Animatronics, Children and Computation

Andrew Sempere
Grassroots Invention Group, MIT
29 Bates Road, Floor 2
Watertown MA, 02472 USA
andrew@media.mit.edu

ABSTRACT
In this article, we present CTRL_SPACE: a design for a software environment with companion hardware, developed to introduce preliterate children to basic computational concepts by means of an animatronic face, whose individual features serve as an analogy for a programmable object. In addition to presenting the environment, this article briefly discusses the reasons and methods used to reach a set of guidelines, which were in turn used to develop the prototype system that CTRL_SPACE is based on.

Keywords
Programming, Children, Animatronics, Developmental framework, Computational Objects

Introduction

With few notable exceptions (Begel, 1996; Borovoy, 1996; Hancock, 2003; Raffle, 2004), our notion of programming and computation belongs to a bygone era. The fairly recent availability of cheap, powerful computation allows us to spend more computational cycles on interface. In the process of rethinking what an interface to computational ideas means we uncover two critical points:

1. The historical trend in promoting "computer literacy" has maintained a focus on learning how to communicate in "computer language." The true power of computation, the ability to use computational thinking to solve problems, has taken a back seat to learning how to co-exist with technology.
2. This state of affairs is the result of a series of interface design decisions, many of which rely on historical precedent that is largely accidental. Rethinking what is truly important about computation while taking into account the possibilities offered by the access to surplus computational resources by designers of educational systems, we arrive at the conclusion that computation and computers are fundamentally different things. At worst, the computer as the instantiation of computational ideas becomes a blocking factor to understanding those ideas.

It is possible to reconsider completely what computation "looks like" and thus reconceive what it means to introduce children to computation. CTRL_SPACE attempts this by rethinking the idea of "programming." CTRL_SPACE is used in conjunction with an animatronic head, called *ALF: Acrylic Life Form* (Lyon, 2003), which allows us to leverage the inherent familiarity children have with face making and the similarity this has to several basic computational concepts such as objects, parameters and command sequencing.

The traditional approach to computation

Computational ideas existed long before the computer on our desks, and yet it is this object that we interact with and that is most often the focus of so called computer literacy programs. In evaluating and creating new interactive systems for children it is important to recognize that the character of the object we call "computer" owes more to the history of computer use than to any principle of computation.

The primary user interface of the computer (the keyboard and screen), while extremely powerful, is the result of a historical convergence of technologies originally developed for different purposes. Text based programming, with its lists of sequential instructions, has served us well and is likely to continue to do so, but there is little inherently computational in such a system.

The following questions have guided this research from its earliest stages: Is the dominant method of manipulating computation (text based programming) which serves traditional modes of use (quantitative analysis) truly appropriate for all uses of computation? Does it provide the most direct access to computational ideas? Is this method appropriate in developing systems for children?

Visual alternatives to text

There are many existing examples of visual programming languages or programming systems that incorporate visual/spatial elements. Visual Basic, MAX/MSP and Macromedia Director are a few such systems. All are designed for use by traditional programmers and whatever their relative merits, none are appropriate for preliterate children.

One example that comes close is LogoBlocks (Begel, 1996), a graphical programming language designed for use with the programmable brick, a precursor to the Lego RCX microcontroller. LogoBlocks uses the Logo language for programming, but adds color and shape to code the commands as a way of assisting young children and novices in programming tasks. This color and shape coding functions as a substitute for linguistic syntax, but ultimately with LogoBlocks the user is still working with text. Why not eliminate the need for written language entirely?

In the project proposal for LogoBlocks, Andrew Begel examines some of the problems of adopting graphical programming languages, the foremost being the Deutsch limit: "Deutsch originally said something like 'Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system?'" (1996, 2).

With regards to the development of an environment for preliterate children the answer to the Deutsch limit objection is simple: We aren't going to write an operating system. To perform such a task in a fully graphical programming environment is at worst impossible, and at best neither straightforward, efficient nor useful. The purpose of this work is to find the space where visual and physical programming are maximally useful, something that seems to be the case only in particular domains.

The challenge remains in allowing for a seamless transition to more "advanced" techniques when the time comes. In part, the answer to this is found in recognizing that for many people and many cases, computation serves a particular task. The contention is that while nearly everyone benefits from an understanding of computational problem solving skills, general-purpose programming is far from universally necessary. While such a statement may seem obvious, the dearth of tools which support such practice seems to indicate that the statement is not obvious enough!

