a "tick to the clock" or a "pie to the cake". The icon is placed next to the position of the pen at the start of the current iteration. Hovering the mouse over the "loop clock" reveals a tooltip showing the current iteration / the

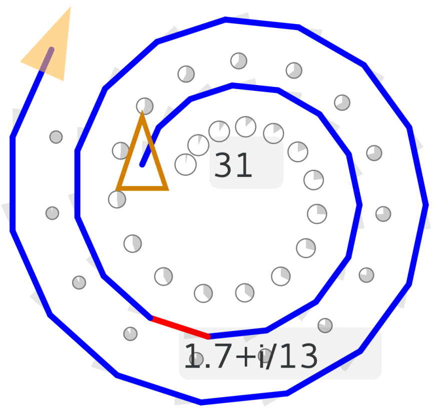
total number of repetitions. The clock invokes the idea of **progressing time**.
This is a useful analogy because each loop iteration also progresses the pen forward, drawing a path in its wake. Each new segment depends on the entire history. Another more subtle advantage is that a clock **transitions smoothly**. If the number of iterations is increased continuously, so does the visual quality of the icon, without abrupt changes. Digits are different because of their symbolic nature.

The first iteration always has an attached input field. It can be used to change the number of iterations. But there is also a way to directly **"wind up the clock"**: by dragging its hand to the desired position:



The higher the iteration that is picked, the "longer the lever". For example, dragging 5/6 to 1/7 squeezes 5 into 1/7, increasing the number of repetitions to 5*7=35.

The current iteration inside the scope of a loop can be used in expressions by accessing the **implicit parameter *i***. The figure to the right shows *Loop 31 [Move 1.7+i/13 Rotate 24°]*. The length of the line increases in each step, creating a spiral. The expression is visible because the red line proxy is selected. All other proxies are blue.



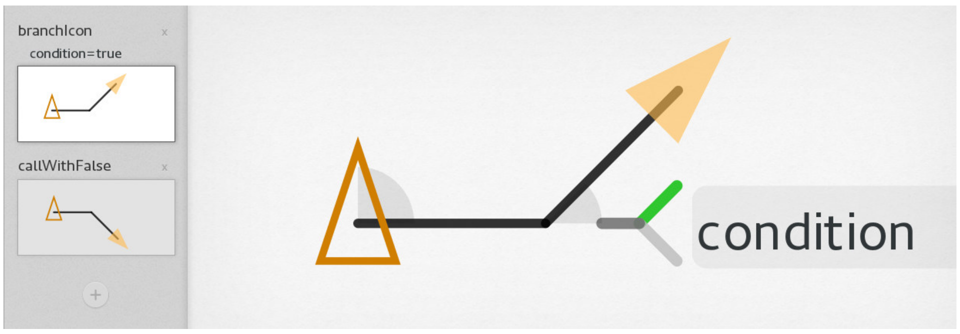
Another important feature of loops is that their **icon's size decreases** absolutely with the number of repetitions and decreases relatively with the number of iterations. This is done to better reflect its decreasing importance, but also to reduce clutter and to introduce the ability to better tell apart multiple loops of different size.

3.7 Branch

Branch is the name and metaphor of this command at the same time. *Branch* is equivalent to a traditional *if-then-else* statement but frames it in terms of **flow control**. Flow direction is either channelled into the *True Branch* or the *False Branch*. The two are mutually exclusive.

Branches have a scope and can contain more branches. The deeper the scope depth, the taller the tree they span. Since the pen "travels on a path", a branch is a suitable analogy for a junction where a decision is made to either follow the flow in the *True* or *False* direction.

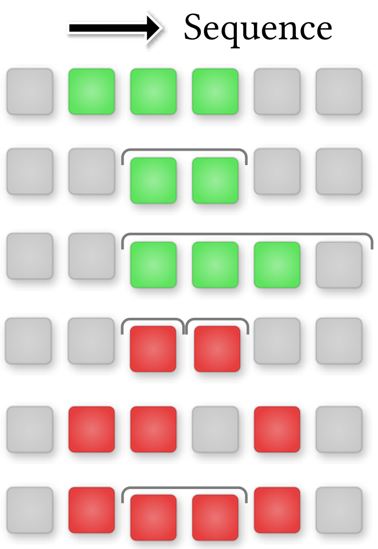
This is the **Branch icon created in Vogo** with the help of *Branch*:



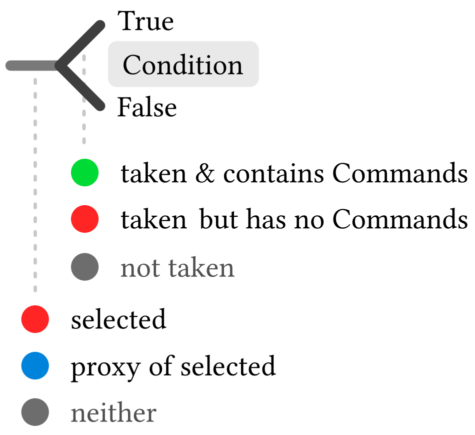
Like the loop icon, the branch icon is placed next to the pen state current at its invocation. The "upper branch" of the icon is coloured green to indicate that the condition is true. The condition here is the parameter with the name *condition* which defaults to *true*. The subpicture *branchIcon* is called by the second subpicture *callWithFalse* with the parameter *false*. Its preview shows that *Branch* indeed controls the angle. The subpicture *branchIcon* in pseudo-code:

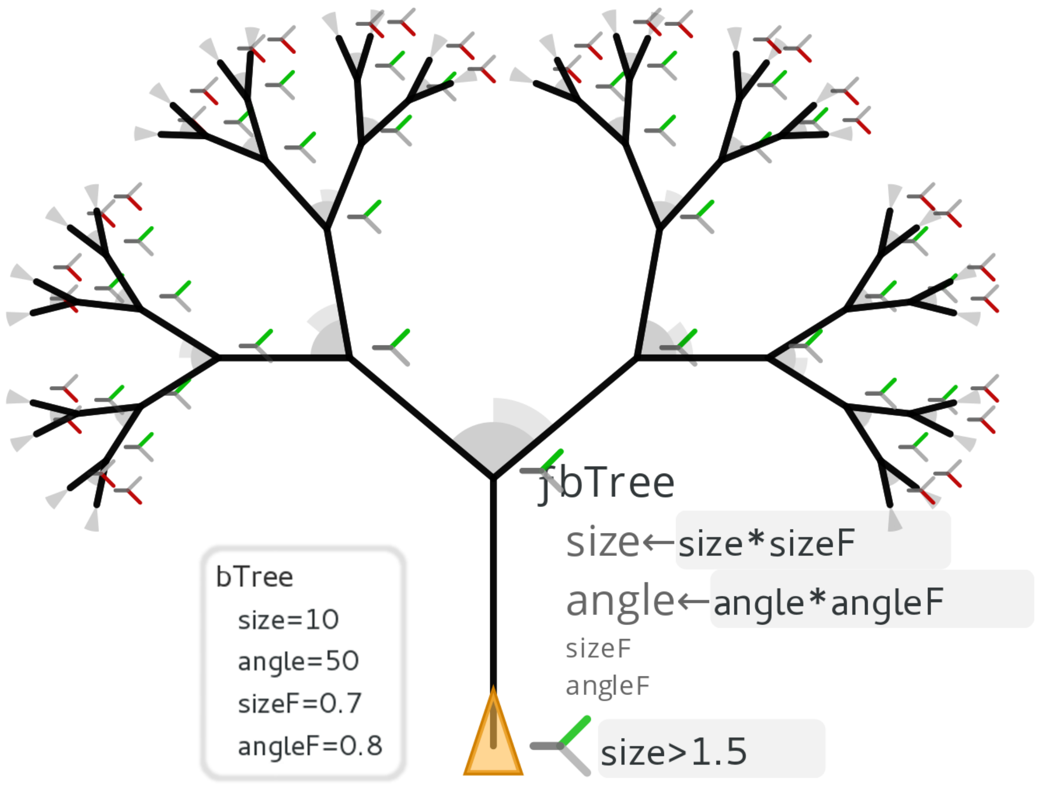
```
Rotate 90
Move 5
Branch condition [ Rotate −45 ] [ Rotate 45 ]
Move 5
```

A branch has three parameters: the condition and the two command blocks. It is created in the same manner loops are. First, the programmer has to select the commands he wants contained inside the *True* branch. He then hits *b* or selects *Branch* from the toolbar. The selected **commands have to be connected** and be inside the same scope, which means there may not be gaps inside their chain. This is true for loops too. The reason for this requirement is that it forces the programmer to unambiguously specify the block he wants to create. If gaps exist, the programmer may have overseen something, or he may have intended to create multiple branches or he may have wanted to lump the parts together. The intent is not clear. Instead of guessing, the environment forces the programmer to clarify. The selected commands are then surrounded by the new scope. The condition defaults to the literal "true" and the *False* branch is initially empty.



The *True* and *False* branches are **allowed to be empty**. After the condition is set to a meaningful expression, *Branch* can be thought of as a switch. I here refer to the *electric switch*, not the code construct, which acts in a completely different way. If the condition is met, the flow is allowed to continue. If not, the flow is stopped because the *False* branch is empty. In this case, the respective icon's branch is coloured red. This analogy is particularly useful if the branch is not followed by more steps but ends the scope it is contained in. Since branches are routinely uses to **terminate recursions**, this is not seldom the case. A binary tree is an illustrative example:





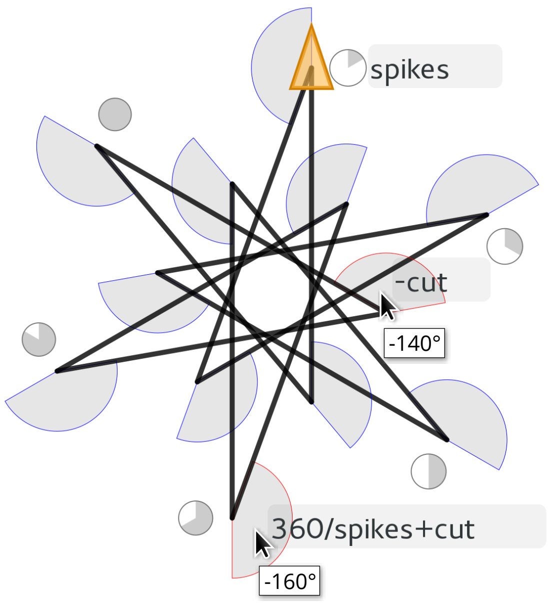
The details of this example can not be understood until later in this chapter. The important thing to note though is that a *Branch* is spanning the entire body of *bTree*. It only lets the two-fold recursion progress if the *size* of each step remains greater then 1.5. Once this condition is violated by the ever decreasing *size*, execution is halted, which is visualised by the **red branches in every leaf**.

A second characteristic of *Branch* is illustrated in this example. The **size of the icons slightly decreases** with the depth of the recursion. As was the case with *Loop*, this is done to reduce clutter and make different branches distinguishable. In addition, only the first or currently selected *Branch* proxy shows the condition.

