How High is the Ceiling? Applying Core Concepts of Block-based Languages to Extend Programming Environments

Sven Jatzlau, *sven.jatzlau@fau.de*
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Ralf Romeike, *ralf.romeike@fau.de*
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Abstract

Since the emergence of block-based visual programming languages, they have been developed and improved to become increasingly accessible, intuitive, and easy to use. Over the course of this evolution, both uncommon and entirely new language concepts have been introduced, such as the cloning of objects, or the nesting of sprites. This paper provides a selection of core concepts and describes a categorization model. It is proposed that the concepts found in block-based languages are the reason they lend themselves to constructionist learning approaches. To illustrate this point, the fundamental computer science concept of image nesting will be selected, its background and origin explained, and its incarnation in current block-based languages outlined. Following this, the constructionist task of modifying a programming environment will be implemented. Using the aforementioned concept proves that even high-level solutions according to the “high ceiling”-principle can be implemented using the basic concepts of block-based languages.

Keywords

visual programming languages; Snap; Scratch; GP; block-based languages; language concepts; nesting; composite objects

Context

Introduction to block-based languages

In 2006, the programming language and environment Scratch made its debut. Developed by the Lifelong Kindergarten Group at MIT, it represented a new approach to introducing programming to learners: based on the ideas of “low floor, wide walls, high ceilings” by Seymour Papert (Resnick, et al., 2009), its popularity grew rapidly. At the time of writing, some 32 million projects, guides, animations, and games have been shared by a continuously growing user base of 28 million users. In schools, block-based languages display a similar success: Compared to text programs, block-based environments enable students to achieve better results on average (Weintrop & Wilensky, 2015), and while students like to shift to text sooner or later, they can nevertheless appreciate that block-based programming is equal to text-based programming (Bau, Bau, Dawson, & Pickens, 2015). Block-based languages are characterized by a number of common features, such as a stage, and code blocks that snap together to create scripts and replace the syntax of text-based languages. These features open up new possibilities and approaches to solving problems. In this paper, we give examples for these core concepts, and elaborate upon the origins of the image nesting concept in particular. Within this context, the proposition that block-based languages enable users to solve problems in new ways will be illustrated with an example: extending a programming environment, which may seem to be one of the most complicated tasks users could undertake. However, given the possibilities of the core concepts found in block-based languages, how complicated will it ultimately be?

The example reinforces the claim that block-based languages such as Scratch, *Snap!*, or *GP*, support constructionist ideas, and demonstrates the complexity of projects users can create with them. For this purpose, the GP environment itself will be modified and extended (using its easily-accessible source

code) to include a feature that makes it more meaningful to us: the ability to search for certain blocks within the program code.

Constructionism in programming

The ideas of constructionism and block-based languages have always been closely linked: the programming language Scratch was built on the constructionist principles of *Logo* and *Etoys* (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). The three basic principles “low floor”, “wide walls” and “high ceilings” laid the foundation for the way Scratch and its family members behave and are displayed to the user (Resnick, et al., 2009).

However, whether Scratch supports the principle of “high ceilings” (high threshold for more advanced programmers (Weintrop & Holbert, 2017)) is a point of frequent discussion: feedback shows that learners and educators alike feel as though Scratch is too simple to enable complex, high-level projects. For this reason, GP was designed with the idea of enabling users to create significantly more complex solutions (Maloney, 2018).

Core concepts in block-based languages

Block-based languages, and the environments used to program with them, enable new solutions to known problems, while creating the opportunity to produce entirely new tasks and problems. The reason for this is that in multiple ways, these languages are fundamentally different from typical text-based languages and environments used for teaching programming novices.

To understand the ways in which these languages are different, it is important to recognize the aspects that constitute a programming language. To characterize these aspects, the term “core concept” will be used; these core concepts embody the nature of programming environments and languages and make them both graspable. They should also be central for the way programming languages are taught. The concepts found in these languages separate them from text-based languages used for introductory teaching situations. In the following, several examples of new and promising, or less well-known concepts will be provided and outlined. These examples were identified inductively by analyzing user and developer-made demo projects, and deductively from the programming environments of Scratch, Snap!, and GP.

