Rethinking the Microworld Idea

Celia Hoyles, Richard Noss, Ross Adamson

Mathematical Sciences Group, Institute of Education

University of London

Abstract

In this paper we reflect on the meaning and evolution of the microworld idea. We point out a crucial distinction between user manipulation and modification at three distinct but mutually dependent levels — the interface, superstructural and platform levels. We exploit a case study of two 8-year-old girls playing and rebuilding a simple video game, to argue for the importance of ease of interplay between these levels. We reflect on the ways in which newly-created alternatives to textual forms of representation are redefining the utility and power of microworlds, and offering advantages (as well as disadvantages) for mathematical learning in the sense of understanding inference and mechanism — how things work and why.

What is a microworld?

The microworld idea is about three decades old, time enough for the idea to have become debased. Like 'constructivism', it is a Good Thing, a designation to which the creators of even the dullest and least instructive software often aspire. Despite this, the idea of a microworld as a place where the student, though playing, may stumble over and then ponder important inspirations and concepts, is one that nicely captures at least one crucial organising feature of the original idea.

The pedigree of the concept is indicative of the inspiration behind it: Hoyles (1993) charts its evolution from an AI description of a simple and constrained aspect of the real world to part of a knowledge domain which is changing and growing and which has epistemological significance. Laurie Edwards's (1995) extensive review of microworlds similarly stresses knowledge as a central element, and makes a useful distinction between structural and functional views of the idea. The former view prioritises the idea of a microworld as a concrete embodiment of a mathematical structure that is extensible (so tools and objects can be combined to build new ones), but also transparent (so its workings are visible) and rich in different representations. The latter view prioritises features of the microworld that become apparent in use, where learners are expected to explore and build, learn from feedback while involved in the iterative design of long term projects — rather than in trying to master decontextualised knowledge fragments (following diSessa, 1985).

Clearly the structural and functional views of a microworld are not antithetical, and microworld designers are at pains to try to keep both in mind, most usefully by focussing on the activity structures the microworld is designed to offer, where students construct their own meanings, representations and expressions. It is this facet which, incidentally, provides researchers and teachers with such a helpful 'window' onto the evolution of learners' thinking, and which we have discussed in depth in our book (Noss and Hoyles, 1996).

Thus microworlds are environments where people can explore and learn from what they receive back from the computer in return for their exploration. It follows, therefore, that a microworld has its own set of tools and operations that are open for inspection and change. In this sense, learners themselves are in the position simultaneously of user and designer (interestingly, this is true for computer scientists too — any good programming language casts the programmer in the role of language designer, as they create tools and objects for the solution of problems; see Abelson and Sussman, 1985). These twin roles for the learner lead directly to the idea of constructionism, which argues that effective learning 'will not come from finding better ways for the teacher to instruct but from giving the learner better opportunities to construct (Papert, 1991, p 3). We view constructionism as giving the learner opportunities to build her own physical, virtual and mental knowledge structures. It is this belief, that building things is a locus of significant educational change, that drove the initial rationale for microworlds and provides a crucial organising distinction between systems which put the child in the role of builder and thinker, and those which place him or her in the role of listener or receiver.

To build physical or mental structures requires tools, and a means of employing them. What should these tools be and how are they deployed and combined? To what extent can tools be recast by learners? This is the technical face of a thorny question that we want to explore in this paper, a question which has troubled many microworld designers: to put it bluntly, does the constructionist ethic (at least in so far as it entails computer use) involve the learner in programming? There is little doubt that it originally did: in the past, the creation and reconstruction of executable actions were precisely what characterised programming, the construction of the bits and pieces of (textual) code which produced an effect on the screen (or in the case of Logo, on the turtle's behaviour). Slowly, as the idea of programming evolved and as what was possible and accessible on the screen developed microworlds — even Logo microworlds — were designed which expressly excluded the learner from delving down into the core language[1]. In fact, Edwards argued that a Logo microworld did not *necessarily* involve Logo programming, modification of Logo code or interaction with Logo at all (Edwards 1995 p. 134, emphasis in original). In these cases, the learner only operated with the microworld tools, which may have been written in Logo but were not open for inspection (examples of such microworlds have been designed by Thompson, 1992, Edwards, 1991 and more recently by Kalas, 1997, Turcsányi-Szabó, 1999).