3.8 Selection & Editing

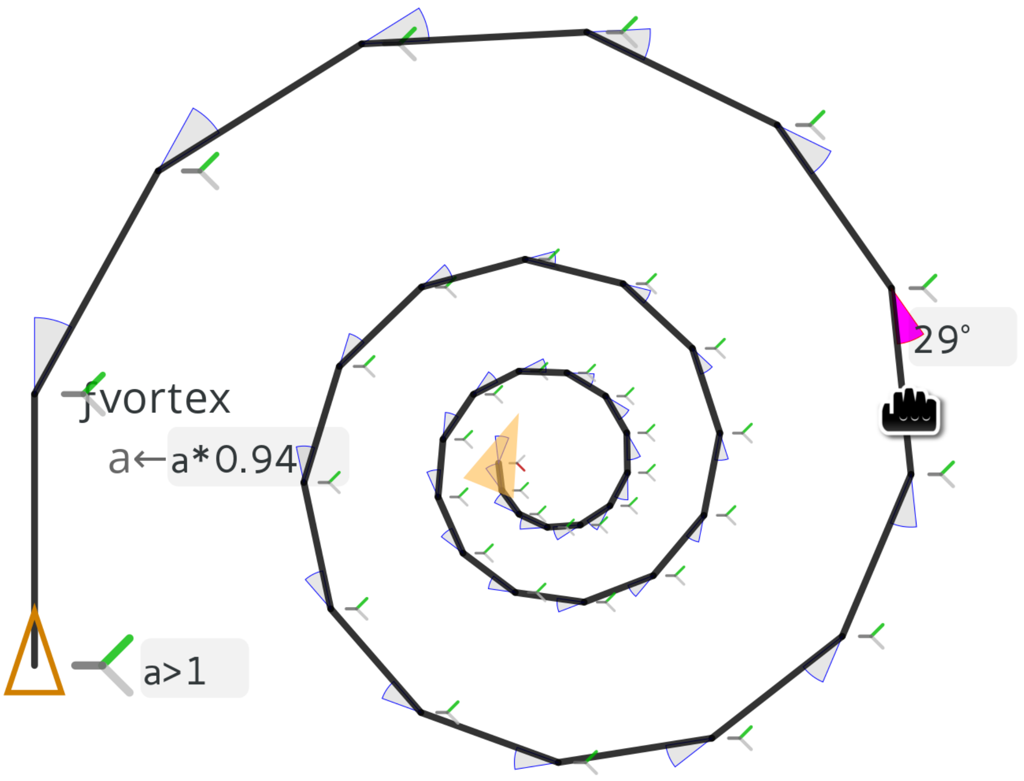
The last section left open how the *False* branch can be filled with commands or how commands in the *True* branch can be added, replaced and deleted. This section explains Vogo's selection and editing system. It was already mentioned that three colours are used throughout Vogo. Red is used to indicate selected commands, blue is used to indicate proxies of the current selection and violet is used to indicate dragged elements. All commands can be **selected by simply clicking** on their visual representation. *Move* is selected by clicking on its line. *Rotate* is selected by clicking on the circular sector. *Loop* is selected by clicking on any clock. *Branch* is selected by clicking the left-hand "root" line of the icon. Clicking on the background of the canvas deselects everything.

Selection is generally **akin to vector graphics editing programs** and therefore works largely as expected. The *delete* key deletes whatever is selected. If commands are deleted that contain other commands, those too are erased. Holding down the *shift* key allows **accumulative selection**. Selecting already selected commands while *shift* is pressed deselects them. Proxies are selectable but refer to their respective root. Selecting multiple proxies with the same root is therefore not possible. If multiple proxies with different roots are selected, all their combined proxies are coloured blue. For example, the star to the right contains two root angles (inner and outer). One proxy of both was selected (red) with accumulative selection. All other proxies are blue.



As a side note, hovering over any *Rotate* reveals in a **tooltip the exact computed angle** for a particular set of parameters, here *cut=140, spikes=6, size=100.* This creates an immediate connection between abstract representation (expression) and concrete example. It also reveals that the inner and outer angles are not identical, which may have remained unnoticed at the first glance.

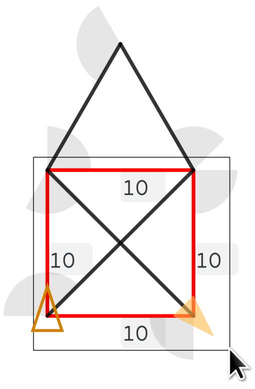
Elements inside calls to subpictures are not selectable intentionally. This is to prevent alterations to functions from within other functions in order to enforce encapsulation. Subpicture are only to be editable when they are focused. The call itself is selectable. There is one **exception** to this rule: when a function calls itself. Elements inside **self-recursive calls** are selectable, but again, only from within the parent function. Calling a recursive subpicture is creating an own scope which is not selectable. This exception does not apply to cyclic recursion schemes which involve multiple functions that reference each other in a way that creates cycles in the call tree. This is true not only for selection but also for editing. Lines can normally be dragged in order to change their length, but this is only true if their root is a native of the focused subpicture.

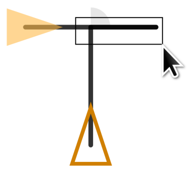


The above example shows the one-fold self-recursive subpicture *vortex*. Angles can be edited at any depth in the recursion to tighten or relax the "pull of the vortex". All but the first *Rotate* proxy are inside a function call and yet selectable and editable.

a=10 in the above example. It controls the length of each segment and is continuously decreased in each call. Again, the *Branch* acts like a switch. $a^{*}0.94^{x}=1$ yields $x \approx 37.2$, which means that 37 calls are made before the recursion terminates, which is indicated by the last red branch.

Dragging with the left mouse over the canvas creates a **selection rectangle**. On release, everything fully inside the rectangle is selected. Multiple elements can be selected at once. If *shift* is pressed on release, the current selection is extended by the new elements, but reduced by those already selected. It behaves like an *exclusive or*. For example, in the figure to the right, if *shift* is pressed on release, the current selection will be deselected, because it is inside the selection rectangle and replaced by the two crossing lines in the middle. No angle is selected because none is fully contained within the rectangle.

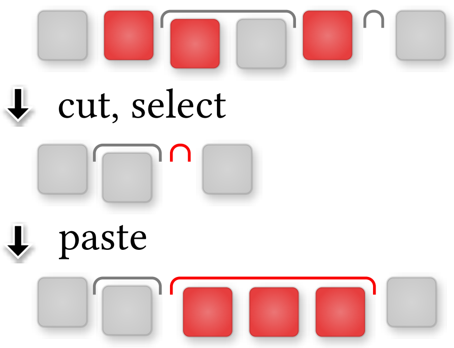




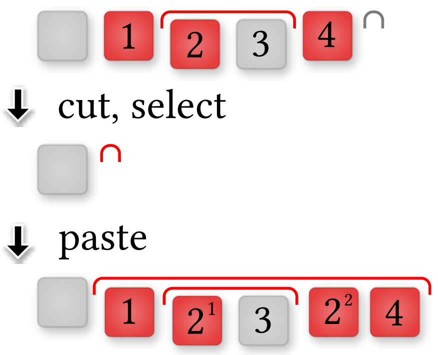
Rectangular selection is also useful to **select elements that hide** behind others. Lines are black but have a slight transparency. Two lines that overlap are therefore slightly darker which makes it possible to spot them, in case the context itself does not already hint at their existence. For example, blind alleys always reveal overlapping lines.

Elements that create own scopes (*Loop, Branch & Call*) are always **favourably selected if everything they contain is**. For example, if the complete *vortex* was surrounded by a selection rectangle, only the *Branch* would get selected because it is the outermost scope that everything else is contained in. But if everything except *home* was included, the recursive *Call* would get selected.

Cut, copy and paste are available with *control+x,c,v*. Cut = copy + delete. Compared to creating *Loop* and *Branch*, cut and copy pose no restrictions on the selection. The selection is simply "flattened" by freeing its elements from their previous scope. The figure to the right shows how the three red elements are moved into a new scope. Their order is retained and they are detached from their old scope.



It is also possible to select and copy an element that has a scope *and* elements inside that scope. This is illustrated in the second example. The command with number two is duplicated by the cut because the scope is copied with all of its contents, irregardless of whether they are selected. Appending copied scopes to themselves does not cause conflicts.



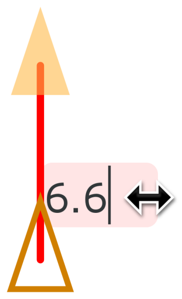
If no elements are selected, **pasting** appends to the end of the program, because this is assumed to be the most relevant location for insertion. If an element that contains other elements is selected, pasting appends *inside* the selected scope. If an element is selected that does not contain other elements, pasting prepends. If multiple elements are selected, only the first is considered for deciding where to insert. The rules described here also apply to creating elements. Elements are created and inserted before or inside the first selected command. *Branch* has two scopes. Commands are always inserted into the active one. Naturally, the *True* branch is active if the condition evaluates to *true* and the *False* branch otherwise. Unfortunately, there is one position that is **not reachable for insertion: before a scoping element**. This is currently a limitation. A possible workaround is to duplicate the element that sits directly before the scope, insert between the duplicates and then delete the second duplicate. What also works is to insert behind the scope instead, then cut the scope and paste it behind the insertion.

Nevertheless, the typical insertion point is at the end of the program. *Move* and *Rotate* create previews at the end by default. As mentioned earlier, a line does not point directly to the cursor position without a *Rotate* in front of it. However, since the *Rotate* does exactly orient the current heading towards the cursor, a line can easily be created by positioning the cursor to the desired location and then hitting *r* and *d* right after one another. *d* automatically finishes the pending *Rotate* preview and vice versa. This allows for **rapid and precise drawing** in Vogo.

3.9 Expressions & Dragging

All text on the canvas is exclusively used in expressions for **parameter assignment**. Expressions are contained in input fields. All can be edited. The only exception is the function call, which is parameterised with an immutable reference to a function. Expressions have to evaluate to numbers for *Move*, *Rotate*, *Loop* and to boolean for *Branch*. If that is not possible the evaluation defaults to *1* or *true*. Expressions may be any valid Javascript[10].

Since it is a declared goal to minimise textual editing, manipulation of expressions is avoided wherever feasible. Expressions are always created and updated automatically and instantly. For example, this is the case when moving a preview line with the cursor or dragging a selected line. Aside from the direct manipulation and the textual editing of the expression, there is a third option: **dragging constants**. Expressions can be dragged with the mouse if they are constant numbers. The idea stems for Victor's *Scrubbing Calculator*[11]. Dragging linearly adjusts the number along the horizontal drag axis. The **adjustment factor depends on the magnitude and the granularity** of the number at the start of the drag. The higher the magnitude, the higher the factor. The more decimal places, the lower the

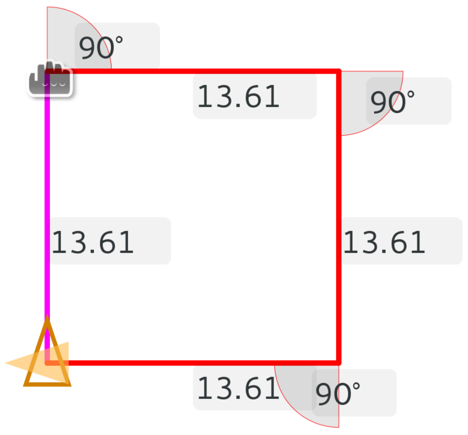


[10] eval(), http://www.ecma-international.org/ecma-262/5.1/#sec-15.1.2.1
[11] http://worrydream.com/ScrubbingCalculator/

factor. Adding or reducing decimal places before starting the drag therefore changes the desired drag range. The number of repetitions in a loop is restricted to integers.

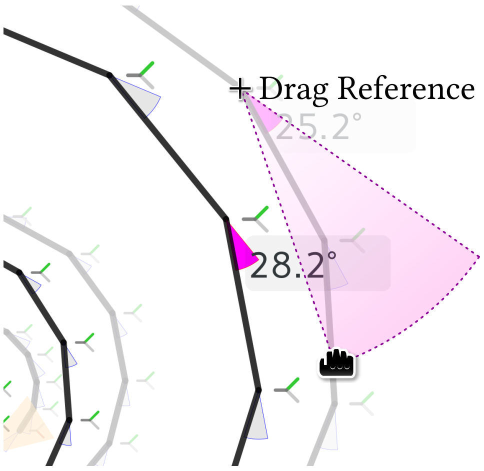
The program is constantly updated whilst dragging. This **encourages exploring** the influence of parameters on the behaviour of the graphic. The movement of the hand is **intimately coupled** with the movement of the construction the screen. The input is tactile (hand movement of the mouse) and the feedback visual. This allows the eye to stay focused on the object of interest while it is being manipulated over the kinaesthetic channel. A simultaneous and synchronous communication with the program can be established. Moreover, the control continuously transitions visual qualities in *correlation* with the hand movement. This is the prime realisation of direct manipulation of Turtle Graphics proposed in this thesis.

Multiple selected commands of the same type can be edited at once. This works with all editing styles: direct spatial manipulation, textual editing of the expression and the aforementioned dragging of constants. Changing multiple at once combined with a choice of editing style can result in a tremendous improvement in editing speed compared to traditional textual editing. In the example on the right, a square was created by first "free-handing" four lines that are connected by angles, then selecting everything and setting the angle to 90° and then dragging any line to the desired side length. Note that no reselection is required to pick out the angles or the lines. Changes only affect commands of the same type inside the selection.