- **Manual control of program flow** (abbreviated as “Buttons”)
  Most block-based languages enable the user to start, to stop, and (in some cases) even to pause the currently-loaded program. This results in a feeling of directness for the user, making it possible for them to stop and observe their program, which offers interesting strategies for debugging.
- **Broadcasting**
  Sending messages between objects is based on the *one-to-many* communication model, i.e. a single object sends messages to all other objects in a program. Other objects can then react to the received message – or not. This type of messaging is commonly used for synchronization and signaling purposes, as it ties into the event-driven program paradigm (outlined below).
- **Sensing**
  In many block-based languages, objects (or “sprites”) are able to detect a number of different factors, including their own position on the stage, the direction they are facing, whether they are overlapping with any other objects on the stage, or whether a color is touching another color.
  This sensing ability makes objects able to react autonomously to different situations, alignments, or conditions.
- **Prototyping**
  Instead of conceptually abstract classes, many block-based languages utilize a system of prototypes and clones of prototypes in order to create new objects. The concept of prototyping can be further extended to include prototypical inheritance or “delegation” (Lieberman, 1986), a form of inheritance that does not require classes. For this reason,

prototyping and delegation can be perceived as more intuitive and easy to understand than other forms of inheritance.

Event-driven programming

If an event occurs, the corresponding script is executed. If multiple events occur simultaneously, or if multiple scripts react to the same event, multiple blocks are executed at the same time. This behavior results in an implicit type of concurrency that is intuitive and simple for learners to grasp. Events can originate from the user (i.e. key presses, or mouse clicks), or from within the environment (i.e. an object sensing overlap with another object, or messages being received).

Weak typing

One variable can hold different types of data; the same variable can store strings of letters, boolean values, or integers without being limited to a single type of data that is set at the time of its creation.

Stage

Most block-based programming languages feature a stage that can visualize the world of the project to the user, showing objects, and their (inter-)actions. Possible applications include using it as a canvas for drawing graphs, or as a playing field for various games.

Delayed execution of code

Typically, block-based languages slow down the execution of blocks within scripts. This is done for visualization purposes: without an inherent delay, objects could fly off the stage instantly without the user being able to observe the process. The concept is based on a non-atomic interpreter, and is typically found in block-based programming environments for didactical reasons.

Drag & drop (abbreviated as “D&d”)

While most of these languages support keyboard editing and input, learners will typically utilize the concept of dragging-and-dropping blocks from the sprite palette into the scripting area to create scripts. The same process also deletes scripts, moves objects on the stage, and enables the user to resize parts of the graphical interface to fit their personal preferences.

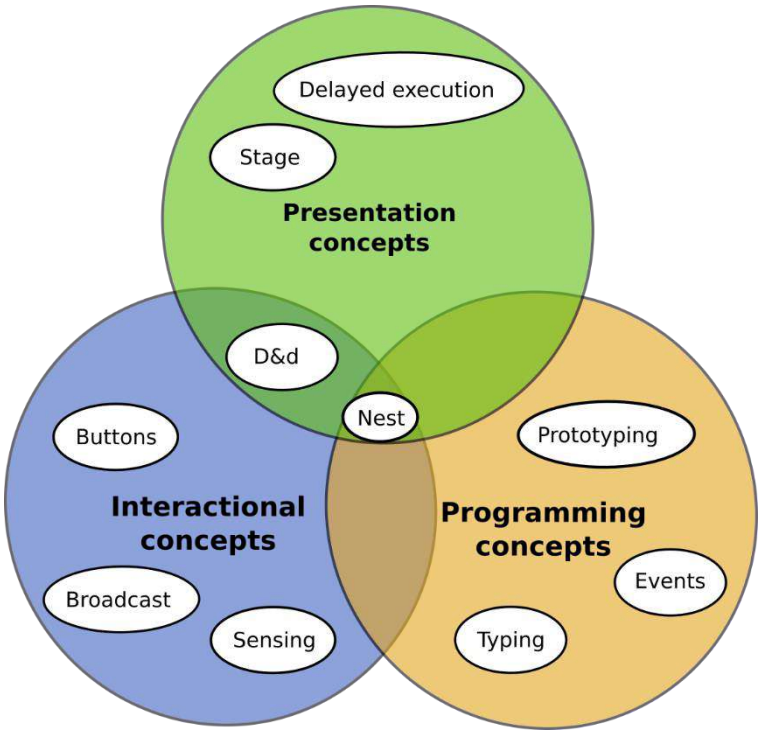
Image nesting (abbreviated as “Nest”)

The attachment of visual objects to other objects can be compared to the way objects are grouped in various software tools (such as Microsoft Powerpoint or OpenOffice Impress). Objects become parts of other, greater objects, while still retaining their individual identity. As a real-life example, a hand can be considered an attachment to the arm, of which it is a part. When the arm moves and rotates, so does the hand. However, both the hand can also still rotate and move (somewhat) independently. Therefore, while the hand is part of the greater object “arm”, it retains its individual nature, and can perform actions independently (such as rotation, and grabbing).