Programming by example

In their work, Allen Cypher, Henry Lieberman and others (1993) describe programming by example. While Cypher, et al were not explicitly concerned with children, the idea of imitation as a method of programming is shared by the research described here. Imitation, especially with young children, is an excellent way to communicate information. Young children are highly self-focused and frequently express what they want to do "like this." This characteristic is similar to that which Papert (1993) leverages when he discusses body syntonicity and "playing turtle." Body syntonicity, however, remains first person egocentric, while in the case of CTRL_SPACE we ask the children to project themselves onto an external object.

The physical, virtual and intermediate

There have been several physical programming environments developed to leverage children's affinity for imitation. In particular, it is worth mentioning *Dr. Legohead*, an animatronic head that is programmed by direct physical manipulation, which results in the object repeating the users physical action. Dr Legohead was a product of Rick Borovoy's (1996) thesis work.

More recently, the Tangible Interface Group at the MIT Media Lab has developed *Topobo: Physical Programming Instantiated* (Raffle, 2004), a "constructive assembly system" which allows users to build an object and program it by physically moving its parts. As with Dr. Legohead, Topobo records these movements and replays them.

In both cases, computation is attached directly to physical objects, removing the intermediate layer between the programmer and the programmable object. While Borovoy and others outline the reasons that introducing physicality is an improvement over purely screen based systems, eliminating the intermediate layer entirely does away with a host of possibilities.

It remains a basic tenet of computer science that given enough time and space, the analog world can produce the same results as the digital. Even so, there are particular classes of problems and actions that are impractical to model in the analog world. Code re-use is difficult if one has to literally construct multiple instances of the same object. Recursion is nearly impossible. Dr. Legohead and Topobo are excellent and necessary steps in breaking from the tradition of requiring a complex syntax for programming. The next logical step is careful reintroduction of an abstract intermediate software layer to enable better access to the rich power of computation.

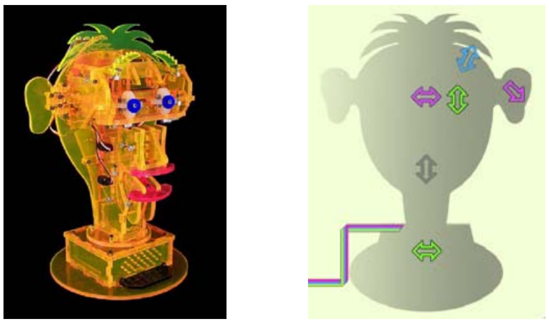
Figure 1. ALF (left) and mapping (right)

Facemaking, containership, debugging

CTRL_SPACE interfaces a general purpose input device with ALF: Acrylic Life Form (Lyon, 2003), an animatronic head (shown in Figure 1) designed and built by Chris Lyon, a member of the Media Lab's Grassroots Invention Group. ALF has six features that are controlled by the Tower modular computer system (Lyon, 2003). As an object, a face can be used to represent a kind of computational containership. It can be broken down into component parts and easily sequenced to create actions. It is easy to discuss the face as a single object and also to refer to its parameters (eyes, ears, mouth). One can issue a command to the object (make a sad face) and then adjust individual parameters (now raise one eyebrow) and the outcome is immediately visible. Considered this way, the potential for addressing a wide range of computational concepts using a face is readily apparent. For example, one could imagine presenting the idea of a state machine with a face. Debugging is made simple by virtue of the fact that the wrong sequence of commands results in a face that is immediately visually recognizable as "wrong."

Perhaps more important than the fact that faces exhibit containership is the fact that faces are intimately familiar objects to all of us. There is a great deal of research that indicates how significant our brains consider facial recognition to be. Piaget discusses the fact that children as young as eight months use imitation (of sounds as well as physical actions) to explore their world. More recent research by Tronick (1986) and Stern (2002) highlights the specific importance of facemaking to early development. By age four, children are fully capable of understanding how to control their own faces and are intimately interested in the notion of representation on the face (what indicates sad, happy, angry). Therefore, the face provides an object that is readily understood by a four year old, has a familiar analog (one's own face) and at the same time demonstrates a kind of containership that is useful for accessing a number of computational ideas.