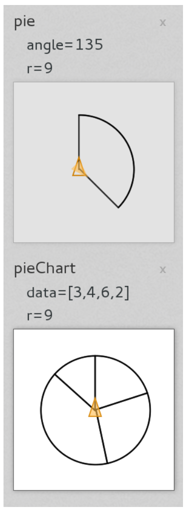


The mentioned **correlation between hand movement and spatial movement** may be more or less proportionate. The above square example is illustrative. Dragging the line in the *west* moves the line's end point exactly with the vertical position of the mouse. The position of the start point remains unchanged. The proportionality factor is 1. Dragging the line in the *north* still (taking the different orientation into account) moves the line's end point exactly with the horizontal position of the mouse. However, vertical motion of the line is added. The proportionality factor is still 1, but the horizontal movement is compounded with a vertical one. Dragging the line in the *east* is the most confusing. Neither the start nor the end point move exactly with the vertical mouse movement. Moving the mouse down moves the line up. The proportionality factor is -1. Again, this is compounded with horizontal motion, which stands in no correlation with the horizontal movement of the mouse, which actually has no effect on either. Dragging the line in the *south* also moves contrary to the mouse, but at least remains static in the vertical. The correlation worsens from the *west* to the *north* to the *south* to the *east*.

How the graphic behaves depends solely on the structure of the program. For example, consider the earlier *vortex* subpicture where any angle could be dragged to tighten or relax its pull. How precisely does the movement of the mouse during the drag correlate with the size of the angle or the pull? That depends on the angle. If the first angle is dragged, which does not depend on any previous angle, the angles position during the drag is constant, which creates a predictable proportionality. The angle and pull increase with the increasing rotation of the cursor around the angle. But other angles depend on their precursors, which means that their **position does not remain static during the drag**. The figure to the right shows such a situation. If the dynamic position of the angle was to be taken into consideration for calculating the angle adjustment, which depends on the relative position between angle and cursor, the first change of the angle would trigger an avalanche of self-reinforcing corrective adjustments. To avoid this, Vogo stores the state of the dragged object at the start of the drag and uses it as the sole reference, irregardless of the actual (potentially altered) position and heading. This stabilises the drag behaviour but does not necessarily create well-formed proportionality. The closer the dragged angle is to the epicentre of the vortex, the more erratic the behaviour (higher proportionality factor), which is obviously a property of the vortex itself, not of the drag interaction that the environment provides. It follows, put in simplified terms, that the environment can not provide a better drag interaction without actually understanding the particular program.



3.10 Subpictures



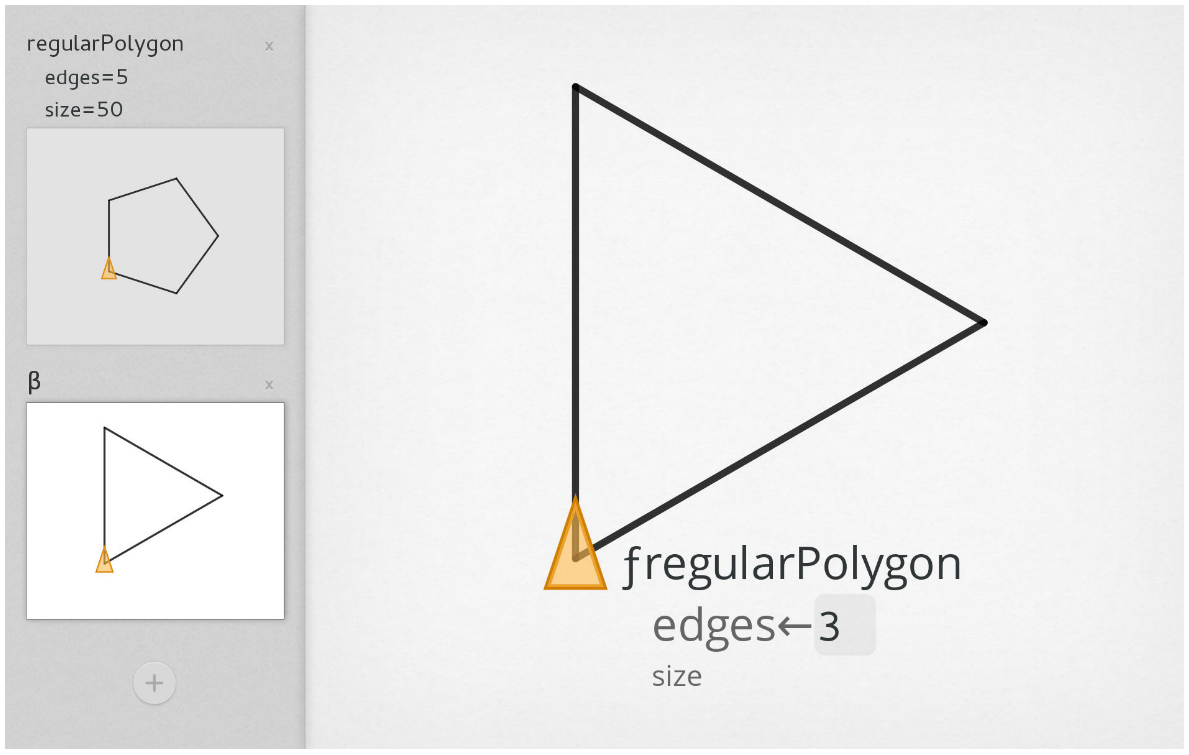
Subpictures are Vogo's functions. They encapsulate a partial drawing for customised reuse. The subpictures panel prominently displays them with a preview image. If the subpicture has parameters, this image is only one incarnation of the function's abstract nature, but it does at least provide a visual hint at it. Subpictures have an adjustable name. The name defaults to a Greek letter in order to free the programmer from having the specify a name, if he deems it sufficient to identify subpictures by their preview only. Focused subpictures are displayed on the canvas, one at a time. The preview only contains the home, pen and the lines, but shows no angles or others controls. The preview is otherwise tied to the look on the canvas and therefore created as a byproduct.

Subpictures "return" a graphic and a pen state change. Their input are parameters, which are listed below the name of the subpicture.

Parameters have a name and a default expression. **Parameters are not variables.** They can be set on invocation of a function, but do not change inside any given scope. It is also not possible to let parameters reference each other. For example, it is not possible to set *a=10* and *b=a+a*. Logo has variables but Vogo strictly bans their usage to enforce a more functional programming style.

Just like any other expression, default **parameters can be dragged** if they are constants. This changes the execution environment of the subpicture, which updated accordingly. Parameters can be adjusted even if the subpicture they belong to is not focused. The changes are immediately visible in the preview and propagate to all subpictures that depend on it through referencing. This makes it possible to understand subpictures "from the outside" by looking at its preview and playing with its parameters. It is therefore feasible to understand whether a subpicture provides a certain desired graphical component without needing to grasp its internal mechanisms, which is important for effective encapsulation.

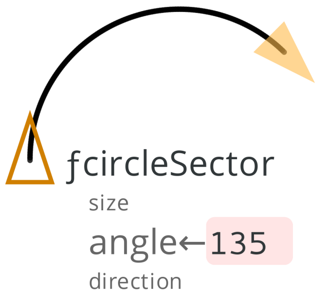
3.11 Call



The *Call* is the fifth and last command in Vogo. As the name suggests, *Call* is used to instantiate subpictures. The *Call* is the only command that is not listed in the toolbox. *Call* is instead invoked by **dragging a subpicture onto the canvas**. The subpicture picked is automatically set as the reference that is to be called. Calling a subpicture in Vogo is simply another drag operation. Programmers do not need to type its name or arguments. By default, the arguments list, *Call's* second parameter, is empty. Since no arguments are initially set to the called subpicture, the default expressions are used. **Arguments can be selectively overwritten.** Typically, arguments can only be overwritten in order. For example, the Javascript function *function f (a, b)* can only be called with none, one or two parameters, but can not be called with only *b* being set. In contrast, setting parameters in Vogo is independent of order, which is more flexible.

***f* is used to symbolise a call.** It is followed by the name of the subpicture. Renaming the subpicture does not break the call. Its name is just updated accordingly. This is another advantage over textual program editing. Only "smart" development environments will refactor a rename correctly. The call is the only command with a symbolic (textual) representation. However, since each call actually draws the subpicture, this representation is only meant to indicate that a subpicture was drawn. The graphic stands for itself. *f* is an analogy to functions in math and programming. I do not consider it to be a good design decision but have not yet come up with a better representation.

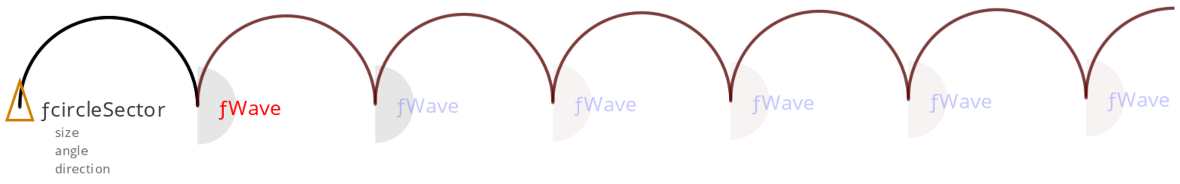
Arguments are listed below the name. Clicking on an argument "activates" or "unbinds" it for custom setting. ← **symbolises assignment.** This differs from the traditional use of = or := to indicate assignment. The arrow does not creating confusions with *equals* and indicates a direction from the expression to the parameter. Note that the default parameters do use the = but in the subpicture context it actually does mean *equals*.



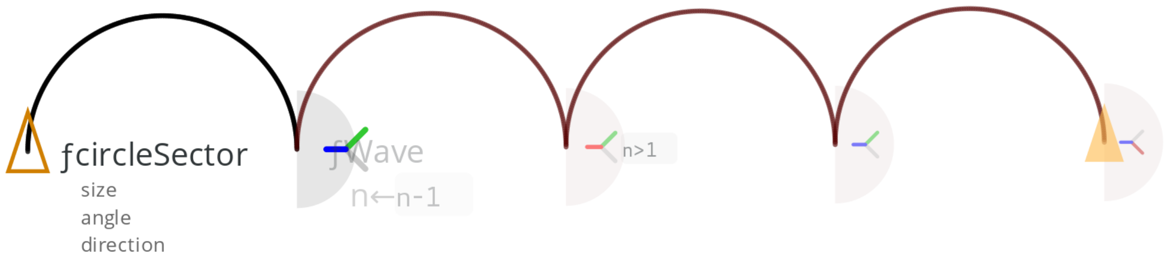
In the figure *angle* is "rebound" or "overwritten" while the other arguments remain inactivated. This scheme allows programmers to quickly access only the functionality they need. It is in line with the environment requirement *create by reacting*, which can be paraphrased as **start somewhere, then sculpt**. The idea is to get a visible result on the screen as soon as possible. The impression can then be used as a stepping stone to inform the next steps.

A part of a called subpicture can also be distinguished by its lack of controls. For example, angles are not displayed. Calls hide the internals of the called subpicture. It can not be edited "from the outside".

3.12 Recursion

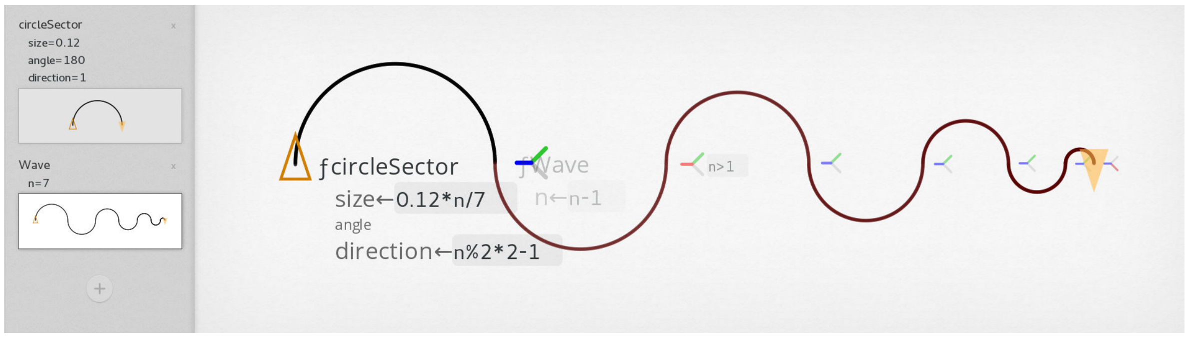


Calls can reference the subpicture they are a part of. The *circleSector* is to be used to create a wave. *Wave* calls *circleSector*, rotates 180° and then calls itself. This creates an **endless recursion**. Vogo is able to visually approximate an endless recursion by computing it until a certain **scope depth ceiling** is reached. The symbols of successive recursive calls are hidden by default to avoid cluttering the interface. In the above figure, the first recursive call is selected, which reveals all of its proxies in blue. In order to terminate the recursion, a *Branch* can be used. First, the parameter *n* is added to *Wave*, which is set to default to 4. Then, the recursive call is set to decrease *n* in every iteration. At last, the recursive call is selected and branched at the condition *n>1*. This results in the following program:



The recursion is terminated after the third call. Note that the second *Branch* proxy is selected. Its root's scope contains the call to *Wave* which itself contains the call to *circleSector*, the angle and the next *Branch* proxy. Containment in scopes of selected commands is visualised by low opacity. This is why the recursive call and everything it contains has a pale colour.