Note: this concept does **not** refer to the idea of nested loops, or code nesting through the use of syntactic elements such as brackets; it deals with the visual representation of objects and their composition into greater composite objects.

This is merely a non-exhaustive selection of core concepts found in block-based languages. As they make up the foundation of how programming languages function, their impact on teaching needs to be considered when designing tasks. These concepts are also the reason block-based languages lend themselves to constructionist solutions and tasks. It is important to note that not all of them are entirely new ideas; many (such as weak typing) have been part of programming languages and environments for decades.

This paper proposes that the core concepts of block-based languages characterize the way they are used and, in the same way, characterize the way they should be taught.



Concepts can be categorized into three overlapping categories:

Figure 1: Categorization model for core concepts found in block-based languages

1. Presentation concepts

"**What** do users see and **how**?"

This category includes all concepts regarding the presentation of the user interface. These design choices were made in order to support the learner's ability to interact with the environment, and to improve their understanding of the underlying programming concepts. The stage, for example, helps learners understand the nature of objects by offering a visual representation in the form of sprites.

While traditional environments make them seem abstract, or "ethereal" to many students, this visualization makes them more graspable and concrete. A common trait of most block-based languages is that for pedagogic reasons, several blocks (such as loops, all motion blocks and broadcasts) have an inherent delay associated with their execution. This delay was implemented to slow down program flow and further support its visualization.

2. Interactional concepts

"**How** do users interact with the environment?"

"**How** do objects interact with each other?"

The second category includes concepts concerned with the way users interact with the programming environment and its user interface, while also including concepts that deal with the way objects interact with each other.
An example within this category is the buttons a user can use to start, pause, and stop the program at any desired point. They enable the user to get a better feel for the program flow. The interaction of objects with each other includes concepts such as broadcasting, and sensing.

3. Programming concepts

"**How** do users implement their solutions?"

This category comprises both typical programming constructs (such as variables, loops and conditionals) and consciously-designed language traits that were implemented with accessibility, intuitiveness and ease of use in mind. For instance, an event-driven program flow (found in all members of the Scratch-family) enables users to implement projects that rely heavily on concurrency – intuitively, while encountering fewer of the typical problems and difficulties associated with concurrency (such as visibility, race conditions, etc.).

Certain concepts do not fit a single category. Dragging and dropping blocks to create scripts is a hybrid concept: while it fundamentally influences the way users interact with the programming environment to create meaningful projects, the possibility of doing so is suggested through the shape, shading, and highlighting of blocks. As another example, image nesting fits all three categories: presentation (due to the visual connection of nested sprites/objects), programming (after nesting, objects can refer to each other as "owner" and "part"), and interactional (as nesting creates a hierarchically-structured part-whole-relation between two objects).

The proposition is that these core concepts enable users to implement constructionist solutions easily – they are the reason block-based languages support constructionism to the extent they do. To illustrate this proposition, the concept of image nesting will be elaborated upon in more detail.

Image nesting: a core concept

The concept of nesting visual objects into others is based on the idea of *composition.* Naturally, the composition of individual, smaller parts to form a greater entity is by no means a novel concept; In the process of identifying fundamental ideas of computer science, Schwill deemed the idea of *structured dissection* a master idea (Schwill, 1994). Two of its sub-ideas are *hierarchization*, and the visualization as a *tree*. All these factors are closely related to the concept of nesting, leading to the conclusion that image nesting is an essential idea of computer science.

Despite its basic nature, nesting has the potential to enable very high-level solutions to complex problems ("high ceilings"), making it a core concept in block-based languages. As has been shown (Resnick, et al., 2009), block-based languages such as Scratch make it simple for users to create projects within the framework of constructionism. Commonly-identified traits of such frameworks are "low floors", "wide walls", and "high ceilings". For Scratch, these three traits have been expanded to include "more tinkerable", "more meaningful", and "more social" (Resnick, et al., 2009).

How do these traits relate to image nesting, and other core concepts of block-based languages? Using the core concept of image nesting, a seemingly very complex, high-level task can be achieved: the programming environment GP will be modified to include a search feature.

