Rethinking Children's Programming with Contextual Signs

Ylva Fernaeus
Computer Science Department
Stockholm University
SE-164 40 Kista, Sweden
ylva@dsv.su.se

Mikael Kindborg
Computer Science Department
Linköping University
SE-58183 Linköping, Sweden
mikki@ida.liu.se

Robert Scholz
Computer Science Department
Linköping University,
SE-58183 Linköping, Sweden
robsc@ida.liu.se

ABSTRACT
We present an approach to children's programming inspired by the semiotics of comics. The idea is to build computer programs in a direct and concrete way by using a class of signs that we call contextual signs. There are two aspects that distinguish contextual signs from other sign systems used for programming. The first is that the signs are displayed in the immediate visual context of the object that they refer to. The second is that the signs are used to illustrate actions and properties in a way that is directly perceivable by the user. We argue that these two properties make contextual signs a promising high-level approach for building systems that are rich in dynamic properties, such as the ones that children often like to build.

Keywords
Visual programming, tangible programming, children, comics, semiotics.

ACM Classification Keywords
D.1.7 Visual Programming, K.3 Computers and Education.

INTRODUCTION
There are two basic motivations for designing programming tools for children, to "make children smarter" and to empower children to create interactive simulations, toys, and games. The work we present aims for the latter goal, and we will focus on high-level programming with behaviours, rather than low-level algorithmic programming. By computer programming we refer to the activity of developing working computational systems. In the case of children's programming, the systems aimed at are often rich in dynamic and interactive properties, similar to games and software children use in their everyday activities on the computer. To build such software requires the user to learn some form of notation that can be interpreted by the computer, using a programming tool. Since programming is known to be complex, much research has concerned alternative ways of representing computational systems [7].

Studying comics is relevant to programming, because like programs, comics depict dynamic activities using a static representation. For readers that have learned the sign language of comics, the visual presentation can produce a very direct reading experience, creating an illusion of for example motion and sound, even though the medium itself is static and silent (see Figure 1). At the same time, a static representation provides an overview and can be edited in a straightforward way, which is important to programming. A basic sign for expressing dynamics in comics is a sequence of panels [2,10]. Previous work has shown that comic strips can be used for programming [9]. However, in this paper we discuss a second approach to using comics for programming, based on the concept of *contextual signs.*

Even though visual symbols and iconic representations are core to all visual programming tools and languages, they are usually not displayed in the same manner as in comics. Contextual signs are used inside the panels in a comic strip, to communicate to the reader what is not possible to express with an ordinary static picture [10]. Examples include motion markers (e.g. speed lines and ghost images), voice balloons, and sound imitating words (onomatopoetic symbols), see Figure 1. The directness and dynamics that contextual signs bring to the characters in comics could also be used for visual programming.

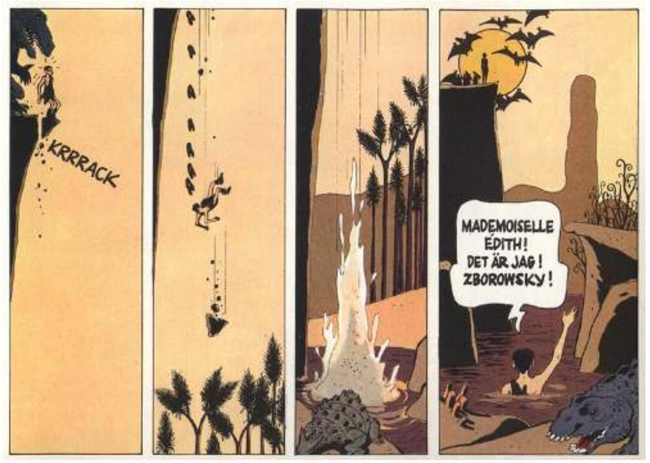


Figure 1. Contextual signs in comics. The sound symbol "KRRRACK" in the first panel contributes to the impression of the rock cracking loose. In the second panel, the motion lines and the sound symbol "A A A A AAA", transforms a static picture of a falling person into a dynamic impression of falling. The voice balloon in the last panel creates the impression of a person speaking. (From page 22 of "Adele och odjuret", original title "Adèle et la bête", by Jacques Tardi. Published in 1979 by Carlsen if.)

IDC '06, June 7-9, 2006 Tampere, Finland
Copyright 2006 ACM 1-59593-316-6/06/07... $5.00

To elaborate on the values of contextual signs in children's programming, we will start by discussing an example of how two children have described the functionality of a dynamic computer game when sketching it out on paper. Thereafter follows a description of what we mean by contextual signs in programming, illustrated by three prototype systems. We end by discussing design aspects related to programming with such signs.