This issue of the availability of programming is not merely of Talmudic importance. After all, programming is the prototypical tool for the constructionist vision, and a microworld without programming runs the risk of avoiding just the thing that gives a microworld its power. If children cannot program at all, how can they build the tools that they need to model and come to understand a mathematical idea? More crucially, how do they know that models can evolve and change? In our view, the key question is: Are students still in the position to build new tools if they have no access to or appreciation of programming?

[1] Actually, such attempts were often accompanied by descriptions and reports which stated that the learner *could* in fact interact with the code and modify the microworld: in our experience hardly any ever did unless very specific help and guidance to do so was on hand.

Our answer to date has been consistently in the negative, although the microworlds we have developed have tended over the years to become more focussed on mathematical structures in order to contain the problems of lack of programming expertise (see for example, Hoyles and Noss, 1987a). Yet in the years since the microworld idea has taken hold, it is precisely this point — the idea of children programming computers — which has become so problematic. For one thing, there are practical objections — programming takes too much time in what is already a crowded mathematics curriculum; programming is too hard (for teachers as well as students); programming diverts attention from the underlying knowledge goals. If the last point is true then there clearly *is* no place for programming, at least within an explicitly educational setting (like school) where the objective is to learn mathematics, say — not programming. But at a deeper level, some have questioned whether the objectives we have outlined for programming are still only attainable in this way.

In this paper, we ask whether it still makes sense to insist on programmability as a vehicle for creativity and constructionist learning. There is an interesting evolution here; of developments in programming languages, the scope of what is possible, and the slow but sure increase in our knowledge base of how children learn within microworlds. We do not intend to trace these developments in this paper (many are available in diSessa, Hoyles and Noss, 1995). Rather we will re-examine our own work, as a way to assess the extent to which both the microworld idea and the meaning of programming have evolved.

The symbolic core of a mathematical microworld

In an early paper of ours (Hoyles and Noss, 1987b), we critiqued a promising and certainly useful program of its time — let's call it software X. When software X is run it displays sequences of numbers that children are asked to inspect, spot a pattern and generalise. We pointed out that:

> There is no opportunity [with X] to develop a constructive approach to the formalisation of the program; the pupils cannot build up the symbolic language for themselves, attach meaning to the specific parts of the syntax or learn through the computer feedback the effects of modification or extensions of the symbolic form. Try to generate a geometric progression within the available framework of the program and it will be clear what the distinction is. (*ibid.*, p. 590-591).

The details of X are irrelevant here. Rereading our critical remark more than a decade later, we are struck by our insistence that the learner *ought* to be able to do the unexpected, to attain expressive power over tasks which the designer had not preconceived — a demanding criterion, which assumes a great deal about the learning culture and activity structures of the learners' environment. From another, more specifically mathematical perspective, we were unhappy that the mathematical structures which underpinned the sequences under investigation were invisible; there was no access to the code so users were restricted to searching for patterns in the output data rather than within the processes by which they were constructed. This point we still regard as central to learning mathematics. All too often children (at least in UK) are diverted by surface relationships between numbers and ignore the structural, mathematical reasons why they may or may not be related (Healy and Hoyles, 1999).

Thus far we think we were right: we remain convinced of the constructionist ethic, and the extent to which programmability of one kind or another is an essential precondition for it. Building up pieces of knowledge on the screen as external representations for knowledge structures necessitates some means of expressing the ideas under construction and that still, by our definition at least, means programming. We are relieved to see that we recognised even then, the importance of serendipitous learning, of finding ways to open rather than close down alternative learning paths, interesting avenues or activities that the designer had not intended. Clearly this aspiration raises challenges for the teacher, (which we have discussed in Noss and Hoyles, 1996) and we recognise that it is precisely this aspect of openness that leads many, if not most, software designers to try to do quite the opposite.