This search feature, and the algorithm that is used to implement it, rely entirely on image nesting, therefore showcasing that block-based languages accommodate the ideas of *high ceilings*, *more tinkerable* and *more meaningful*.

In the following chapter, to gain a more substantial understanding of the concept in question, the origin of image nesting will be analyzed in more detail.

The origin of image nesting: Morphic

Morphic is a user interface construction environment. Originally developed as part of the language *Self*, it would eventually play a central role in the Scratch-language family: the first versions of Scratch used Morphic for the user interface creation, which may be the reason for the existence of nested sprites in Snap! and GP.

The first central principle that acts as the foundation for the entirety of Morphic's functionalities is the *morph* (Greek for "shape"). Every visual object is graphically represented by a morph (Bouraqadi & Stinckwich, 2007), which is the base class of everything that can be displayed in a "world". As graphical entities, morphs are able show behavior and handle events: they can sense and react to user input, periodically perform actions, detect their own position and size, and sense overlap with other morphs. The second major concept found in Morphic is "composition". Any morph can be made into a composite

morph through the attachment of others. The attached morphs are then considered submorphs, which keep a pointer to their owner they are part of. When the composite morph moves, turns, is copied, or deleted, the same action is applied to all its submorphs – and their submorphs recursively. While they depend on their owner in many ways, they retain their individuality: they are given a chance to handle events (such as button presses) before their owner does (Maloney & Smith, 1995).

The entire IDE of Morphic is based on this special structural relationship between morphs. As each morph has an owner (be part of a greater composite morph) or can have submorphs (be a composite morph), a compositional hierarchy is created. In this tree-like structure of part-whole-relations, the "world" is at the root, ultimately containing all the visible morphs within itself (Maloney & Smith, 1995).

How does Morphic relate to block-based languages?

Nesting in Snap! and GP is based on Morphic's structure of composition, owner-morphs and parts. Comparing both systems yields the following central aspects:

- Nested objects behave the same way composite morphs do (act as a single entity when performing tasks).
- Individual parts know their owner, owners know all their parts; both can communicate with each other using these connections.
- Individual parts have a degree of independence from their owner (can perform actions and handle events with a higher priority than their owner).
- Both systems create a part-whole hierarchy between objects, and can therefore be visualized by a tree structure

Due to these similarities, it appears that composition in Morphic was used as the foundation of the way nested objects are handled in certain block-based languages; therefore, nested sprites are a new incarnation of composite morphs. The central ideas of composite morphs and image nesting are vital to understanding of how nesting works and what is meant by it.

GP particularly adheres to Morphic's design philosophy. Each part of the user interface is a morph attached to an owner: the script editor, as an example, is a composite morph.

Its components are the individual scripts, which are composite morphs themselves. They are comprised of the individual blocks. Blocks consist of several parts: a colored frame, a text label, and typically one or more input slots. Blocks react to mouse clicks by running their associated code (the so-called "handler"). It becomes apparent that the hierarchical structure and functionality of morphs is part of *GP*'s design.

In the following chapter, these central aspects of image nesting will be utilized to add a search feature to the programming environment GP. The implementation of this feature will demonstrate how image nesting, and other core concepts found in these languages, enable new and intuitive solutions to previously complex problems.

How do concepts enable new solutions?

Block-based languages such as Scratch, Snap!, or GP, enable new solutions to problems due to the concepts they use. The very same concepts are also the reason block-based languages lend themselves to constructionist learning approaches. To give an example of this proposal, the programming environment of *GP* itself will be modified by utilizing its application of the nesting-concept. The implementation will highlight that image nesting enables three aspects in particular: "high ceilings", "more tinkerable", and "more meaningful".

To illustrate this point, we will solve a two-fold problem outlined by the developers of GP: 1) Code represented by blocks typically takes up more space than it would with a text-based representation. This makes it harder to get an overview of the code at a glance (Mönig, Ohshima, & Maloney, 2015).

2) The visual stimuli, such as colors, borders and other graphical elements of blocks, can make it harder for the user to scan the actual labels of the blocks. These label texts, however, hold most of the meaning.

Text-based language editors typically solve this issue by providing a search function that can locate certain code snippets based on a given search term. For block-based language environments, such a feature does not yet exist – even if the user knows the block they need to find in the program, they must visually scan the entire scripting area to find it (Note: most block-based environments provide a search bar for the blocks palette. This searches the library of available blocks based on a search term; it does **not** search the code blocks in the scripting area).