THE USE OF COMICS IN CHILDREN'S GAME DESIGN

In our previous research, we have involved children aged 8-12 in making their own computer games [3,4,14], using existing programming tools such as ToonTalk [6]. Figure 2 shows a sketch of a game created by two girls in sixth grade who participated in one such project. In this study the whole class was involved in weekly sessions of building computer games in school, and as part of the game design the children often made drawings on paper. We selected this particular example because it contains several important features that we find common in sketches that children make when expressing game ideas on paper. What we would like to call attention to is the rich visual language that the children use, and in particular how this relates to conventions used in comics.

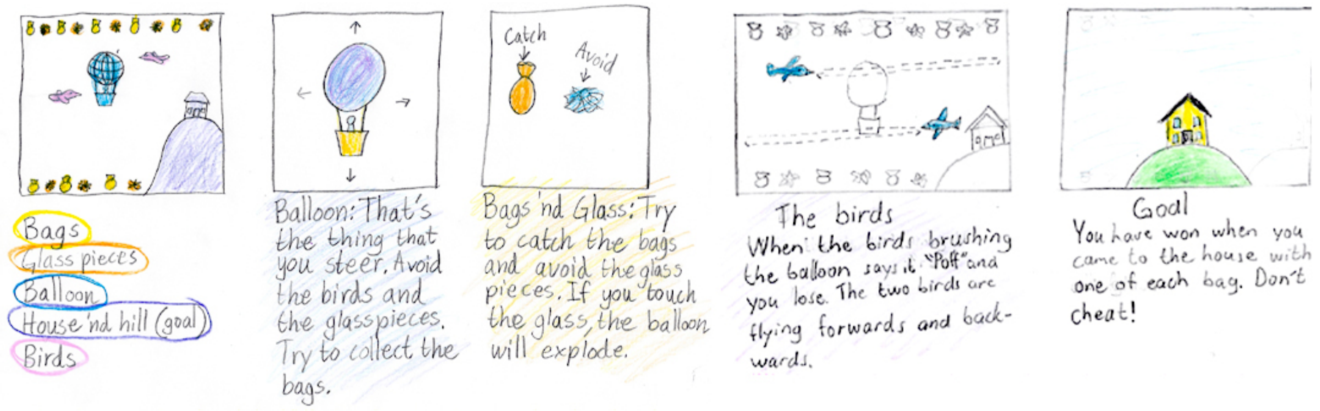
The design sketch is based around a *sequence of panels*, making it possible to read almost as if it was a comic strip. Pictures of the active objects in the game dominate the visual representation, and the names of the objects are listed below the first panel. This is also typical for comics, where pictures of domain objects (e.g. characters featured in the story) are the most central elements of the presentation.

To further point to the dynamic properties of the game, the girls have made use of contextual signs, such as arrows and labels. The description has been augmented by using textual explanations of how the different objects should animate, as well as instructions for use. All these elements in combination provide an elaborate image of how the proposed system should work, especially how the girls want their game to appear to the player. To the reader, it is clear that the basic idea of the game is to fly and control a hot air balloon, and that the objective is to collect bags and go to a house, and at the same time avoid birds and dangerous pieces of glass that will destroy the balloon.

There are two aspects of this sketch that we find especially relevant to visual programming. The first regards the visual layout of elements within each panel. The dotted lines in the fourth panel do for instance give a quite vivid image of how the birds are meant to be moving in the game. An important part in achieving this effect is the layout of the symbols in the context of the game objects. Related to this is also how the children combine textual and visual descriptions. Much effort in the development of visual programming tools has focused upon representations that are entirely "text free". However, despite the visually rich language in the sketch, textual descriptions and labels seem to serve an important role here.

The second aspect concerns what parts of the dynamic actions that the girls have chosen to account for in their illustration. Notably, the panels do not express rule-based "if…then..." programming constructs, which are used in some programming tools (see e.g. [8] and [13]). On the contrary, each panel focuses on different aspects of the behaviour of the objects in the game. Many children's programming tools are based on a paradigm of programming that is oriented towards algorithmic structures and conditional rules. The girls in this example however seem to be primarily focusing on the behaviour of the objects and how the game should be experienced by the player, rather than on how it should work on a more technical level. The sketch does for instance *not reveal exactly* how the balloon should be steered, e.g. whether it should have a constant speed or if it should move in a stepwise manner. In this sense, even though the description involves a lot of detail, the concepts used are on a higher level than is supported by most programming tools.

If programming is to be conceptualized in a style similar to game designs like the one in Figure 2, the basic primitives of "programming" may have to take on a different form.



Bags
Glass pieces
Balloon
House 'nd hill (goal)
Birds
Balloon: That's the thing that you steer. Avoid the birds and the glasspieces. Try to collect the bags.
Catch
Avoid
Bags 'nd Glass: Try to catch the bags and avoid the glass pieces. If you touch the glass, the balloon will explode.
The birds
When the birds brushing the balloon says it "Poff" and you lose. The two birds are flying forwards and back-wards.
Goal
You have won when you came to the house with one of each bag. Don't cheat!