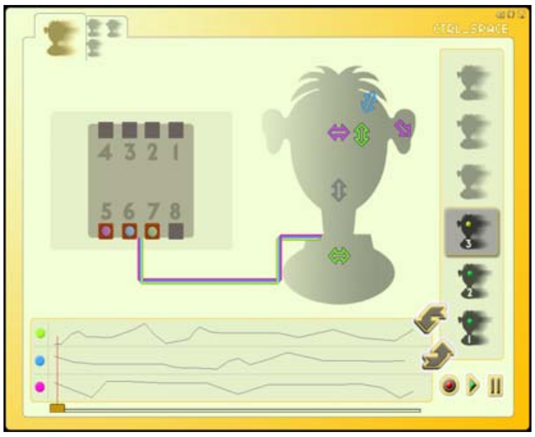

Figure 2. Action creation mode

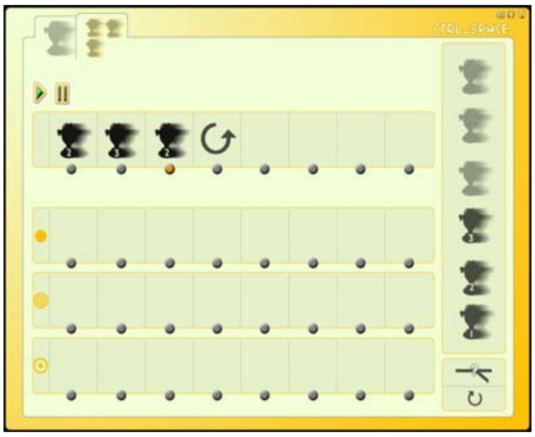

Figure 3. Action sequencing mode

Storytelling and sequencing

While it is the face robot that allows for "object-orientedness," it is the nature of storytelling that allows for children to establish a rule set. By treating ALF as a puppet in a play, you have an actor in a story. The story becomes a script and can be thought of as programmatic sequencing. The introduction of sensor data as an event trigger provides a mechanism to introduce logic structures and conditionals. The addition of multiple ALFs or similar objects would allow for increasing complexity, multiple characters and parallel rule sets.

Representation of actions

The CTRL_SPACE environment is an attempt to leverage both the power of physical interface and of software abstraction. The system is fully graphical, contains no text, and centers around the idea of action. The environment supports two modes of use: action creation and action sequencing. Figures 2 and 3 show screenshots from the action creation mode and the action sequencing mode respectively.

Actions are represented by two related fields: the timeline, which shows a visual representation of change over time, and the mapping of these values to particular features of ALF, as shown by the color coding of the arrows on the ALF head (Figure 1.)

Creation mode allows for the creation and editing of actions, which may be stored for later use. Saved (or minimized) actions are represented by the "ALF in motion" icon and are stored in the action palette on the right side of the screen.

Once a child has built up a library of actions, the action sequencing mode may be used to define a "program" consisting of a sequence of a number of actions. Sequencing mode also introduces basic logic structure and branching on the basis of conditionals.

Representation of conditionals

When users drag the conditional branch icon onto a frame (or click on a frame which contains a conditional), they are presented with a dialog box that allows them to adjust the type of conditional. There are two types of conditionals, blocking and non-blocking, which correspond roughly to *wait…until* and *if…then…else* statements respectively. Blocking conditionals are indicated by a red question mark and cease program execution until the condition is true, at which point the program branches as indicated. Non-blocking conditionals are indicated by a yellow question mark. With non-blocking conditionals, the condition is tested once when the frame is executed. True evaluation branches the program to the indicated subroutine. False evaluation continues the program on the next frame.

The destination of a program branch is indicated by an icon which corresponds to one of the three optional sequences specified below the main sequence. By using a loop or another conditional in the frame following a non-blocking conditional, the user can create more complex logic structures, as shown in Figure 4 along with a more traditional textual representation in pseudocode.

```
if CONDITION A
{
subroutine ⦿
}

else
{
   if CONDITION B
   {
  subroutine ◎
   }

   else
   {
          waituntil(CONDITION C)
     {
            subroutine ◎
     }
}
```

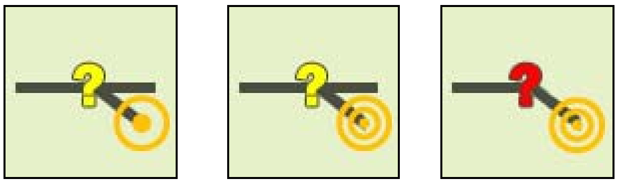

Figure 4. Logic Structure and Equivalent

CTRL_ARM physical interface

Early on, it became apparent that it would be useful to have a device that would bring the act of issuing commands to ALF closer to the physical act of puppeteering. At the same time, it seemed important to create an interface that lent itself to the use and discovery of computational abstraction. This requirement seemed to call for an interface that was *not* a replica of ALF.