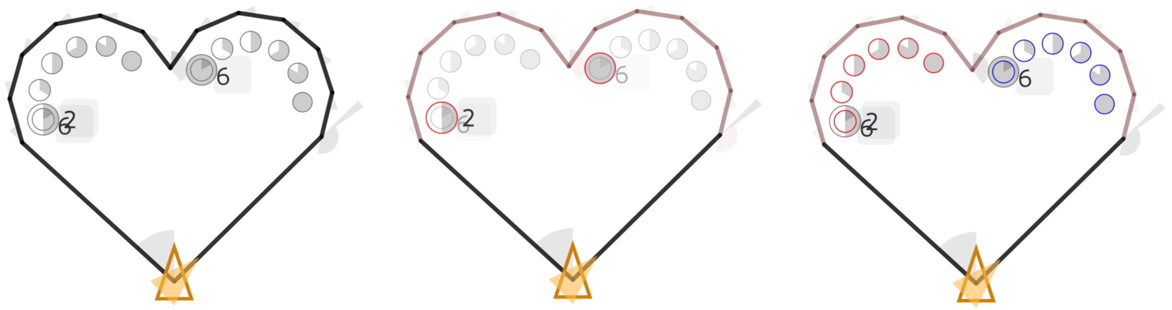
This is not a proper wave yet. The direction of the *circleSector* can be set to alternate between 1 and -1. The previous angle is then obsolete. And waves do decrease in size over time.



This walkthrough example shows many individual design principles in action, but above all illustrates **how they play together**. Artists could not *create by reacting* if they had no *immediate feedback*, which is available only due to the ability to *see every step*, which in turn relies on the *grounding of abstractions to concrete examples*.

3.13 Scope

Code structures programs with indentation into blocks of commands. At run-time those blocks create scopes. **Blocks** can contain blocks and **scopes** can contain scopes but the two are not identical. The crucial difference is that a call creates a new scope, but it does not create a new block. A block is structuring the static program representation while a scope is structuring the dynamic program representation. There are still considerable overlaps. Everything inside a block is also in the same scope, but the reverse is not necessarily true. For example, the scope of an endlessly recursing program is also endless, but the function's command block is not. Logo uses dynamic scoping while **Vogo uses lexical scoping**.



Loop, *Branch* and *Call* create scopes in Vogo. Seeing scopes is important to understand the behaviour and structure of the program. The scope of a command can be **made visible explicitly by selecting** it, which results in a reduction of opacity in all contained elements. Nested loops, as illustrated in the above example, are indicated by the reduced size of the loop icons. Selecting first the outer and then one of the inner loops reveals the program structure unambiguously.

```
Rotate
Move
Loop 2
  Loop 6
    Rotate
    Move
  Rotate
Rotate
Move
```

In the previous *Wave* example, the scope of its *Branch* was shown. Its block only contains the recursive call, but its scope contains the complete recursive chain of events *including its own proxies.* The reason for choosing opacity as a means to visualise scope is that it works on all elements without occluding the visibility of other properties, like selection and proxies. It also creates a contrast to the surrounding program.

3.14 Parametrisation

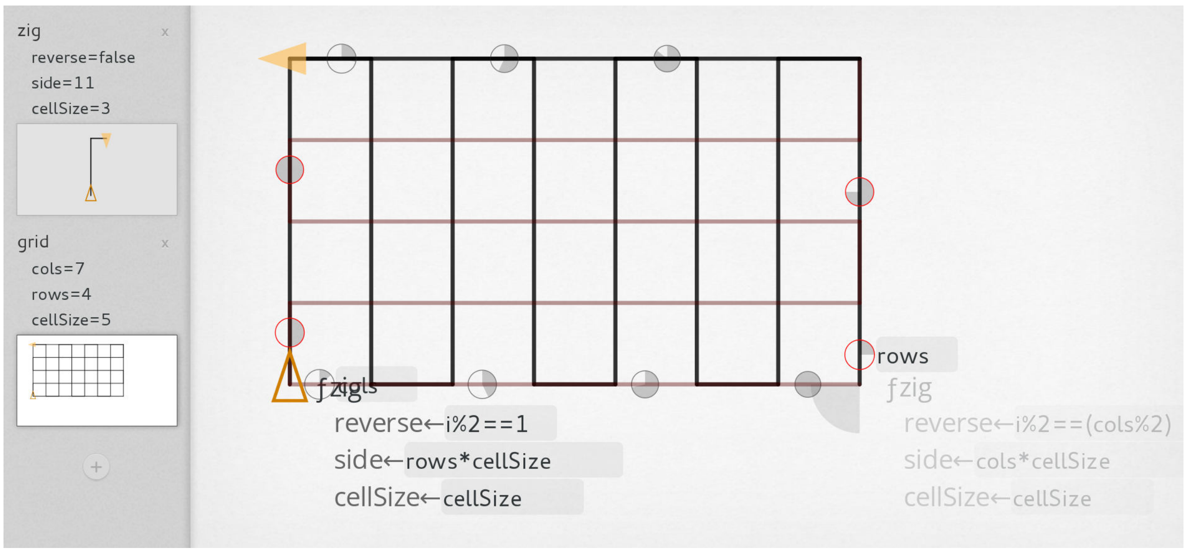
Parametrisation is the **principal mechanism for introducing abstraction**. It has its own place in the toolbar and is symbolised by a ladder, which is inspired by Victor's *Ladder of Abstraction*. It is the first item in the toolbar that does not create a command. Pressing it adds a new parameter to the focused subpicture. Its name defaults to a Latin letter, its value to 1. Both can be changed. However, references to renamed parameters have to be updated manually because Vogo does not (yet) "understand" expressions semantically, which would require the parsing of Javascript inside itself.

Everything that is currently selected is set to the newly created parameter: *Move's* length, *Rotate's* angle, *Loop's* number of repetitions, *Branch's* condition. Only *Call* can not be parameterised that way. In addition, if the selection is not empty, the parameter does not default to 1 but to the primary parameter of the first selected command. That is useful for two reasons. **1)** the attributes of all existing commands already provide a default value. If the programmer wants to abstract over a command, he probably wants to retain the values that are already in place. **2)** Multiple commands can be set to refer to the same new parameter at once, which increases editing speed.

Parameters can be deleted by setting its input field in the subpictures panel to the empty string. Theoretically, parameters can also be Javascript functions *a=(function(b) { return b*b })* that are evaluated in the program *a(b)* using other parameters.

3.15 Flow

What happened when? It is not always trivial to figure out the flow. The following program draws a grid with an adjustable number of columns, rows and cell size. Due to many intersections, overlaps and only two dominant headings, it can be hard to figure out the path the pen travelled. The most important fact to know is that **the pen does not jump around**. Otherwise it would be a hopeless quest to figure out the flow in Vogo without any further help. In contrast, Logo's turtle can raise the pen with *penup* and lower the pen with *pendown* which does allow visual jumps of the stroke to occur in the graphic. The same is possible with commands like *setpos* or *home*, which allows absolute positioning. Logo's code view provides a way to tell the sequential structure of the program. Since this is not the case in Vogo, absolute positioning is intentionally left out. The downside of that decision is that **everything has to be connected**. This may be alleviated in the future by adding the ability to hide lines on export.



zig
reverse=false
side=11
cellSize=3

grid
cols=7
rows=4
cellSize=5

ƒzig
reverse←i%2==1
side←rows*cellSize
cellSize←cellSize

rows
ƒzig
reverse←i%2==(cols%2)
side←cols*cellSize
cellSize←cellSize

Flow control statements and the discussed ability to see scopes and blocks are crucial for understanding the pen movement. As shown in the figure, one click on the loop reveals the last pieces of the puzzle. First of all, there are two loops: one for the rows and one for the columns. Each even iteration starts on the opposite side of the grid. *zig* is called in every iteration and its preview reveals that it draws one half of a complete column or row. Its ability to *reverse* indicates that it can change direction and indeed it is called in the loop with an alternating argument. It follows that the grid is woven by first drawing the long vertical lines from left to right, in a zig-zag pattern. *zig* does not draw the final closing border. This is done "behind" the loop. This leaves the pen standing at either the top or bottom right end of the graphic depending on whether the number of columns is odd or even. Then, the horizontal lines are drawn. The rows-loop is preceded by a rotation, 90° or -90° depending on whether columns is odd or not. In pseudo-code:

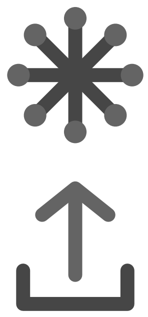
```
zig:
  Move side
  Rotate 90*(reverse ? −1 : 1)
  Move cellSize
  Rotate 90*(reverse ? −1 : 1)

grid:
  Loop cols
    Call zig {reverse: i%2==1, side: rows* cellSize , cellSize : cellSize }
  Move rows* cellSize
  Rotate 90*(cols%2==1 ? 1 : −1)
  Loop rows
    Call zig {reverse: i%2==(cols%2), side: cols * cellSize , cellSize : cellSize }
  Move cols* cellSize
```

The naming of parameters also significantly contributes to the understanding. If it is not clear that `%` is modulo, it is possible to play with the parameters and observe how the angle alternates. And if nothing else helps, the programmer can simply *deconstruct* the program. Deleting a loop immediately unravels the zig-zag pattern. The following characteristics of Vogo **improve the understandability of flow**:

- the pen draws an uninterrupted path without jumps
- its start (home) and the end (pen) state are visible
- every state change operator leaves a spatial trace or iconic hint
- flow control statements reveal how they unfold at run-time
- scopes and blocks are visible
- subpictures provide previews and expose all parameters
- the effect of parameters can be explored, change is continuous, feedback is instant
- semi-transparency reveals overlapping elements
- ability to see proxies of selected commands
- explicit and distinct spatial representation for rotations

Nevertheless, flow can not be said to be unambiguous in all cases. Overlapping elements are the gravest thread to flow visibility.

3.16 Export



Vogo graphics can be exported in a **static and a dynamic format**. The static format is SVG. The first icon in the figure on the right is the SVG icon. This export is static because the relationships and structures build between the elements is not retained. The second icon stands for dynamic export. Both can be found in the toolbar. Dynamic export generates an HTML file that contains a script, which contains the program. It is written in Javascript and tied to the Vogo API[12]. Vogo in turn depends on D3, which is loaded first. Its general structure looks as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <title>Vogo Export</title>
  <script src='http://d3js.org/d3.v3.min.js'></script>
  <script src='http://mgrf.de/vogo/js/vogo.js'></script>
</head>
<body>
<script>
```

[12]Application Programming Interface. A library of functions Vogo exposes.