Figure 2. Game designed on paper by two girls in 6th grade.

PROGRAMMING WITH CONTEXTUAL SIGNS

Comic book artists use contextual signs to communicate features and actions of characters and objects that can not be expressed using a plain static image. Such signs usually have no meaning in isolation – rather the meaning becomes an effect of what happens to the object that the sign is visually referring to. Importantly, such signs are shown in the context of objects, making the reading experience visually direct. The dynamics and directness that contextual signs bring to comics could also be used for programming. Contextual signs in comics are shown as an effect of what happens to a character, and in programming similar signs could be used to produce an effect.

The field of semiotics, which is also applicable to computer systems [1], studies the use and interpretation of signs. Three classical types of signs used in semiotic analysis are *iconic signs*, *indexical signs*, and *symbolic signs*. An icon is a sign that has a similarity or likeness to the signified (e.g. a picture of a person). An index is a sign that is contiguous with and has a spatial, temporal or casual relationship to what it signifies (e.g. smoke as a sign of fire). Finally, a symbol is arbitrary and gets its meaning by convention.

Some contextual signs in comics have indexical and iconic properties. Speed lines can be thought of as an abstraction of iconic "motion blur", or as a variation of indexical traces like footprints. Other contextual signs are symbolic and are learned by convention. However, within the internal sign system of comics, these signs may take on indexical and even iconic qualities [8]. An onomatopoetic symbol, like the letters ***Zzzzzz*** (used to signify that someone is sleeping), has indexical qualities, because (like smoke) it appears in context. In the mind of the reader, seeing this symbol is almost like hearing a snoozing sound. McCloud writes: *"Within a given culture these symbols will quickly spread until everybody knows them at a glance."* [10]. If a comic book style of programming would catch on, a similar evolution might take place for visual programming.

For the purpose of visual programming, we define a contextual sign as a sign that appears in direct relation to another sign (commonly an iconic sign representing a domain object), and that signifies some property, action, or behaviour of the other sign. There are two core characteristics of contextual signs.

- First, the sign is shown in *the immediate visual context* of the character or object having the feature represented by the sign. This is not the case in all visual programming tools, where the "code" is sometimes hidden, for instance at the "back" of the objects, or in a dialogue box.
- Second, signs refer to *perceivable actions and properties*, not detailed actions at a low abstraction level. Compared to lower-level visual "programming code", contextual signs refer to higher-level concepts, similar in spirit to the symbols used in the sketch provided by the girls in our first example.

Thus, an important aspect of using contextual signs for programming is to recognize that programming can be performed on different conceptual levels. Programming in ToonTalk [6] is for instance essentially based around animated robots that represent algorithmic program operations. However, ToonTalk also supports a higher-level mode of programming with behaviours [14]. Behaviours consist of robots packaged in a special way, so that they can be added to the back of pictures and other objects to make them perform specific actions in a game, for instance, move in a certain direction, play a sound, or disappear upon collision with other objects. Previous work has shown that using libraries of such predefined programming behaviours is a useful approach for children to be able to practically realise the systems they want to build [14].

Figure 3 and 4 illustrate the difference between rule-based programming and contextual signs, using graphical rewrite rules as a frame of reference. Programming with graphical rewrite rules (also called visual before-after rules) is based on specifying how a part of a grid-based visual world should change when a visual precondition is fulfilled [13]. When programming using this model, e.g. for making an object move left, the child has to transform her conceptual thinking into a rule, specifying the correct precondition ("if the square to the left is blank") and the corresponding action ("then move to that square"). Figure 3 gives an example of how such a rule may look. Even if this model may be understandable for many children, specifying every aspect of a dynamic game on this level of detail can be quite inefficient. Moreover, defining actions in such detail is a big step from the design considerations that the children are primarily concerned with, as shown for instance in the balloon-game in Figure 2.

Behaviours, in our case expressed as contextual signs, are different in that they may abstract away algorithmic details,

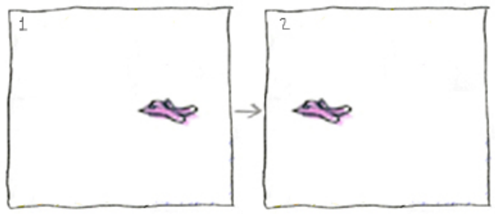


Figure 3. A graphical rewrite rule for a "move left" behaviour.

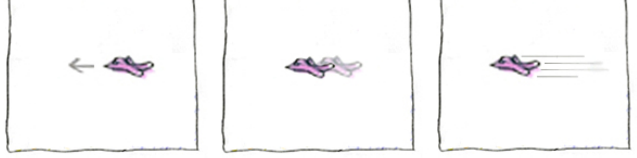


Figure 4. Alternatives for the move left behaviour expressed through three different contextual signs: a basic sign of an arrow pointing to the left, a "ghost image" motion marker, and comic-book style speed lines.

such as explicit conditionals. For example, a behaviour for moving to the left, would when applied upon an object make it move left, without any need for specifying a precondition and an action. Figure 4 gives three examples of how different contextual signs can be used to express a basic move left behaviour. Note how the use of comic book signs like ghost images and speed lines adds a certain directness to the representation.