There is, however, one interesting assumption behind our early work, which is evident in the quotation above. It is the identification of *formalisation* with *construction*, between the *symbolic* language and the mathematical ideas embedded in the microworld. Our statement rested on an assumption that the expression of complex ideas and its communication to a computer in a program necessitated formal, rigorous collections of symbols in the form of textual strings. This, we thought, was at the heart of the microworld's utility for learning and we were unashamed on this point. Regarding software X, we said,

> [It] fails as a microworld because it does not offer an algorithmic representation of the concept to be explored. It thus does not provide the 'hooks' to the power of the language that, in our view, provides the essential interplay between mathematical concepts and their formalization. (*ibid.*, p. 591)

And some ten years later, we were still saying in *Windows on Mathematical Meanings:*

> … the fundamental facet of programming environments is the imperative of formalization inherent within them. We will provide many examples of the critical role played by the *linguistic formalization* of programming in aiding learners' mathematical expression and in developing their mathematical understandings. (Noss and Hoyles, 1996; p. 62, emphasis added).

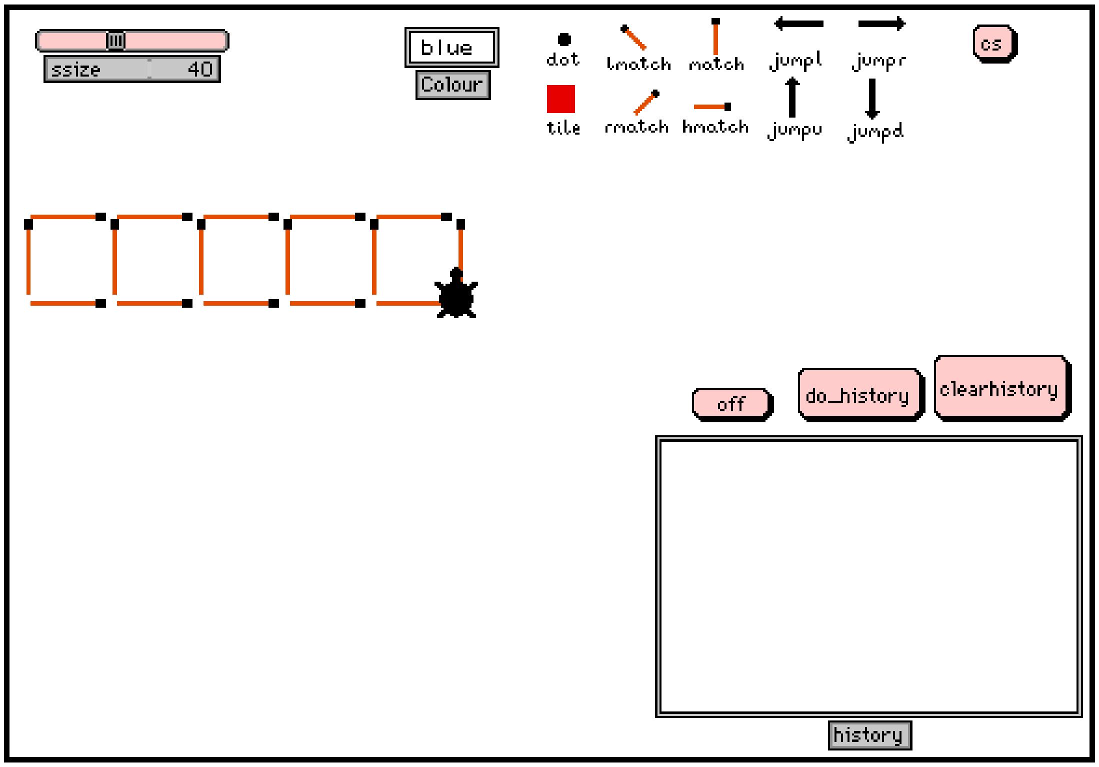
Before we criticise (with considerable hindsight) the assumptions behind this position, we should try to defend ourselves. In *Windows* we give a number of examples of the critical role played by the linguistic formalisation of programming in aiding learners' mathematical expression. Unsurprisingly, these privileged the textual side of programming. But we were also keen to find ways to forge links between textual and other representations. It may help if we describe the examples, one sentence each. They were: a classical microworld for exploring non-Euclidean geometry, employing an object-oriented version of Logo; a conventional application of Logo as a cumulative device for exploring convergence and divergence of series; a new way of representing functional relationships among terms of a sequence; an unexpectedly powerful solution to an old problem, using a massively parallel version of Logo (StarLogo); and a Logo approach to the mathematics of banking, which illustrated programming as a means to unify apparently disparate pieces of banking knowledge.