To that end, a two axis armature called CTRL_ARM (Figure 5) was constructed. CTRL_ARM uses analog potentiometers to measure hand movements, sending data to the CTRL_SPACE software. CTRL_SPACE allows a child to map sensor inputs in real time to one or more of ALF's features. The software also allows users to record the sensor input and to play it back at any point in time.

The act of mapping the world to digital space is itself a computational idea and is supported most directly by allowing the CTRL_ARM motions to be mapped arbitrarily to one or more of ALFs features. Motion occurs in real time, but computation allows it to be manipulated in any number of ways. CTRL_ARM provides concrete access to ALF in the sense that it involves physical motion, but abstract access through computation in that it allows for arbitrary mapping of sensors.

The power of augmented reality

In terms of introducing children to computation, the physical world is a wonderful starting point because of its familiarity and children's natural inclination to explore the way objects move, bend and break. Augmented reality marries the familiarity of our analog world and our natural inclinations to hold, shape, poke and prod with our hands to the infinitely malleable world of computation.

The design of a physical interface for use by children is in no way a trivial task. CTRL_ARM is presented as one example of a possible physical interface (and a very simple one at that). The topic of design of physical interfaces for children deserves a thorough investigation and may even serve as a site of learning for children, who may benefit from designing their own interfaces. In an effort to better support this, CTRL_SPACE was built around the idea of generalized sensor input rather than "CTRL_ARM input," allowing any number of existing or future input devices to be built and used in conjunction with animatronic objects.

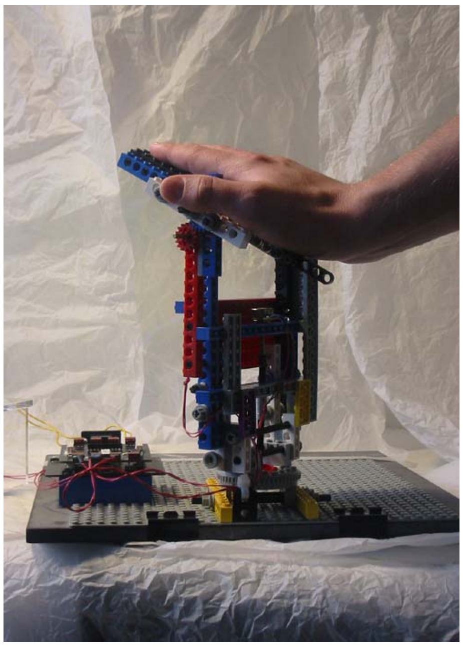
Figure 5. CTRL_ARM

ALF represents a class of objects

An animatronic head with discrete movable parts is a good choice for this research, but it is important to note that ALF is presented here not as an ideal object, but as a representative of a class of objects.

ALF is a head-only robot with a limited number of movable parts. While it has proven more than sufficient as a proof of concept device, much can be gained by using other animatronic objects. The CTRL_SPACE software environment is ultimately intended to be multi-purpose in its ability to control a wide range of motor driven programmable objects.

In order to encourage this, ALF's control structure (based on the Tower modular computing system) may be completely removed and re-used with CTRL_SPACE to manipulate whatever objects one wishes. In addition to using other existing objects, the possibility of designing one's own animatronic character or object is quite compelling, opening up an entirely different and interesting set of ideas which intersect the fields of engineering, materials science, physics, electronics and control feedback. This idea deserves close future attention, especially as it provides a continuum from basic to more complex computational tasks as the students grow in age and understanding.

Interaction as a method of design

Participatory design methodology by definition involves the end users in every stage of the process. This is markedly different from a traditional "focus group" approach, where the project is completed, presented and revised based on a "study" of user interaction. Instead of divorcing it from the development cycle, assessment of the system is an ongoing process inextricably linked to development of the system. Each stage of development is literally the result of real world evaluation.