```
var regularPolygon = new vogo.Func({
  name: "regularPolygon",
  args: {"n": 5},
  viewBox: {x:−17, y:−17, w:60, h :52}});
regularPolygon.setCommands([
  new vogo.Loop("n", [
    new vogo.Rotate("360/n"),
    new vogo.Move("100/n")])]);

var fd = new vogo.Drawing(regularPolygon, {arguments: {n: 20}})
fd.update({n: 3})
</script>
</body>
</html>
```

Vogo exposes the following classes: *Func*, *Move*, *Rotate*, *Loop*, *Branch*, *FuncCall* and *Drawing*. *Drawing* acts as a wrapper for functions that are drawn into a custom inline SVG container. Since none is provided here, Vogo creates a new one. *update* demonstrates that the graphic is indeed dynamic and controllable "from the outside". It can be used to **turn the dynamic graphic into an interactive graphic**.

This is not the place to explain Vogo's API in detail, but I want to hint at one other option. Vogo can also be used to **create data-driven graphics with D3**. The following snippet illustrates the principal mechanism:

```
var data = d3.range(1,9)
d3.select("#mysvg").append("g")
  .call(vogo.draw(barChart, {data: data}))
  .call(vogo.update({data: d3.shuffle(data)}))
```

3.17 Compiler Design

For a lack of a better word, I shall call it *compiler*, when in fact what is meant is more like a mini **program manipulation operating system**. A compiler translates a source into an executable form and then terminates. This is not the case in Vogo. Its compiler does not shut down. In a way, the executable form itself is what is constantly manipulated. The most important task that the "compiler" has to fulfil is to rather update the "source" in response to changes to the working representation of the program. By *update* an **incremental change** is meant, not a classical re-run that "destroys the world". This requires the compiler to "know" what parts of the program are affected by an edit as well as what parts are not. For example, if a loop's number of repetitions is reduced from three to two, it does *not* first remove the complete loop only to recreate it with two iterations, but just removes the last iteration. This approach diverges from simply re-running fast in order to achieve real-time programming (Hancock, 2003) (Tanimoto, 2013).

In spite of its name, the canvas is not implemented with the <canvas> element, but with an **embedded SVG**. The primary reason for this decision is the simplified implementation of interactivity. Vogo is build on D3, which is a superb library for manipulating the DOM[13]. In contrast, <canvas> provides a raster graphics interface. This is the secondary reason for the usage of SVG: Vogo is meant to create dynamic vector graphics, not raster graphics. SVG is a native format and natural choice. Trivial geometric manipulation does not require a (manual) redraw. However, those benefits come at the cost of an increased performance penalty. DOM manipulation and styling is performance intensive. Vogo is **optimised towards reducing the load on the DOM**. This is where the aforementioned *incremental updates* go hand in hand with performance.

An integral part of the compiler is the **proxy system** that was mentioned in section 3.3. The proxy system can be understood as a new **semantic layer** on top of the static program structure. Proxies are run-time objects that are spawned by root commands. Everything on the canvas is a proxy, even if it is only a trivial one (has no siblings). For example, a function call spawns proxies of the called function's root commands. Proxies depend on their roots. If any command in the called function is altered, all proxies update accordingly. Due to the persistent, bi-directional link between roots and proxies, the effect of changes stays localised. This is true up the level of the function itself. Changes in one subpicture only trigger redraws in dependent subpictures. In addition, Vogo tries to reuse outdated proxies instead of recreating its entire DOM tree. Other than that, the compiler has a fairly ordinary design. Every update triggers a re-run that "marches the pen". It creates scopes, evaluates expressions, executes state change operations, sets up parameters and so on.

Compared to Recursive Drawing, Vogo's **endless recursion system** is crude. Recursive Drawing progresses the recursive front incrementally in an asynchronous fashion. It also walks into the breadth first, not the depth. This is problematic in Logo because different branches are not independent of one another. Simply opposing a depth limit to the computation does not guarantee a correct execution. This is only the case if the assumption holds that each recursive branch "returns" to the state it started from. This is presumably desirable in endless recursive programs with more then one recursive branch, for example a binary tree. But multiple recursive branches also trigger a combinatorial explosion that is not well met with a static depth limit. For such a case, Vogo has a static execution time limit.

Suffice it to say that many design decisions are simple solutions. However, they are sufficient to demonstrate the principle of **creating a working representation as soon as possible** and provide feedback instantly. As shown in the *Wave* example, each step is meant as an intermediate representation only. Once an idea has solidified, the program can be enhanced to terminate the recursion wilfully. Vogo allows programmers to quickly explore ideas without forcing them to imagine multiple steps ahead without seeing results.

[13]Document Object Model, http://www.w3.org/DOM/

3.18 Summary

Vogo abdicates the use of a code editor by adding a new semantic layer of control on top of the graphics canvas, which turns it from a mere vehicle of output into a means of direct manipulation of Turtle Graphics. A minimal yet flexible set of five central commands was presented: *Move*, *Rotate*, *Loop*, *Branch* and *Call*. I described the metaphors they use, their iconic representations and the way they are organised and controlled spatially. Concrete suggestions were made towards reducing ambiguity and handling abstraction. Those include how the proxy system is used to create a tight coupling with examples, how subpictures foster modularity and encapsulation, how scope and flow are made visible and how expressions and parameters help create dynamism in graphics.

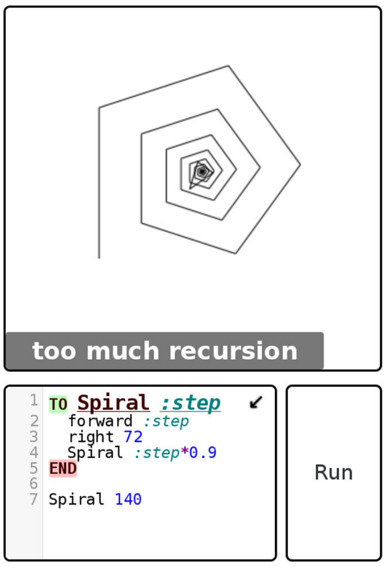
I presented numerous examples at various degrees of detail: arrowhead, spiral, branch icon, binary tree, star, vortex, haus des nikolaus, square, regular polygon, pie chart, wave, heart and grid. This is just a fraction of all the examples that are integrated in Vogo. Its newest version contains an additional icon in the toolbar to load all examples. I commend the interested reader to explore them.

4 Discussion

Vogo demonstrates a novel approach to programming Turtle Graphics. This chapter takes a critical perspective on the proposed interface ideas. How does it fare compared to existing programming environments for dynamic pictures? Does Vogo meet the research goals? Where does it excel, where does it fall short? What are its limitations? What needs to be improved? What new questions does Vogo raise, which remain open?

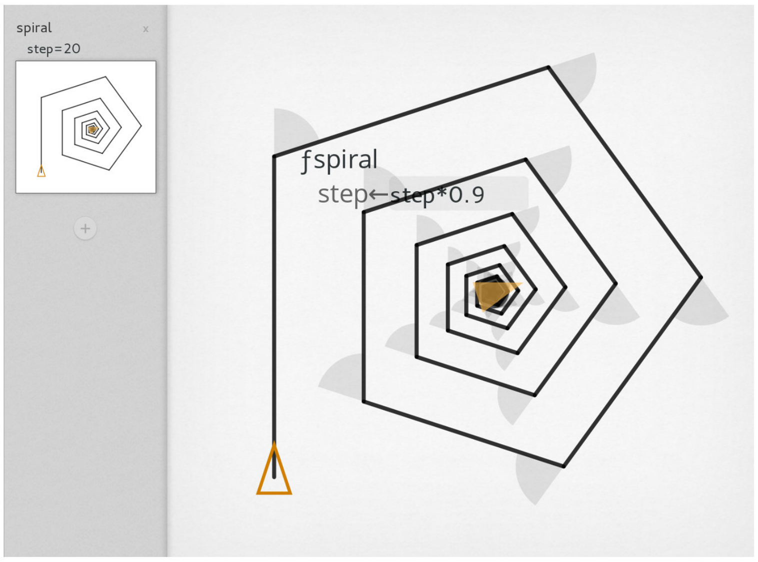
4.1 Comparison with JSLogo

JSLogo is comparable to Vogo due to the following reasons: 1) both are an implementation of the classic Logo: a command language that moves a pen on a plane. Other interpretations and dialects exist, including moving the turtle in three-dimensional space and so called DynaTurtles, multiple turtles that are in constant motion, more akin to a particle system. 2) Both use the structured programming paradigm. The classic flow control mechanisms are used: sequence, repetition, conditionals and calls. 3) Both provide the same kind of basic mechanism for abstraction: parametrisation of encapsulated functionality. 3) Both are a modern, free, web-based interfaces build on HTML5. 4) Both have roughly the same expressive power in terms of shapes they can produce. 5) Both prominently feature a canvas to display the Turtle Graphic. 6) Both rely on the same physical devices for input and output: mouse, keyboard and monitor.