EXAMPLE SYSTEMS

Below we present three different programming systems that have been developed by the authors to exemplify how contextual signs can be used for programming. Note that these should be seen as examples, and that many other approaches to using contextual signs for programming could be imagined.

MagicWords

MagicWords is an experimental educational toy where children can practice reading and explore the meaning of words. Children use words and phrases to give behaviours to characters. Figure 5 shows a screenshot of the system. Words for characters and objects are in the menu on the left. Words for behaviours are on the right. Backgrounds are at the bottom. Each word has a small speaker icon, and by clicking it one can hear the word spoken. Children play with the program by dragging words into the play area.

Character words are transformed into a picture (with the word visible beneath the picture) when dropped on the play area. Behaviour words can be dropped on a character, and the character will get the corresponding behaviour. The example in Figure 5 shows two objects. The first is a “Dog” with the behaviour “Steer with A D” (used to move the dog horizontally with the keyboard keys A and D). The second object is a “Spaceship” with the behaviours “Left” and “Bounce” (the “Left” word represents the motion and will change to “Right” when the spaceship bounces off the left edge of the play area and changes direction).

The following are examples of behaviours we experiment with: Left, Right, Up, Down (motion words), Bounce (makes an object bounce against walls and other objects), Steer (use arrow keys to steer an object), Bigger, Smaller (size changing words), Remove (deletes the object), Nice (attracts other objects), Scary (makes other objects move away), Dangerous (make other objects be deleted when touching them). Many more words and behaviours could be imagined. It would also be straightforward to use phrases, like “Move left” or “I move left”, and “Eat other characters” or “I eat other characters”. Phrases could also have variable parameters, for example “Eat X” or “I eat X” where X would be a name of a character selected from a menu. The user can chose to hide words to make the scene less cluttered, and can also pause the animation to make it easier to add and remove behaviours.

The use of textual word pads in MagicWords is somewhat similar to the use of “tiles” in e.g. Squeak eToys [5]. The difference is that pads in MagicWords are placed directly on the graphical objects, and that they represent high-level behaviours rather than algorithmic program code.

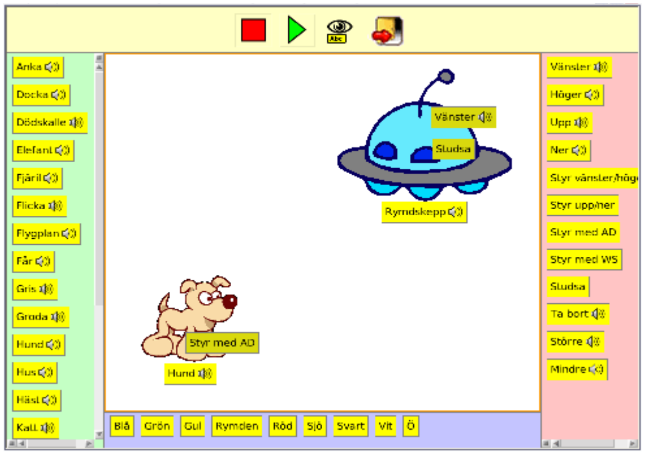


Figure 5. Screenshot from MagicWords. Note that words are in Swedish. See text for translations and additional details.

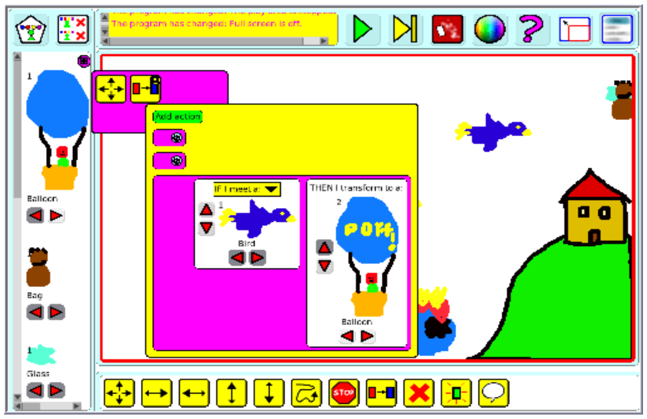


Figure 6. Example of graphical signs used to specify behaviours in BehaviorCards.