Prior to the development of CTRL_SPACE, several prototype software environments were developed and tested by a small group of children of the target age (4-7). During each phase of software development, a workshop was held with one or two preliterate children, during which the children interacted with ALF, CTRL_SPACE and the system's designer. As a direct result of these workshops, changes were made to the software and hardware, and the process reiterated. Over time, these workshop experiences, coupled with research into the historical role of computation, led to the compilation of a set of ten general design guidelines intended to aid the development of environments which involve children and technology (Sempere, 2003). This process is detailed in the paper just mentioned, but here it is worth stressing: none of these guidelines could nor should have been developed without the involvement of the target audience in the form of workshops. All of the guidelines (as well as the CTRL_SPACE environment itself) owe their existence to design by participation.

The environment described above represents the final result of a process of participatory design. Evaluation of the system as presented continues the iterative modify/try/revise process.

Process of Evaluation

In order to evaluate progress and validate design decisions, a number of workshops were run which children of the target age group. ALF and the software were presented in an informal setting and the children were allowed to determine the direction the session went. For the first ten minutes or so of each session, I gave no introduction whatsoever, allowing the children to experiment with the controls, push buttons and ask any question that came to mind (including a vast number of which had absolutely nothing to do with this research). Allowing the children to continue exploring on their own, I observed what was going on. When a particular activity seemed to me to imply an understanding of a concept I would ask questions to try and determine (without leading) what the child was thinking at the time. If the child seemed frustrated, I would offer them help, trying as much as possible to ask them what they were trying to accomplish and what they thought “broke” rather than assuming anything at all. In some cases, when I felt the child had a fair grasp of the system and was ready for a challenge, I would ask them to complete a task designed to help me understand if they understood a particular idea. If a child “failed” in a task or stumbled at any point, I worked from an assumption that these points were weaknesses in the system, and it was this that guided my design process for the next revision.

Example of Process: The Two Second Wait

All sessions were audio-recorded for later review. In the following excerpt, Alex notices the *wait* functions in the programming mode of an earlier version of the software. I have provided a quarter, half and full second options. I am hoping that Alex will understand that he can execute these no-op commands multiple times to achieve longer wait times. I ask him outright.

> How do you think you would do that if you wanted to have more than one second?
>
> I don't know…
>
> Well, like… click it again? Like click, like click it again?
>
> Try it and see
>
> Okay, let’s see… program… okay let's see "Silly".
>
> *click*
>
> Okay. Wait one second.
>
> *click* *click*
>
> Mmm hmm! I think so!
>
> So what will this do? Tell me before you click run.
>
> What??
>
> What do you think this is going to do? right now?
>
> Those two?

Uh huh

Uh I think it's going to do what I want it… what I think it would do. I think…

Which is?

I think it would wait two seconds.

OK

Let's see.

Okay I'm just going to do this. Run.

One… one two… Yup! Cool!

With very little prompting from me, Alex understands one basic notion of containership - that multiple instances of the same object can be created, and that they will exhibit the same properties as the original primitive. Two one second waits in succession is the same as one two second wait.

This short example should serve to demonstrate the methodology used. In this case, the interaction provides one data point in favor of a design decision. In many cases these exchanges provided data indicating problem areas, which were revised and presented again in another workshop. For example, while my stated intention was to create an environment free of text, early version of the software contained text labels largely for my benefit. Alex (the student featured above) was a particularly precocious child able to read quite well at four. Reading my label off the screen, Alex began to refer to his functions as "user defined" as opposed to the set of "built-in functions" I provided. During a later session with the same software, another child (Sam) became quite frustrated at his inability to read the labels on various buttons. Neither frustration nor the adoption of my own computational vocabulary were desirable outcomes – I made it a priority to produce the next version of the system as text free as possible.

Guidelines for the development of software and hardware environments for children

In an effort to formalize what was learned from his process, a series of ten guidelines were created and then used to create the software environment presented earlier. Although it is beyond the scope of this article to cover all ten guidelines in depth, they will be listed here for consideration, and we will take a brief close look at three considered most critical.