The proposed solution to this problem, therefore, is the addition of a “search in class”-function that enables the user to find a block containing a given search term within the currently selected class. This should operate similarly to the “search”-feature of a text-based language editor, scanning the code contained in a project for a matching search term and pointing the user to its location.

Implementation of a search function in *GP*

In this step, the programming environment of GP itself will be extended to include a “search in class”-function. But what is the role of image nesting for this extension?

Image nesting as a core concept of several block-based languages applies not only to objects on the stage (so-called “sprites”): it can also be found within the graphical interface of GP itself. It is one and the same core concept – a hierarchical part-whole relationship between two objects.

Due to *GP*’s graphical interface structure relying on image nesting, searching for a specific block label only needs to follow a basic algorithm.

However, there are several terms and their relation to each other that need to be clarified (Figure 2):

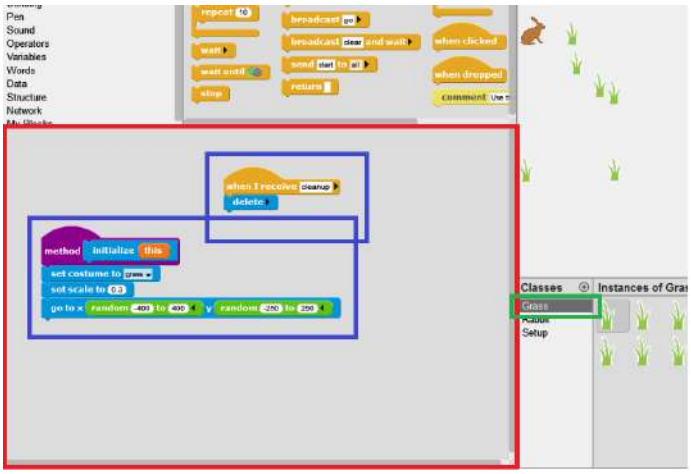


Figure 2: Interface of GP

ScriptEditor (red box): The area that contains all the code of a class. The ScriptEditor is a composite morph; its parts are the individual scripts (blue box).

Class (green box): GP uses classes and objects. The ScriptEditor always displays the code blocks for the currently selected class; this code is shared among all its objects (or “instances”). The currently selected class for the example in Figure 2 is “Grass”.

Scripts (blue boxes): Any number of blocks (>= 1) attached to each other form a script. Every script is a composite morph; among its parts are the individual blocks. The selected class has **two** scripts: “when I receive”, and a method definition for the “initialize”-block.

Blocks: Parts of a script, internally attached by nesting. The “when I receive”-script has **one** block attached.

labelText: A part of a block, contains the actual readable text on the block. The labelText of the “delete”-block is the text string ‘delete’.

The hierarchical structure of the graphical interface that is created by image nesting can be visualized by a tree (Figure 3). This tree shows the position of the label texts in the hierarchy of the graphical interface: the texts are nested into the individual blocks, which in turn are nested into the scripts. Finally, the scripts are nested into the ScriptEditor, which is the part of the GP interface that holds all the scripts of a class.

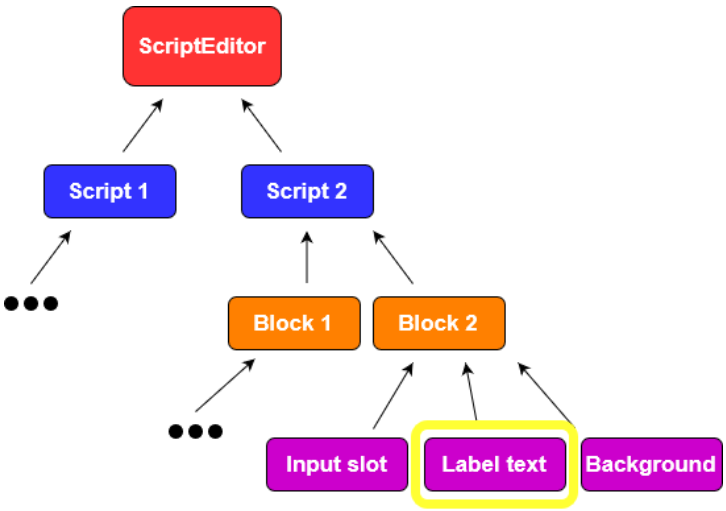


Figure 3: Hierarchical structure of the interface

The algorithm below (simplified) is outlined in pseudo code. This representation was chosen due to the inevitable simplification of the algorithm in place, although a block-based representation may have been more fitting.