BehaviorCards

BehaviorCards [12] is an experimental visual programming system that uses graphical signs to represent behaviours. In BehaviorCards, the signs are displayed on a “card” next to the character being programmed. The screenshot in Figure 6 shows an example from the game the girls designed in Figure 2, which was programmed in BehaviorCards as part of this research. The behaviour card that is visible describes what should happen when the balloon meets a bird. The text reads “IF I meet a [picture of a bird] THEN I transform to a [picture of a broken balloon]”.

BehaviorCards uses a concept similar to classes. The objects being programmed (the “classes”) are in the gallery to the left. Instances of these objects are created by dragging them into the play area. An alternative approach would be to display the behaviour symbols directly on the instances on the play area, as contextual signs. Figure 11 shows an experimental design that illustrates this concept.

Both MagicWords and BehaviorCards are implemented in the Smalltalk-system Squeak [5].

The Tangible Programming Space

Our third system (see Figure 7) differs from the other two since it does not only use contextual signs displayed on the computer screen, but also physical blocks and cards that are used for the interaction with the system [3,4]. The system is designed explicitly for groups of children to create dynamic simulations, games, and interactive play worlds that are run on-screen. Programs are made by positioning blocks on a mat on the floor. The blocks are indicated as squares on the screen (see screenshot in Figure 8). By placing plastic cards on the blocks, the users add pictures of objects to the screen and give them properties and behaviours. Cards are also used for playing, stopping, and saving a game or simulation.

Currently, the system is used at the Swedish Museum of Natural History in Stockholm, with a library of behaviours for building dynamic food-web simulations. The museum setting implies that the children will only get approximately one hour to complete a simulation, hence the library of behaviours and the set of pictures has been tailored especially for this use context, with focus on eating, starving, and moving. For use in other settings other sets of behaviours would be provided.

Figure 8 shows how contextual signs are used to indicate which behaviours and properties that are attached to each of the objects on the screen. Figure 9 shows a detailed image of one of the animals in the simulation. The animal itself is colourised in a red shade (indicating the colour-property "red"), and the three "PacMan" symbols show that the animal will "eat" anything that is either green, blue or yellow. Thus the colour property works to specify interactions between objects. The arrow at the bottom left represents the motion behaviour that has been attached to the animal. When the program is executed, all the programming signs are hidden, and the objects start to act according to the behaviours given to them. Contextual signs as well as textual labels are used also on the cards to illustrate actions in ways that are richer than can be captured in the graphical symbols on the screen. Figure 10 shows the design of two such cards.

The Tangible Programming Space was built using RFID technology and Macromedia Director.

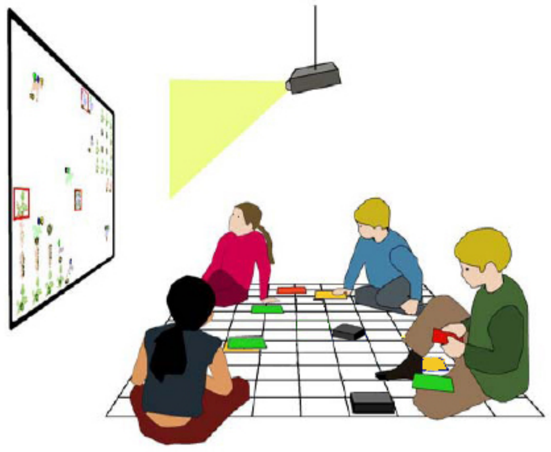


Figure 7. Children in the Tangible Programming Space.

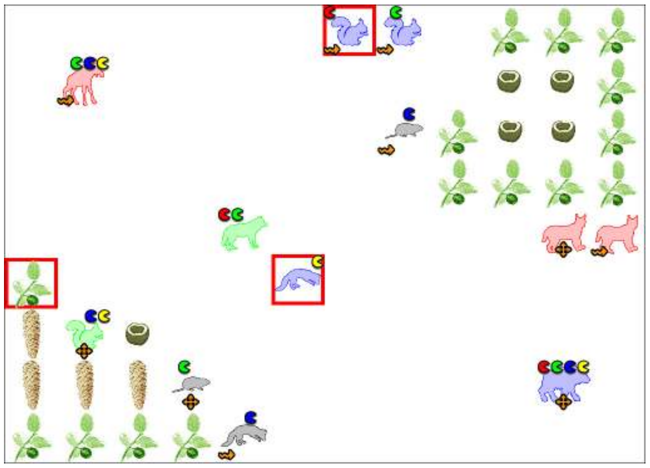


Figure 8. Screenshot from a simulation built with the Tangible Programming Space.

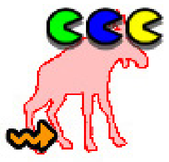


Figure 9. Detail of the elk in Figure 8. Note the signs for "eating" and moving.

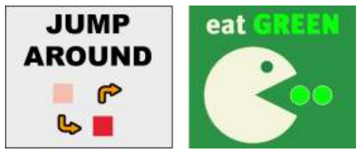


Figure 10. Two cards used for adding "Jump Around" and "Eat Green" behaviours to objects.