1. Guidelines are in the service of the participants and subject to change by them.
2. The use of computation should serve a clear purpose.
3. Users must be able to play with underlying rule set, not only its parameters.
4. The designer should avoid excessive error correction.
5. Ambiguity is a good thing.
6. Difficult doesn't mean better.
7. The system should allow connection to the familiar.
8. The system should support growth.
9. The system should encourage reflective public interaction.
10. The system should encourage the creation of an artifact.

As stated the guidelines themselves are general purpose and may seem superficially obvious. Careful consideration, however, will reveal that stating the obvious is a worthwhile activity, made evident by the fact that the list of existing environments which do not follow basic common sense (let alone any kind of design methodology) is far longer than the list of environments which do. In any case, it is the specific context and application of each which makes them effective in designing software environments for children

Guideline 2: The use of computation should serve a clear purpose.

Technology should never be used to justify an activity. Often, computers are used to give credibility to what is otherwise seen as a frivolous pursuit. Educational systems that have cut their art programs, for example, might allow children access to drawing or media manipulation tools on the grounds that the ability to operate these tools is a "useful skill." While this may be true, the emphasis on the industrial utility of image manipulation is demeaning to the user and to the process of expression.

The "wow factor" of new technology is often used to disguise otherwise insufficient material. The most egregious examples of this can be seen in the "edutainment software" that flooded the early personal computer market. Of questionable educational value, these programs purported to be effective merely because they made use of cutting edge technology. "Drill and Kill" math training programs are similarly flawed: the computer in this case has merely taken the place of a teacher holding flashcards – an effort of questionable pedagogical value that, at very least, cannot possibly justify the expense of the computer itself.

The introduction of computation must enable access to the things that make computation useful. For example, a drawing program that mimics the workings of pen and paper is of questionable usefulness: why not simply use pen and paper? The answer to this should be that the drawing program enables more, or at least allows for a different approach to drawing than is possible with paper alone. If a system cannot make a strong claim for why computation is present, it is likely that computation is not necessary!

What we are striving for is an environment that empowers personal expression with the possibilities of computation. This requires first a respect for the expressive nature of the task at hand and second, the ability to correctly match desirable outcome with computational concepts.

Guideline 3: Users must be able to play with the underlying rule set, not only its parameters.

Tangible and graphical user interfaces allow us to abstract away the quantitative nature of computation and file down the rough edges of an otherwise difficult to use device. While this can be used to great effect, the designer must be careful not to remove all access to computation. There is a fine line between true interactivity, where the user actually has some effect on the system, and the relegation of the user to the status of "cue issuer," where the only effect one has is on the pacing of pre-scripted events.

In an abstract sense, programming can be described as the ability to manipulate a logical rule set for interaction, while the running of the program enables others to tweak the parameters. As any programmer knows, the act of writing a program consists of a great deal of tweaking, but the programmer always has the option to change the underlying logical assumptions that define the behavior of the computational object.

In an environment that seeks to introduce very young children to computational ideas, it is not optimal to support every logic structure known to computation, nor is it practical to introduce the kind of syntax necessary to construct elaborate systems from basic computational primitives. At the same time, we do not wish to provide so many prefabricated modules that the use of "computation" becomes the mere stringing along of objects with no understanding of what it going on.

As a final point, it is not likely that this type of understanding will come from an interaction between a child and the system alone. This is one part of the system that clearly depends on the presence of a facilitator.

Guideline 4: Ambiguity is a good thing.

Ambiguity is one of the most powerful features of human communication. While it is sometimes problematic, it is ambiguity that allows for humor, art, poetry and efficient context sensitive communication. Imagine if, like a computer, human beings demanded an entire rule set and strict syntax for communication. What would you do if such a being was crossing the road in the path of a truck? Write and debug a program?

Access to computation has too long required users to formalize their desires in unfamiliar ways, but only recently has it become possible to endow computers with the processing power capable of dealing with ambiguity. This can be done by providing a system with a predefined context, placing the onus of interpretation on the environment rather than on the user.

While formalizing a thought is a fundamental part of thinking computationally, if this is done unnecessarily or in a manner which appears nonsensical, it only causes frustration. If a system can instead provide a clear context and is capable of understanding ambiguous commands within this context, it will free the user to think more thoroughly about what they want to do with computation and less about how to shape their thinking to match the computer's preferred model of the world. This is particularly true for young children, whose sense of context is strong and whose ability to abstractly describe location is far less developed than their ability to point and say "there!"