```
too much recursion
```

```
TO Spiral :step
  forward :step
  right 72
  Spiral :step*0.9
END

Spiral 140
```



The figure shows the identical program *spiral* side by side in JSLogo and Vogo. The same example was used in section 2.3. The only minor difference is that *step* is initialised in JSLogo with 140 and with 20 in Vogo. The reason is simply a matter of length unit scale, negligible for the program behaviour. Not visible in the screenshots is the availability of example programs and a command reference in both JSLogo and Vogo.

The most obvious visual similarity is the spiral itself, while the most obvious visual difference between the two interfaces is the **lack of a code editor** and *run* button in Vogo. Those are part of a major **transformation of the way the program is presented and created**.

Vogo generally uses **less text**. Much of the information that the code contains in JSLogo is encoded in Vogo in a spatial way. For example, the parameter *step* in JSLogo is represented *primarily* through the application of syntactic rules: parameters are declared following a function name, separated by a white space and prefixed by a colon wherever they are used. The *secondary* representation is colour. Parameters are purple. It is *secondary* because 1) colour is redundant and 2) not used to specify a property, but only to help display it. For example, a programmer can *not* write a word and then colour it purple in order to indicate that it is meant to be a parameter. This is an important distinction because it means that the programmer can not use colour as a way to create the program, but only as a way to verify syntactic correctness. Without knowing the syntactic rules, it is not possible to create a program. This is not the case in Vogo. Except through the use of wrong expressions, it is **not possible to create syntactically incorrect programs** in Vogo because the interface enforces them. A parameter is primarily represented through its position in the interface: below a subpicture name and in an indented list. A colon is not needed. Vogo does not internally store the program as text and therefore does not need to parse it either. **Text is only used as an export medium** to transmit the dynamic picture over traditional channels. But the exported code is not meant to be altered much like bytecode is in Java; the only difference being that bytecode is a lossy format. Vogo tried to hide the complexities of this low-level format. Commands and functions can be parametrised by a spatial selection and the subsequent pressing of a toolbar button or shortcut alternatively. The interface itself takes care of the correct insertion, setting of a default name and value. The overall amount of required typing is reduced, which can increase speed and accessibility.

Vogo's interface to commands also has a number of downsides. For example, it is **easier to implement a new command in JSLogo compared to Vogo**, because of the way it is represented. JSLogo at best only needs to define a new keyword. All modes of interaction stay the same. On the other hand, Vogo requires the specification of a new spatial representation and interaction method for each new command. Its smooth integration into the existing command structure is not trivial, because a range of dependencies have to be taken into consideration. For example, the new spatial representation is not allowed to collide with the way other commands are represented in order to avoid ambiguity. It is easier to find or define a new word and assign meaning to it than to find a metaphor that conveys the desired meaning and provides a strong way of **referencing existing**

knowledge structures in the programmer. Once found, the second considerable hardship is its **implementation**. Vogo pleads for direct manipulation programming but is itself written in Javascript out of necessity. The argument brought forward was that programmers should not be forced to think about a geometric construction (like the Turtle Graphic) in algebraic terms. The same analogy holds true for Vogo's implementation. Tools and interfaces for textual processing of text are abundant, but not for visual processing of dynamic pictures. The existing tools amplify the dominance of text and the hardships of implementing other forms of representation.

This is also one of the reasons why **JSLogo has a much broader vocabulary than Vogo**. I here neglect expressions for parameter setting, mainly arithmetics and logical operations. Arguably important commands not available in Vogo are:

- *penup & pendown*: disjunct paths
- *pencolor & pensize*: stroke styling
- *setpos & setheading & home*: absolute positioning
- *stop & bye*: breaking the execution
- *while & until*: conditioned iterations
- *fill*: flood fill area with colour
- *make*: creating variables
- *label*: printing text

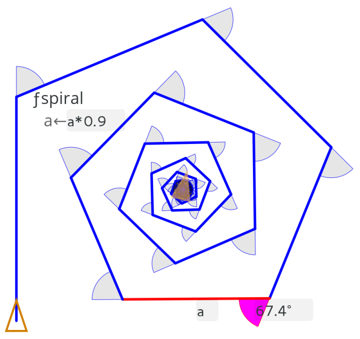
The upside of having a multitude of commands to choose from is an increase in overall flexibility. The downside is a decrease in the flexibility each individual command possesses, which is a consequence of its increased specialisation. This is a trade-off. **Having too many tools is a burden while having too few tools is restrictive**. With only five commands Vogo is very restrictive. However, due to the selection, careful design and the way they work together, their expressiveness is still high.

Another major difference between JSLogo and Vogo is the **way the program is organised and navigated**. In Vogo, functions are listed in the subpictures panel. Only the internals of one function are visible at a time. In the library of JSLogo multiple functions can be visible at once. On the other hand, each function provides a preview in Vogo, which improves its "readability" because it is immediately understandable what each function does. Navigation in JSLogo is done by **scrolling in the code**. The program is organised in a sequential list. In Vogo the program is organised spatially on the canvas. Navigation happens through **panning and zooming**. Zooming provides a smooth transition between overview and detail that is not available in code. However, due to the spatial arrangement of the program, it may be harder to grasp its sequential structure. On the other hand, the advantage of the spatial organisation of commands is its tight integration with the program flow. While it can be hard in JSLogo to find the command that drew a certain part of the picture, this is easy in Vogo because of the spatial proximity between cause and effect. The trade-off lies between the readability of the sequential structure and flow visibility.

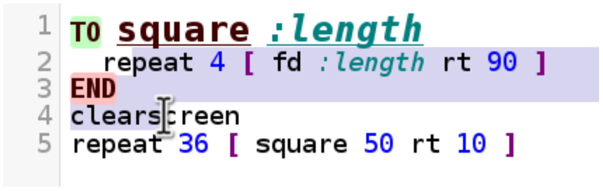
Another kind of proximity that is vastly improved in Vogo is **temporal proximity**. Vogo's *run* is implicit. Changes to the program are visible immediately. For example, changing the number of iterations in a loop is updated at an interactive rate. Furthermore, changes do not have to be discrete jumps, but can be continuous on a customisable level of detail. The dragging of constants supports this kind of manipulation. Even more direct is the dragging of angles and lines because it operates on the spatial representations, is continuous, bi-directional and immediate. The **principle of locality is honoured** and, apart from expressions, **no switch in thinking mentality** is forced on the programmer.

For example, assume a programmer wants to **change the direction of an existing line to be horizontal**. In JSLogo the programmer would have to **1)** switch the visual context from the canvas to the code view, **2)** find the command that adds the angle before the line in question (which presumes understanding at least parts of the program first), **3)** compare the line's current heading with the horizontal heading, **5)** mentally translate the delta into a number of degrees, **6)** determine the right turn direction and add or subtract the number of degrees to the existing angle by **7)** selecting the current number with the cursor and **8)** typing in the new number, **9)** hit the *run* button, **10)** switch back to the canvas and **11)** verify the result. Unless the programmer completely understood the chain of all previous rotations, the solution can only be an approximation. Thus, the verification may identify a required readjustment, which restarts the same procedure from anew.

In Vogo, the very same task can be accomplished as follows: **1)** figure out the flow direction of the line in question by tracing forward from home or backward from the pen, **2)** find its point of origin, **3)** find the angle at that location, **4)** point the cursor to it and **5)** drag it into the horizontal (west or east). Since the program provides immediate feedback, the drag can be released whenever the line is straightened to the horizontal. This too is an approximate solution, but faster and more precise. The programmer had not to think about any numbers, algebraic operations or switch context between graphic and text.



Now consider the **spiral** program at the beginning of this section. Say the programmer wants to change the direction of the *fifth line from home* to be pointed west, to be in the horizontal. In JSLogo, using the described step by step procedure would fail in this example, because the only one rotation *right 72°* in the code does not stand in a direct relation with the heading of the line in question. Instead, since it is called multiple times, the relation is compounded by a factor. To rotate west from home by turning right four times, each turn has to be *270°/4=67.5°*. This is the analytic solution. Apart from *wild guessing*, it is the only workable approach in JSLogo. For the analytic solution a complete understanding of the program is required. In contrast, the approach described for Vogo still works in this example. The only difference is that the drag of the angle does not as beautifully align the position of the mouse with the line in question. However, rotating the cursor still allows a smooth easing in on the approximate solution.