DESIGN ASPECTS

Based on our experiences with the design of the systems described above, we discuss some of the design options we consider relevant for developing construction tools based on the idea of programming with contextual signs.

Layout of Signs

The basic property of contextual signs is that they are placed in the immediate visual context of the object that they refer to. This helps showing which object is controlled by a sign, and can give a direct visual impression of the resulting behaviour of the program. For example, the signs displayed on the elk in Figure 9 provide a way to "read" the basic runtime properties of the elk.

Another example is given in Figure 11, which shows a design based on BehaviorCards where the behaviour symbols are placed directly on the objects on the play area, as contextual signs. The birds and the piece of glass have motion behaviours. The two bags have delete behaviours that make them disappear when touching the balloon. There are parameters for specifying this behaviour on a card that can be opened for the sign. The balloon has a behaviour for moving it with the arrow keys, and a behaviour that changes its picture when colliding with other objects. This behaviour has a quite complex behaviour card, and handles both "picking up" bags and the destruction of the balloon in case it collides with a bird or piece of glass (this is the card shown in Figure 6). This example hints at how the game the girls designed in Figure 2 could be programmed in the style we propose.

There are different ways to layout contextual signs. When adding several signs or labels to the same object, a consequence may be that they obscure the object, so that it is difficult to clearly see the object that the signs refer to. Moreover, the signs could themselves overlap and obscure each other. This is a serious issue that needs to be considered when designing systems based on this paradigm of programming.

As a way of dealing with these issues, the Tangible Programming Space uses a structured layout, meaning that all signs have a predefined position in relation to the object they refer to. This is possible due to the specialised scope of the system with a limited number of behaviours, meaning that an object can have all available behaviours attached to it without any of the behaviour signs obstructing another.

In BehaviorCards, the signs for an object are shown in a panel displayed to the right of the object, and thus no sign obscure the object and signs do not obscure each other. However, the user can rearrange the order of the signs in the panel. MagicWords uses a free layout, meaning that the user may arrange the signs the way she likes. A free layout can become cluttered, but freedom to layout signs could also improve readability (and be fun). Thus, a user-made layout can be both an advantage and a source of confusion (similar to badly formatted textual program code).

A problem with placing contextual signs directly on objects on the runtime stage is that it can be undesirable to see the signs when a program is running, e.g. when playing a game. In MagicWords there is a button for hiding and showing the signs, in BehaviorCards the sign panels can be hidden, and in the Tangible Programming Space the signs are automatically hidden when a program is started.

Visual Expressiveness and Generality

In comics, contextual signs closely interplay with characters and objects. Characters in comics commonly display facial expressions and body postures that convey the action in the story, and the signs are carefully crafted for a specific character drawing. Such drawings communicate very effectively, but this level of graphical expressiveness can be difficult to achieve in a general programming tool. The examples we have created are not very comic book like. Additional work is needed to explore how it may be possible to create visually effective signs for program behaviours.

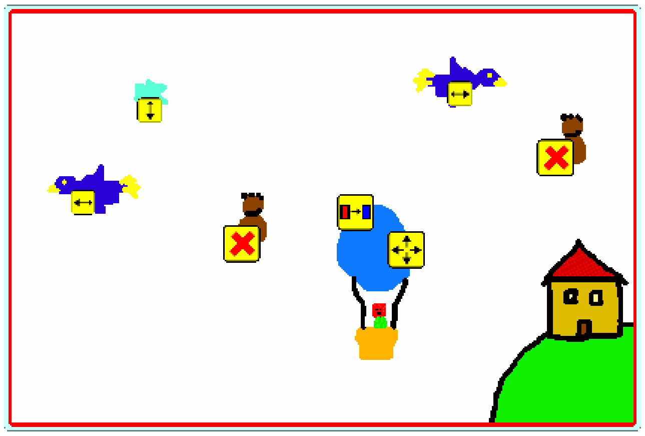


Figure 11. Design sketch based on BehaviorCards that uses contextual signs. Note that the behaviour signs are placed directly on the objects in the play area. The behaviours can be opened to show a card where the behaviour can be configured, similar to the behaviour card shown in Figure 6.

The problem with a programming tool, especially when the programmer makes her own drawings, is that it is not possible to tailor the design of the signs to a specific character or object. The signs need to be designed in a generic fashion, which can make the visual presentation less striking and effective compared to comic books. In addition, if the layout uses a free format, it is up to the programmer to position the signs.

Behaviour Parameters and Conditional Triggers

Behaviours can have different levels of complexity. Some high-level behaviours can be used without need for specifying parameters. "Move with arrow keys" is such an example. But even for this simple behaviour there could be a need to specify additional parameters, for example how fast should the object move, should it move one step at each key press, or should it start moving in a specific direction?