At the same time, it is recognized that excessive ambiguity can quickly spiral out of control. An environment that provides no structure, vocabulary or system of organization would quickly become impossible to debug, as it would rely entirely on the user's memory of intention to explain a particular step in time. Accordingly, a system should seek to balance support for ambiguity with structure in such a way that minimizes frustration.

Corollary: The role of the facilitator

Out of this set of guidelines should emerge an understanding of what it means to be a facilitator. It should be clear that there is a valid and worthwhile place for facilitation, but that the role is not one of instructor, expert consultant, lecturer or test grader. Rather, a system that satisfies these guidelines requires an active participant whose prior experience gives them particular insight into what makes a better experience, and whose adaptability ensures the material covered remains contextually relevant to the learners.

This definition should also make clear that the current criteria for selecting a teacher (the one with the higher paper credentials) rings false. The criteria is not book knowledge, hours logged or status awarded by an institution. Rather, anyone at any time assumes the role of teacher by participating actively in a community of learners and becoming genuinely engaged in what is to be learned. In a very real sense, this kind of system seeks to flatten the hierarchy that characterizes industrialized teaching methods. The teacher in this case is free to make mistakes because the process is as important as the product.

Conclusion

In CTRL_SPACE, the use of the face robot as an analog to one's own face enables access to computational ideas in a familiar manner. The act of imitation allows the child to teach ALF what to do and the incorporation of sensors for input allows one to literally program ALF by example. An intermediate software environment provides a layer of abstraction that allows access to powerful computational concepts, but remains text free, trading generality of purpose for specificity of task that eases understanding. A careful balance is maintained by virtue of the fact that the CTRL_SPACE software is deliberately focused in scope.

A number of choices have been made in CTRL_SPACE which, while they allow for easier access to complicated concepts for very young children, may prove frustrating for more experienced users. In such cases, it is important to note that the choice has been made deliberately in an effort to make concepts more accessible. The problem of growth is mitigated by the fact that CTRL_SPACE should be seen as one of a family of projects. The hardware is based on the Tower modular computing system; there is little to prevent (and much to assist) students in continuing their work using a high level language of their choice.

Finally, while we have reviewed some of the participatory design methodology used to develop this system, the main focus of this paper has been a particular software and hardware environment. With this in mind it is important to stress again that a crucial and often overlooked component of children, technology, and education is: children!

CTRL_SPACE is the result of a participatory design process that led to a series of design guidelines. The experience of the children and the development of these guidelines are both critical. While technology affords us new ways of communicating ideas to learners, education begins and ends with the people involved - individuals whose learning process must never take a backseat to technology.

References

Begel, A. (1996). *LogoBlocks: A Graphical Programming Language for Interacting with the World (AUP)*, Cambridge, MA: MIT Media Lab.

Borovoy, R. D. (1996). *Genuine Object Oriented Programming*, Cambridge, MA: MIT Media Lab.

Cypher, A. (1993). *Watch What I Do: Programming by Demonstration*, Cambridge, MA: MIT Press.

Hancock, C. (2003). *Real-time programming and the big ideas of computational literacy*, Cambridge, MA: MIT Media Lab.
Lyon, C. (2003). *Encouraging Innovation by Engineering the Learning Curve*, Cambridge, MA: MIT Electrical Engineering and Computer Science.

Papert, S. (1993). *Mindstorms: Children, Computers, and Powerful Ideas* (2nd Ed.), New York, USA: BasicBooks.

Raffle, H. S., Parkes, A. J., & Ishii, H. (2004). Topobo: A Constructive Assembly System with Kinetic Memory. *Paper presented at the CHI 2004 Conference*, April 24-29, 2004, Vienna, Austria, Retrieved October 25, 2005, from, http://tangible.media.mit.edu/content/papers/pdf/topobo_CHI04.pdf.

Sempere, A. (2003). *Just Making Faces? Animatronics, Children and Computation*, Cambridge, MA: MIT Media Lab.

Stern, D. (2002). *The First Relationship*, Cambridge, MA: Harvard University Press.

Tronick, E. Z. (1986). *Maternal Depression and Infant Disturbance*, San Francisco, USA: Jossey-Bass.