```
TO square :length
  repeat 4 [ fd :length rt 90 ]
END
clearscreen
repeat 36 [ square 50 rt 10 ]
```

Selection works completely different. Vogo uses spatial selection that operates on commands while JSLogo uses textual selection that operates on characters. But characters do not matter. **Textual selection relentlessly cuts through all the semantic properties of the program.** It does not respect words, it does not respect blocks, it does not know about values or operators. Selections in text are not allowed to have gaps, which means that no two positions can be selected at once. Furthermore, editing can only happen in one position at a time. It is practically impossible to edit a program without breaking its semantics during the editing process. None of this is true in Vogo. Selection and editing do respect the semantics of the program. Furthermore, Vogo actively simplifies common tasks and tries to provide reasonable defaults. For example, it is assumed that moves and rotations tend to follow each other in alternation. For a fast construction, Vogo provides a preview for the creation of moves and rotations. Vogo also harmonises the way they work together by letting the angle be pointed into the direction of the subsequent line's end point. In addition, multiple commands can be edited simultaneously.

In Vogo, the classical way of selection and editing only applies to the "manual" manipulation of expressions. For example, this is required for setting conditions in branches. While text manipulation is the only way to edit a program in JSLogo, it is **limited to non-constant expressions** in Vogo. Still, this is a severe downside, because Vogo does not "understand" expressions and can therefore offer no assistance to the programmer in formulating and manipulating them. A second problem is that some expressions may be surprising to novices. For example, the use of the implicit parameter *i* in loops is not obvious. The same is true for array parameters that are queried.length and accessed[i]. Javascript syntax has to be known.

The visibility of blocks is a strength of JSLogo compared to Vogo. Blocks are surrounded by [brackets] and its body is typically indented. It is interesting to note that **indentation is actually an "artificial" spacial organisation inside text**. I say "artificial" because it is, just like colour and highlighting, redundant[1] and not typical in prose. In Vogo, blocks are not immediately visible. They have to be deduced from the flow and scope, which requires previous sampling through selection. This is a downside of Vogo. On the other hand, program flow and scope may be harder to understand in JSLogo. This is particularly true if the program contains many conditioned calls, because each condition alters the flow at run-time and each call creates a new scope inside the current one. Accomplishing the visibility of the program flow, blocks, scope and structure with clarity is hard and involves trade-offs. For a more detailed study I refer to Mike Bostock who wrote about the

[1]Python is a prominent exception - https://www.python.org/

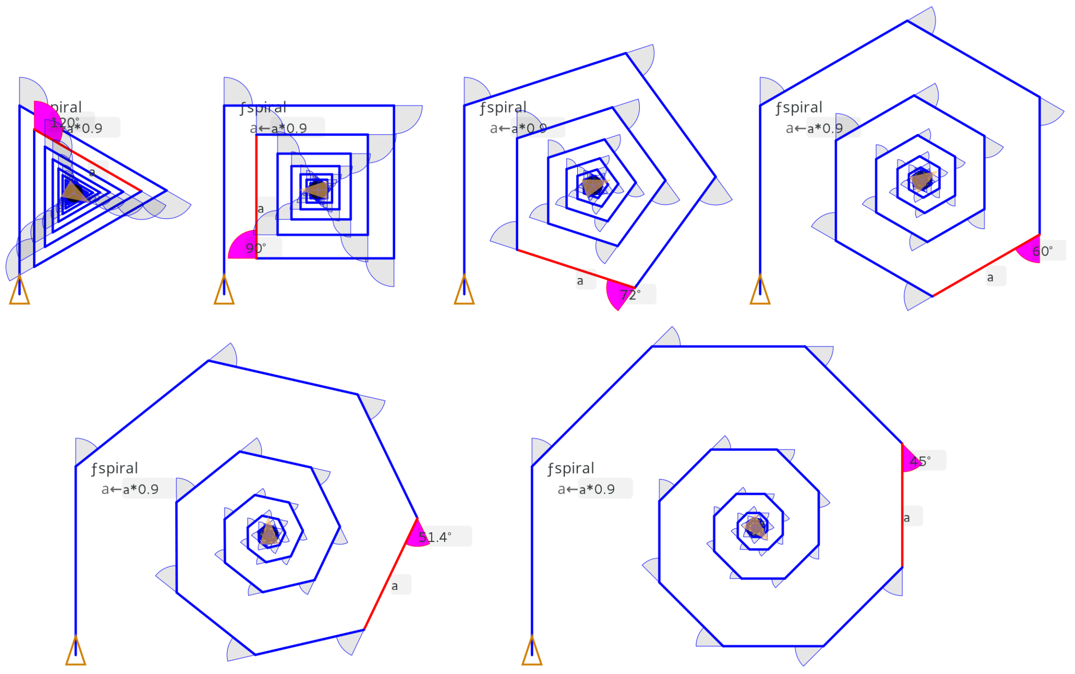
difficulties of algorithm visualisation (Bostock, 2014). Vogo emphasises the visibility of the flow to **make the program behaviour transparent** and easier to understand. This choice has adverse effects on the visibility of other program qualities.

This section compared JSLogo and Vogo. Some significant differences and similarities, advantages and disadvantages were discussed. The main difference lies in the way the program is portrayed and edited. JSLogo relies exclusively on textual editing while Vogo incorporates spatial forms of representation and direct manipulation. The main aspects that were discussed are in summary:

- what metaphors are used and how they reference existing knowledge
- the extend and properties of the provided set of tools
- how orientation and navigation is accomplished
- whether the principle of locality is honoured in both time and space
- the way of thinking that is most prominently employed
- which program qualities are most prominently visible
- how syntactic and semantic support is provided by the environment
- ease of use for novices

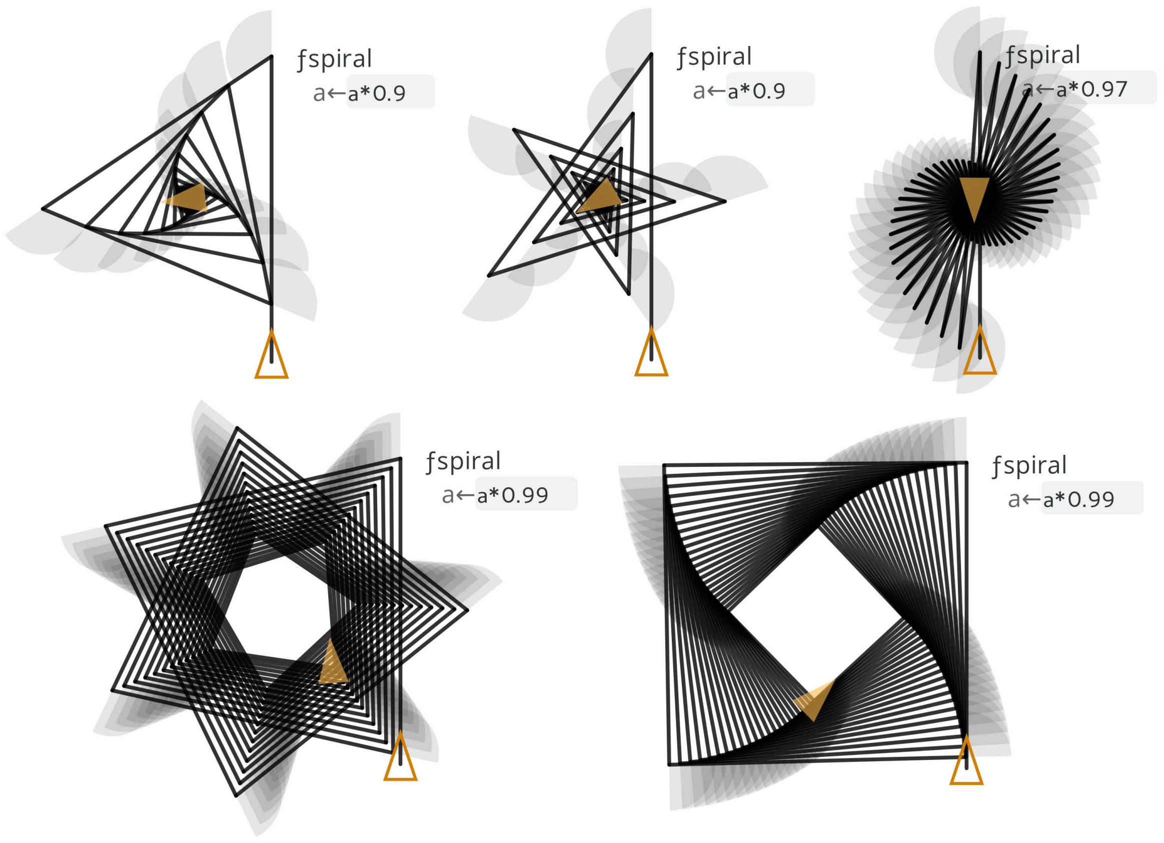
4.2 Exploration

Vogo's interactive drag is an invitation to exploration and curiosity. It is one of the central accomplishments of Vogo. It opens the door to ideas and insights that were previously literally *unthinkable*. The *spiral* shall serve again as a demonstration. Dragging the angle immediately provokes the question: What happens at different angles?



Distinctive patterns emerge when the headings of multiple lines converge. The following angles stand out: 120°, 90°, 72°, 60°, 51.4°, 45°. Do they have something in common? No analytic genius is required to find that the angles must have something to do with the number of edges used for a complete turn: 360°/n. This was found by *inductive reasoning* to explain the findings. It was **not necessary to already know the formula** in order to produce these graphics. But this is exactly the case in JSLogo because it is not possible to easily explore the whole space of possibilities. The best programmers could do is "poke around" by trying different values one at a time. Or they already know the formula and *reason deductively*. But then this is not discovery. The third option is *imagining* how changing the angle would affect the graphic, which is limited by the amount of "blind precomputation" imagination can do. Thoughts beyond this boundary become unthinkable without external aid. *Writing* is a good analogy. **Writing made thought visible.** Writing was a new interface to language that aided the reflection on previously fleeting thoughts (Victor, 2013b). In the same line of argumentation, Richard Hamming wrote:

> Just as there are odors that dogs can smell and we cannot, as well as sounds that dogs can hear and we cannot, so too there are wavelengths of light we cannot see and flavors we cannot taste. Why then, given our brains wired the way they are, does the remark "Perhaps there are thoughts we cannot think," surprise you? (Hamming, 1980)



The above graphics shows more examples of discoveries of interesting patterns that can be found while exploring the effects of the angle and the factor in *spiral: Move a, Rotate r, Call spiral a←a*factor*. Considering its simplicity, the range of shapes it can produce is astonishing. Note that the "holes" in the fourth and fifth example are due to the limited execution depth of recursions. They would be closed otherwise.

Most of these findings were surprising to me. One of the thoughts the first example, the "triangular steps", invoked was: In what relation do angle and factor stand given the constraint that the start and end point of all lines (except for the first *n*) need to touch or "stand on" each other, as demonstrated for *n=3* in the first and *n=4* in last example? Vogo takes a **small step towards enabling these serendipitous discoveries**, which were previously cumbersome to make in Turtle Graphics.

4.3 Programming or Drawing?

Today, *programming* is largely synonymous with *coding*. Coding may be defined as the textual manipulation of symbolic abstractions. Source code is a way of representing a program and coding the act of manipulating it. If programming is identified merely by the style of interaction and the way a program is represented, then neither Vogo nor Recursive Drawing nor Drawing Dynamic Visualisations are programming environments. Judged by these criteria, the mentioned tools are more akin to drawing then to programming. But they do not create static pictures. They create programs. Vogo's export proves that. Equating programming with coding is insufficient. Papert writes:

> Programming a computer means nothing more or less than communicating to it in a language that it and the human user can both "understand". And learning languages is one of the things children do best (Papert, 1980, p. 6).

In the case of Vogo this language is Logo. This is the reason why Vogo was not compared with Logo but with JSLogo. Multiple interfaces to the same language exist. Speech, writing and signing are all interfaces to language. In the same sense, **code is not language and coding is not programming**. Vogo attempts to expand the view of what programming is and who programs.

4.4 Environment Analysis

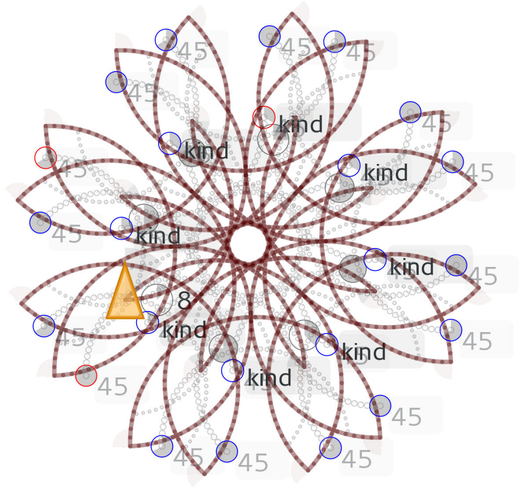
I defined several criteria for the environment as research goals in section 1.4. **Direct manipulation depends on the successful interplay of many interface elements.** For example, instant feedback can not be achieved when the execution performance is slow, nor when the editing granularity is too coarse. Direct manipulation also requires the visibility and tangibility of the object of interest. Arbitrary symbolic representations need to be replaced by iconic, visual representations that resemble the signified and create strong references to appropriate existing mental schemata.

Vogo demonstrates that such an interplay is possible. It implements a substantial subset of Logo. Abstract procedural paths can be created and manipulated in a visual manner. All important programming **tasks** are supported (add, select, delete, copy, paste, adjust, rearrange, insert, abstract, iterate, recurse, branch, call, ...) without relying on code manipulation. Multiple spatial **metaphors** for the direct manipulation of Turtle Graphics were found and presented: drawing with a pen, movement and orientation in space, the geometric angle for rotations, clocks and winding them up for loops, dragging moves to alter their length, subpictures for decomposition, the toolbox for commands, home for the origin of the program, tree branches for conditions, geometric objects like spirals, trees, waves for the program run-time behaviour and the ladder of abstraction for parametrisation.

The environment fulfils the following **criteria** (Victor, 2012a): **1)** the "vocabulary is readable". Meaning is transparent through the use of recognisable spatial representations that are embedded into the context of the object of interest. **2)** The program flow can be seen, retraced and comprehended. **3)** State is either eliminated or visible. There are no variables. The environment implements a functional style of programming. What the "turtle is thinking" is visible throughout its path. **4)** Graphics can be created by reacting. The "parts bucket" is shown and results are instantly visible. Each step to the next intermediate result is kept small. Defaults allow the programmer to start somewhere, then sculpt. **5)** Abstractions are grounded to concrete examples. The environment encourages starting on the ground and climbing the ladder of abstraction in iterative stages.

The most recent conceptual contribution to the research challenge posed in this thesis is **Constructive Visualisation** (Huron et al., 2014). Huron et al. present a new paradigm for the creation of dynamic visualisations which is focused on supporting amateurs to create their own data visualisations. They identify three challenges that a system would have to support: **1)** keep it simple, **2)** enable expressivity and **3)** incorporate data dynamics. Arguably to different degrees, Vogo meets all of them. **Simplicity** does not reach kindergarten play, but is a clear improvement over existing environments like JSLogo and Scratch. Huron et al. propose *sketching* as an activity "routed in deeply familiar activities", similar to Vogo's *drawing*. **Expressivity** is moderately high. For example, no styling of lines and areas is supported yet. However, the versatility of producible shapes is high and exceeds existing direct manipulation environments like Recursive Drawing and Scratchpad. **Dynamics** are fully supported through parametrisation and encapsulation. Even data arrays can be used inside Vogo, though some knowledge is required for their referencing. For more complex data operations, *export* provides the opportunity to design a data graphic in Vogo and then use it in D3.

4.5 Limitations



Vogo is a prototype with many insufficiencies. Some of them were already hinted at throughout the thesis. **Clutter and occlusion** on the canvas are two of the most important. They severely limit the scalability of programs. The figure to the right shows an extreme example. It contains **three nested loops**. To make matters worse, their visuals **interleave one another**. The second loop is selected. It strikingly demonstrates the importance of a reduced opacity and the need to size loop clocks depending on their number of iterations. Nevertheless, it is hard to understand and extend the program. Admittedly, this example does not make any use of encapsulation, which would much improve the situation. One consideration to solve this problem would be the use of something similar to the *steps* panel in Drawing Dynamic Visualisations, which lists the program structure much like code does, but not for editing, but for browsing and selecting only.

Another limitation is that only one branch of conditions is visible at once. This is a symptom of a larger problem: **only one snapshot of the complete functionality of a subpicture is visible** at once, because abstractions are grounded to examples. This principle helps to make the subpicture visible in the first place and abstractions graspable. However, since the program is presented in terms of only one instance, other paths of execution may be hidden. The ability to understand a program in Vogo is therefore highly dependent on illustrative default parameters. Bad examples or naming may seriously impair the ability to grasp parts of the program.

Vogo does not "understand" **non-constant expressions**. It only evaluates them, just like code does, but can not guarantee their syntactic correctness or provide help for their construction and editing. This limits the amount of direct manipulation that Vogo can offer. For example, an expression like *3*x+90°* should allow the programmer to interactively adjust via drag *3*, *x* and *90°* individually. Or if an angle with the expression *x*x* is dragged to *25°*, Vogo should be able to solve the equation and set the default parameter of *x* to *5*. This is not currently possible.

Another limitation to the scale of programs are **performance issues**. They threaten the interactivity of the execution that direct manipulation depends on for providing instant feedback. Again, expression evaluation is one of the major factors that decrease the performance. DOM manipulation, which includes the creation, adjustment, styling and destruction of SVG elements, is another major performance drain. Of course, Javascript itself is slow due to its dynamic, interpreted nature.

Vogo's **vocabulary is restrictively small**. Although the five commands accomplish a wide breadth of expressive possibilities, they do pose limits. For example, paths can not be disconnected, can not be closed to form areas, which can not be filled with colour or otherwise styled. Furthermore, absolute pen positioning is not possible and the properties of the stroke can not be altered. SVG offers a wide range of elements: circles, rectangles, bézier curves, text; geometric operations: resize, rotate, shear; styling: radial gradients, patterns, blur, etc. that are not available in Vogo.