To find the matching string, we need to compare the text labels of all visible blocks. To this end, we utilize the concept of image nesting. We will not, however, nest objects into other objects ourselves. Instead, we will utilize what happens when code blocks are snapped together: they are nested into one another.

This impacts our approach to a solution significantly: we merely need to descend the hierarchical structure (Figure 3) that makes up the graphical interface in order to find the label texts of blocks.

```
1 set answer to (ask “Enter search term”)
2 set searchTerm to answer
3
4 for every script of ScriptEditor
5       for every block of scripts
6             for every part of block
7                   if (searchTerm == part)
8                         return true
```

After the user’s search term has been stored (lines 1-2), the concept of nesting becomes important.

Descending the hierarchy of the graphical interface is done in lines 4, 5, and 6. In each of these lines, the composite object (ScriptEditor, scripts, and block, respectively) is prompted for a list of all its parts, that means other objects that have been nested into it.

This hierarchy is followed downwards until the label text becomes available (line 7). The algorithm is therefore no more complicated than parsing a tree and its nodes.

As an example, breaking down the “delete”-block yields three parts – a blue puzzle-shaped image, the label text “delete”, and an image of a black arrow. In this manner, every block of every script shown in the ScriptEditor is decomposed into its nested parts in order to find the label text.

This label text can then be compared to the *searchTerm* entered by the user. If both strings match, the result has been found. The search menu is accessed through the context menu by right-clicking the scripting area. After a matching block label has been found, the ScriptEditor scrolls to it and a highlight is added to signal its position.

Discussion

The reason for the simplicity of the algorithm described here is the architecture of the user interface: as each visible object (like the ScriptEditor, the scripts, the blocks, etc.) is a part of a larger (composite) object, and is in turn made up of parts (making it a composite object), a hierarchical structure is created. Descending this hierarchy downward to the point where the label text of blocks becomes accessible makes it possible to keep the search algorithm simple and intuitive. Therefore, the search algorithm makes use of the **nesting** of images that are used to make up the user interface of GP. Furthermore, it also makes use of three central ideas of constructionism in block-based languages:

- **High ceilings:** Image nesting enables simple solutions to complex problems: An example could be implementing an autonomous car: the task is to create a car that is able to follow a race course reliably without user input; the issue is that to turn appropriately, the car needs to be able to determine where exactly it touches the edge of the course. In block-based languages, this can be implemented simply by nesting sensor objects onto the car; these sensors notify their owner if they touch the edge of the race course, to which the owner reacts by turning according to the notifying sensor.
  At the same time, however, nesting also enables solutions on a much higher level; as described in this paper, the seemingly complex task of modifying a programming environment can be done by utilizing its reliance on image nesting.
- **More tinkerable:** As GP is open to modifications, the search algorithm described above can be implemented entirely in blocks through the usage of GP's system class browser.
- **More meaningful:** By extending the programming environment of GP according to our individual needs, we modified it to become meaningful for us (Resnick, 1996).

The search feature is by no means indicative of the limits to image nesting: with some adjustments and extensions, this feature could also be used to parse blocks in a project for various purposes, for example, an analysis tool examining the block types users tend to use; the "high ceiling" has not yet been hit. Furthermore, while the language of this implementation is specific to GP, the algorithm itself is not; the algorithm is based on the usage of image nesting. Therefore, similar implementations are possible in other programming environments that utilize nesting to create their graphical interface, such as Snap!.

Conclusion

As the example illustrates, the graphical representation of elements in block-based languages enables the reduction of algorithms to their basic principles. For the user, this means that the implementation of personally-meaningful, constructionist projects is simplified. In the case of image nesting, which originated from Morphic's composite morphs, the unique part-whole hierarchy it creates between objects makes it possible to solve a wide range of different tasks. Among these tasks are typical programming projects implemented by novices in introductory courses, such as autonomous cars. Even on a higher level ("high ceilings"), however, block-based languages enable constructionist solutions to known problems, as has been demonstrated with an extension of the GP programming environment. In conclusion to the posed question, "how high is the ceiling?": we have not hit the ceiling quite yet. With even basic concepts such as image nesting, high-level solutions to problems can be implemented in a simple manner. Indeed, the particular feature implemented in this paper may be part of future GP versions.

References

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. S. (2015). Pencil Code: Block Code for a Text World. Proceedings of the 14th International Conference on Interaction Design and Children, 445-448.