The user interface of the programming tool could provide pop-up menus or dialogue boxes to specify behaviour parameters. This approach is used in BehaviorCards, see Figure 6. BehaviorCards also uses the size of signs to specify parameters such as speed, making it possible to directly set the speed by resizing a motion sign.

Behaviours often implicitly include preconditions, e.g. "Pressing the left arrow key makes me move left", but the user does not necessarily have to think in terms of executing low level algorithmic operations. Behaviours have the potential of adopting the tool to match the thinking of children in the process of building a dynamic game, rather than requiring them to adopt their thinking to the function of the programming tool.

There are however several kinds of behaviours that require specification of conditional triggers, for instance behaviours that involve preconditions related to interaction between objects. In the Tangible Programming Space, the use of conditional statements is replaced by the ability to make your own classes or groups of objects, using the colour-property. Assigning a logical "colour" to an object is primarily used to define interaction with other objects. The "eat" and "chase" behaviours are examples of how this works (see illustrations in Figures 8 to 10). It should be noted though, that this adds an extra layer of complexity, which some of the younger children we have worked with found confusing at first.

Another kind of behaviour we have observed in our work is that children want an object to change its appearance when colliding with another object. For instance, making the picture of a person change from "sad" to "happy" when touching an ice cream. Another example is that an object should be deleted when touching another object, for example deleting the ice cream to show that it is "eaten".

In systems like Stagecast Creator [13] and ComiKit [9], characters can have several pictures and the current picture can be used in the precondition for a rule or event. When placing a behaviour sign directly on an object, the current picture of the object could automatically become part of the precondition, allowing for expressing similar constructs as with graphical rewrite rules. This is also how BehaviorCards works, the behaviour for the balloon in Figure 6, for example, is conditional on the currently selected picture of the balloon in the gallery on the left.

A sketch of how contextual signs for picture-changing and delete behaviours can be designed is given in Figure 12. In this example, the behaviours are conditional on the picture of the character the signs are placed on. One could also design a setting for generalising the behaviour, so that it would trigger regardless of the current picture of the object.

Note that it is not always clear on which object a behaviour should be placed. This can be illustrated by the "who eats who" example, meaning that the delete behaviour in Figure 12 could go on the character ("I delete X") as well as on the ice cream ("X deletes me"). All the three systems that we have discussed use different ways of dealing with this specific issue, and there are probably many more options.

Other examples of behaviours with conditional triggers include time-based actions (e.g. "Move left every 3 seconds"), and keyboard actions (e.g. "Delete when D is pressed"). By displaying menus "inside" a contextual sign, the programmer may specify the trigger for an action in a way that is both flexible and visually direct. This style of programming would be somewhat similar to graphical rewrite rules, but the connection between the program signs and the objects influenced takes on a different perspective, and is visually quite differently represented.

The Use of Textual Labels

Certain behaviours are difficult to illustrate using graphics only, and a picture that is ambiguous can be made easier to make sense of when used in combination with a textual

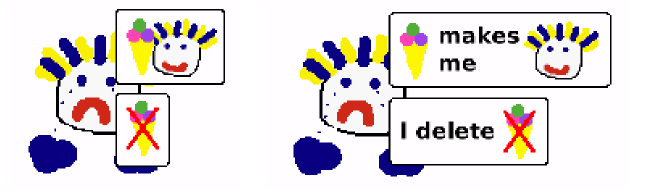


Figure 12. Two design sketches of contextual signs with behaviours for becoming happy when touching an ice cream and for deleting it so that it gets "eaten". The rightmost sketch uses textual labels to clarify the meaning of the signs.

label (cf. [11]). The Tangible Programming Space, for instance, has a behaviour called "Jump Around", which through only its visual representation could be interpreted in many different ways. However, the words "Jump Around" on the physical card make it easy to make a connection between the graphical sign and its corresponding behaviour. (Note the difference between creating text and recognising text; while programming by typing text requires "knowledge in the head", programming by using menus of textual "tiles", as in e.g. Squeak eToys [5], provides "knowledge in the world".)

An argument against relying on text for understanding signs is that many younger children are not able to read. However, we believe that children who can not read are usually fully capable of learning and remembering textual symbols, as well as graphical ones. Furthermore, as is illustrated by MagicWords, this could be a way to help learning how to read. In addition, it should be noted that comics is a multi-modal medium in that it uses both text and concrete and abstract images.

The issue of using words and/or graphical symbols to represent behaviours is also related to the visual appearance of contextual signs. Text can be less ambiguous and more self-explanatory (like a program comment). Graphical symbols on the other hand consume less space, you do not have to be able to read, and they may be quicker to recognise. We believe that different behaviours require different designs. For some behaviours a graphical representation may be most natural, such as speed lines for moving, and for other ones, a textual or mixed textual and graphical representation will be more appropriate. The nature of the system and its users are of course the most important factors when considering design options.