Those are the most significant limitations that Vogo imposes on the direct manipulation of Turtle Graphics. However, the variety of limitations is much broader. For example, subpictures can not themselves be parameters. This would be desirable in a functional language. Another problem is that Vogo does not yet provide an *undo*. Endless recursion has a fixed depth and execution time limit. New commands can not be directly inserted before a scoping element. Two subsequent moves and angles are hard to tell apart. Moves, angles and loops with a parameter of zero are invisible. Cyclic recursion schemes can not be exported. Calling a subpicture lacks affordances. This list continues far beyond the scope of this section. For an exhaustive enumeration I refer to Vogo's documentation.

4.6 Open Questions

One of the leading questions behind the thesis project was: how can the program be manipulated directly on the canvas instead of indirectly through the code? Inspired by the example of Recursive Drawing, this lead to the abolishment of the code editor. But the previous section mentioned the disadvantages of not having a separate view for the dependencies - "structures" - that exist within the program. Showing this structure only in terms of a concrete example may be misguided. **It remains an open question how these dependencies can be effectively represented.** Two directions may provide an answer. The first is to find a way to spatially portrait abstract functionality "exhaustively". The second is to reintroduce a revised equivalent of traditional code. Drawing Dynamic Visualisations provides an example of how this may look like in its *steps* panel, because it only serves as a reference, not as the primary means of manipulation.

The second open question from the same line of insight is: **Given that direct manipulation and instant feedback are desired design goals, how can the program be structured to ease their accomplishment?** Vogo was initially approached from a different point of view: Given the structure of Turtle Graphics, how can direct manipulation and instant feedback be designed *on top of it?* This is the wrong approach if the design is to be user-centered. The crux is not to think of the new in terms of the old. The traditional way to approach the construction of a program may be inappropriate in the first place.

Introducing **constraint-solving and goal-oriented programming** may be better perspectives for future research. For example, given the program *Loop 4 [Move 10 Rotate 10°]* the programmer may just indicate that he wants the pen's end position to be moved home. The program may then automatically offer two possible solutions: *Loop 4 [Move 10 Rotate 90°]* or *Loop 36 [Move 10 Rotate 10]*. In order to solve the ambiguity the programmer may decide to constrain or "fix" the number of iterations. This constraint and the goal may then form a part of the program. The program may afterwards look as follows: *Loop (fix:4) [Move 10 Rotate 10°]; solve: pen is home.* Changing the number of iterations later is dynamically maintained. For example, changing 4 to 5 automatically adapts the angle from 90° to 72°. Sketchpad's constraint system also serves as a demonstration of the principle idea. How can constraints and goals be specified, represented and solved?

How the program **flow and scopes** are to be visualised spatially remains an open question. Vogo provides just one initial idea. There is still much potential for improvements. For example, a time slider may help to better understand the flow. Scopes are currently only visible once a command is selected. This may be improved by a cascaded transparency that depends on the scope depth. Further research is required to find and evaluate suitable visual mappings.

How can occlusion be avoided? Vogo's icons and command representations - "labels" - are currently blindly positioned on the canvas, independent of nearby elements. This may lead to heavy occlusion which can make elements unselectable and even invisible. This is the case in the binary tree example. The left and right recursive call for the two branches overlap entirely. This may be solved by a collision-free positioning. However, since positions are meaningful, because they indicate the pen's position at the time of invocation of a commands, finding such a positioning algorithm may be hard. Labels are currently not resized on zoom. Giving them an absolute size is one initial idea to allow programmers to gradually zoom in on the details of command clusters.

Perhaps one of the most interesting open question is: **Are the design principles that Vogo proposes generalisable to other domains?** I suspect they are - at least partially - if the nature of the object of interest is spatial. However, more research is needed to arrive at definite conclusions.

Vogo is designed to be user-friendly and accessible to non-programmers. Whether this is actually the case, how its **usability** fairs and how it is perceived is an open question. A user study needs to be conducted in order to answer those questions.

The limitations mentioned in the previous section all raise further questions. For example:

- How can conditionals be edited?
- How can subpictures be first-class citizens, higher-level functions?
- Instead of using Vogo inside D3, how can D3 be used inside Vogo?
- How can Vogo's expressivity be increased without crippling direct manipulation?
- *and of course:* how can the direct manipulation presented in Vogo be further improved?

5 Conclusion

Vogo is the first programming environment that allows Turtle Graphics to be directly manipulated on the canvas. Vogo demonstrates that it is possible to **create complex dynamic shapes without the need to code** in the traditional sense. Instead, the construction of the program is approached with the analogy of drawing in mind. Programmers can use their spatial orientation and geometric thinking mentality throughout, instead of being forced to manipulate the graphic through a layer of symbols that requires an algebraic way of thinking. This was not previously possible. Vogo implements a new form of representation of and interaction with Turtle Graphics. Several design guidelines were identified, motivated and successfully applied. They are:

- provide immediate, continuous feedback
- honour the principle of locality
- visualise every step
- couple program structure and behaviour
- ground abstractions to concrete examples
- make state transparent
- encourage creating by reacting
- reference existing knowledge through metaphors
- reduce tools to a minimal yet flexible set

Vogo demonstrates how they play together in order to achieve direct manipulation. One of the central consequences is that it opens the door to **exploration and curiosity**. How dynamic graphics behave can be experienced in a novel way that can lead programmers to a better understanding of their underlying principles and possibly to new insights by enabling serendipitous discoveries.

Vogo presents an opportunity to **redefine what programming is and who programs** by reaching out to novices. New programmers do not think in the programming paradigms that are currently established. Putting these users first in every regard allows us to question traditional ways of thinking about programming. Instead of asking *how new programmers can be made to understand programming*, we should instead ask ourselves: *how can we transform programming into something that is understandable by newcomers?* This is not asking to trivialise programming, but rather to better adapt programming interfaces to the human mind. They need to play to our natural strengths.

Vogo implements the components of a **constructive visualisation** system; it combines simplicity, expressivity and dynamics. Yet each of them is limited, which leaves manifold open questions for future work. The presentation and nature of the structure of the program will need to be rethought. I hinted at the potential of goal-oriented programming and criticised the grounding of abstract functionality to one example. Future programming environments will not only need to "understand" the program in terms of its syntactics but also in terms of its semantics.

Bibliography

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011. ISSN 1077-2626. http://d3js.org/. p. 5

Mike Bostock. Visualizing algorithms. June 2014. http://bost.ocks.org/mike/algorithms/. p. 61

Pavel Boytchev. Logo tree project. Online, March 2014. http://www.elica.net/download/papers/LogoTreeProject.pdf. p. 15, p. 22

Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. Using vision to think. In Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors, *Readings in Information Visualization*, pages 579–581. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-533-9. p. 5

Jason Davies. Animated bézier curves. Online, 2012. http://www.jasondavies.com/animated-bezier/. p. 5

Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005. ISSN 0001-0782. p. 22

Douglas C. Engelbart. Augmenting human intellect: A conceptual framework. *Air Force Office of Scientific Research*, October 1962. http://www.dougengelbart.org/pubs/augment-3906.html. p. 9

John Fiske. *Introduction of Communication Studies*. Routledge, 2 edition, 1990. ISBN 0-415-04672-6. p. 8

Louise P. Flannery, Brian Silverman, Elizabeth R. Kazakoff, Marina Umaschi Bers, Paula Bontá, and Mitchel Resnick. Designing scratchjr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*, IDC '13, pages 1–10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1918-8. p. 23

Michael Gleicher and Andrew Witkin. Drawing with constraints. *The Visual Computer*, 11 (1):39–51, January 1994. ISSN 0178-2789. p. 16

Richard W. Hamming. The unreasonable effectiveness of mathematics. *The American Mathematical Monthly*, 87:81–90, 1980. p. 62

Christopher Michael Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, 2003. http://dspace.mit.edu/handle/1721.1/61549. AAI0805688. p. 54

Brian Harvey. *Berkeley Logo Reference Manual*. The MIT Press, 2 edition, 1997a. http://www.cs.berkeley.edu/ bh/docs/usermanual.pdf. ISBN-13: 978-0262581493. p. 20

Brian Harvey. *Computer Science Logo Style: Advanced techniques*, volume 2. MIT Press, 1997b. http://www.cs.berkeley.edu/ bh/. ISBN: 0-262-58149-3. p. 10

Brian Harvey, Daniel D. Garcia, Tiffany Barnes, Nathaniel Titterton, Omoju Miller, Dan Armendariz, Jon McKinsey, Zachary Machardy, Eugene Lemon, Sean Morris, and Josh Paley. Snap! (build your own blocks). In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 749–749, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2605-6. p. 10, p. 15, p. 22

Samuel Huron, Sheelagh Carpendale, Alice Thudt, Anthony Tang, and Michael Mauerer. Constructive visualization. In *Proceedings of the 2014 Conference on Designing Interactive Systems*, DIS '14, pages 433–442, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2902-6. p. 28, p. 64

Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human Computer Interaction*, 1(4):311–338, December 1985. ISSN 0737-0024. p. 9

Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 318–326, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. p. 15

Alan Kay. Doing with images makes symbols: Communicating with computers. Online, 1987. http://archive.org/details/AlanKeyD1987_2. University Video Communications. p. 19, p. 21

Alan Kay. The real computer revolution hasn't happened yet. *Viewpoints Research Institute*, pages 1–25, June 2007. http://www.vpri.org/pdf/m2007007a_revolution.pdf. p. 5

Paul Lockhart. *A Mathematician's Lament*. 2009. http://mysite.science.uottawa.ca/mnewman/LockhartsLament.pdf. ISBN-13: 978-1-934137-17-8. p. 22

Marshall McLuhan. *Counterblast*. Harvest book. Harcourt, Brace & World, 1969. ISBN: 978-1-58423-063-2. p. 13

Marvin Minsky. *The Society of Mind*. Simon & Schuster, Inc., New York, NY, USA, 1986. ISBN 0-671-60740-5. p. 18

Brad A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 59–66, New York, NY, USA, 1986. ACM. ISBN 0-89791-180-6. p. 11

Donald A. Norman. *The Design Of Everyday Things.* Basic Books, Inc., New York, 2 edition, 1988. ISBN-13: 978-0-465-06710-7. p. 8

Seymour Papert. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., 2 edition, 1980. ISBN 0-465-04627-4. p. 6, p. 7, p. 10, p. 16, p. 17, p. 18, p. 19, p. 22, p. 63

Seymour Papert and Idit Harel. *Constructionism: Research Reports and Essays 1985-1990.* Ablex Publishing Corporation, Massachusetts Institute of Technology. Epistemology & Learning Research Group, 1991. http://www.papert.org/articles/SituatingConstructionism.html. ISBN-13: 978-0893917869. p. 5, p. 18

Jean Piaget. *The Construction of Reality in the Child.* Routledge and Kegan Paul, London, 1955. http://www.marxists.org/reference/subject/philosophy/works/fr/piaget2.htm. ISBN-13: 9780415210003. p. 16, p. 21

Jean Piaget. *To Understand is to Invent: The Future of Education.* Grossman Publishers, New York, 1973. ISBN-13: 978-0670720347. p. 17

Karl R. Popper. *Logik der Forschung.* 1934. ISBN 3-16-148410-X. p. 17

Mitchel Resnick. Lifelong kindergarten. In *Culture of Creativity: Nurturing creative mindsets across cultures*, pages 50–52. LEGO Foundation, 2013. http://www.media.mit.edu/ mres/papers/CulturesCreativityEssay.pdf. p. 18

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11): 60–67, November 2009. ISSN 0001-0782. p. 10, p. 11, p. 15, p. 22

Toby Schachman. Alternative programming interfaces for alternative programmers. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 1–10, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1562-3. http://recursivedrawing.com/. p. 8, p. 9, p. 15, p. 23

Ben Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, August 1983. ISSN 0018-9162. p. 9

Ben Shneiderman. *Leonardo's Laptop: Human Needs and the New Computing Technologies.* MIT Press, Cambridge, MA, USA, 2002. ISBN 0262194767, 9780262194761. p. 9

Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *The ACM Transactions on Computing Education*, 13(4):15:1–15:64, November 2013. ISSN 1946-6226. p. 11

Gerald Jay Sussman. We really don't know how to compute. Strange Loop Conference, Oct 2011. http://www.infoq.com/presentations/We-Really-Dont-Know-How-To-Compute. p. 5

Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, pages 6.329–6.346, New York, NY, USA, 1964. ACM. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf. p. 9, p. 15

Steven L. Tanimoto. A perspective on the evolution of live programming. In *Live Programming (LIVE), 2013 1st International Workshop on*, pages 31–34, May 2013. p. 10, p. 54

Larry Tesler. A personal history of modeless text editing and cut/copy-paste. *interactions*, 19(4):70–75, July 2012. ISSN 1072-5520. p. 9

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 2 edition, 1986. ISBN 0-9613921-0-X. p. 20

Bret Victor. Dynamic pictures motivation. Online, March 2011a. http://worrydream.com/DynamicPicturesMotivation. p. 5

Bret Victor. Up and down the ladder of abstraction. Online, October 2011b. http://worrydream.com/LadderOfAbstraction. p. 12, p. 31

Bret Victor. Explorable explanations. Online, March 2011c. http://worrydream.com/ExplorableExplanations/. p. 5

Bret Victor. Learnable programming - designing a programming system for understanding programs. Online, September 2012a. http://worrydream.com/LearnableProgramming. p. 8, p. 12, p. 64

Bret Victor. Inventing on principle. Online, 2012b. http://vimeo.com/36579366. p. 5

Bret Victor. Drawing dynamic visualisations. Online, May 2013a. http://worrydream.com/DrawingDynamicVisualizationsTalkAddendum. p. 15, p. 25

Bret Victor. Media for thinking the unthinkable. Online, April 2013b. http://worrydream.com/MediaForThinkingTheUnthinkable/. p. 62

Bret Victor. Stop drawing dead fish. Online, 2013c. http://vimeo.com/64895205. p. 5

Fernanda B. Viegas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. Manyeyes: A site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128, November 2007. ISSN 1077-2626. p. 10

E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '97, pages 258–265, New York, NY, USA, 1997. ACM. ISBN 0-89791-802-9.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.