We have not yet explored the topic of how the meaning of a textual label changes depending on the exact wordings used to express the same computational action. When using textual labels to describe behaviours, there are several possible ways to phrase the texts. Depending on the perspective taken by the programmer, the phrasing of the same computational action take on different forms. For example, one option is to use first person statements like "I" and "Me", which may help children take on the perspective of the objects in order to understand their behaviour (e.g. "I delete Ice cream") (cf. [11]).

Since rephrasing of contextual labels does not change the computational logic, it is well worth considering how children themselves tend to describe similar actions. In the sketch provided by the girls in Figure 2, the perspective used is for instance consistently that of the imagined player of the game. It is then inherent in the design style that one of the objects (the balloon) is "you", and accordingly, labels such as "catch" and "avoid" become unmistakable. We believe that such examples can be valuable for the design of programming resources for children.

CONCLUSION

We have illustrated how programming tools for children can benefit from actively exploring the visual sign language of comics. More specifically, we have shown how contextual signs could be used for representing programs in a way that is similar to how we have observed children to sketch games on paper.

Contextual signs have two important properties. First, the signs are displayed in the *immediate visual context* of the objects that they refer to, which adds a characteristic "visual directness" to the representation. Second, such signs depict *perceivable actions and properties of objects*, implying programming at a higher level than in algorithmic and rule-based models of programming. These characteristics could make it possible to perceive how objects in a program are meant to be working, just by looking at the visual appearance of the objects and their surrounding signs.

A limitation of the approach is that the details of more complex behaviours are difficult to represent in the format of contextual signs. It can be problematic to specify parameters and options for behaviours in a way that is easy to use. For more general and algorithmic programming at a more detailed level, a complementary lower-level language would be needed. Such a language could be based on any existing approach for textual, visual, animated, or tangible programming.

Nevertheless, we believe that a simplistic programming tool that is limited to predefined behaviours is sufficient for children in several use contexts, and that this approach can make children able to create simple games and simulations in shorter time and with less trouble compared to programming in a lower-level style. Examples of appropriate contexts are schools and public settings like museums, settings where time is a limiting factor and the focus is on creating and exploring dynamic content, rather than on learning "how to think with algorithms".

The systems we have presented are examples of how comics can inspire the design of programming tools, and many more systems based on similar approaches could be imagined. As systems that use this style of programming become more widely used, designers and researchers can learn from children using them, and children could also participate in inventing new kinds of contextual signs and programming constructs.

ACKNOWLEDGMENTS

We would like to thank Jakob Tholander who has taken part in this research, and all the children who have participated in our projects. We would also like to thank Koray Duhbaci, who was closely involved in designing the visual symbols used in the Tangible Programming Space.

REFERENCES

1. Andersen, Peter Bøgh. *A Theory of Computer Semiotics*. Cambridge University Press, 1997.
2. Cohn, N. *Early Writings on Visual Language*. Emaki Productions, 2003. http://www.emaki.net/ewovl.html
3. Fernaeus, Y., Tholander, J. Looking at the computer but doing it on land: Children's interactions in a tangible programming space. *HCI 2005*, Edinburgh, 2005.
4. Fernaeus, Y., Tholander, J. Finding Design Qualities in a Tangible Programming Space. Proc. *CHI 2006*, Montreal, Canada.
5. Guzdial, M., Rose, K. *Squeak – Open Personal Computing and Multimedia.* Prentice Hall, 2002.
6. Kahn, Ken. ToonTalk – An Animated Programming Environment for Children. *Journal of Visual Languages and Computing*, vol. 7, no. 2, 1996, pp. 197-217.
7. Kelleher, C., Pausch, R. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, vol. 37, no. 2, June 2005, pp. 83–137.
8. Kindborg, M. Concurrent Comics - Programming of social agents by children. Ph.D. Dissertation, Linköping University, 2003.
9. Kindborg, M., McGee, K. Comic Strip Programs: Beyond Graphical Rewrite Rules. Proc. *VLC 05*, Banff, Canada, 2005.
10. McCloud, S. *Understanding Comics.* New York: HarperCollins Publishers, 1993.
11. Rader, C., Cherry, G., Brand, C., Repenning, A., Lewis, C. Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages. Proc. 1*998 IEEE Symposium of Visual Languages*. Nova Scotia, Canada.
12. Scholz, R. Behavior Cards. Master's Thesis LIU-KOGVET-D—05/13--SE. Linköping University, 2005. http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-66 (in Swedish)
13. Smith, D. C., Cypher, A., Tesler, L. Novice Programming Comes of Age. *Communications of the ACM*, vol. 43 no. 3, March 2000, pp. 75-81.
14. Tholander, J., Kahn, K., Jansson, C.-G. Real Programming of an Adventure Game by an 8 year old. Proc. *ICLS 2002*. Lawrence Erlbaum Associates, pp. 473-480.