Looking back on these examples, we remain convinced that student-accessible programming is an essential prerequisite for expressivity (in addition to all the Logo literature here, see also the work on Boxer, diSessa and Abelson, 1986). In one example however, programming was given a slightly altered role, namely to serve as the symbolic representation of a mathematical function *as well as* the glue that bound all the representational modes together. We have considered this example in much detail elsewhere (see, Noss, Healy and Hoyles, 1997). Here we simply summarise the main points relevant to the idea of a microworld.

An example of a microworld

We built a microworld, Mathsticks, (written in Microworlds Logo) which involves the learner in constructing sequences of objects on the screen, and manipulating them directly. The key interesting feature of this program is that while the user is manipulating objects *directly*, there is also graphical feedback as to the results of their actions and a visible (conventional, Logo) program constructed to provide a formal representation of what has been done. By exploiting the relative advantages of direct manipulation and text-based programming, we aimed therefore to construct an environment in which users would learn by linking their actions with the graphical or the symbolic representations — both representations were consistently available. We have noted in the past the problems that children have with maintaining a connection in Logo programming between the symbolic commands and their visual effects (Hoyles and Sutherland, 1989). In terms of elementary geometry, Turtle Math has gone a long way to overcome this problem through its maintenance of a close correspondence between representations and the bidirectionality between the different modes — that is the student can move the mouse and the Logo commands are created automatically (see Clements & Sarama, 1995, Clements, Battista, Sarama, Swaminathan, 1996). We tried to build something similar in our Mathsticks microworld.

The main feature of the microworld was that purpose-built tools were designed to simulate the arrangement of matches in a sequence (see Figure 1), with the mathematical goal that students would come to appreciate number patterns as functional relationships.



An example screen from the Mathsticks microworld

Here we simply summarise our agenda and findings. First the students would be asked to build by direct manipulation of the screen matches, a model of how they saw a given term in a sequence of matches. For example in Figure 1, the students had constructed the fifth term of a sequence of squares. Through this process of construction, we anticipated that students would come to notice the structure of the given term in itself and in relation to others; they would begin to see the particular case as an instance of the general. Our expectation stemmed from two sources: firstly because their mental image of the task could be reflected in their actions in the microworld (for example picking up and putting together the matchstick icons shown on the top right of the screen), and in the rhythm of their pointing and clicking; and secondly because it could be realised simultaneously in symbols (programming code) appearing in the box labelled 'history' and visually as an array of graphical objects. (For details of how the different images the students held of the task, see Healy and Hoyles, 1999).

But Logo did not only serve the role of a symbolic representation of the mathematics involved. We encouraged the students to write a general Logo procedure that could be used to build any term of the sequence. This, we hoped, would force the distinction between those aspects of their code that were significant for sequence structure from those that represented the characteristics of a particular term. In this way too, the dynamic algebra of the programming code would — and did — become a means for them to think about the relationships in the sequence.

Beyond Mathsticks

Our findings convinced us of the power of the Mathsticks microworld and its potential for learning. Our conclusion was that it was the interplay between learners' actions and the different representations of the mathematical relationships embedded in the microworld that was crucial to our students' learning. It seemed to us that we were not only seeing the power of students building connections between multiple representations — although this is of course of considerable importance (see for example software such as SimCalc; Kaput and Roschelle, 1999). The design of Mathsticks aimed for more: we were strongly influenced by the notion of a general programming language from which to build an application, and we saw this as generative of expressive power, beyond that which is possible with the multiple representation approach. It is instructive to consider this approach in the light of the 'programmable application' proposed by Eisenberg, 1994, although this starts at the applications end. His ideal is, however, one with which we have considerable sympathy: to create software that 'allows students to work with both a powerful (and ideally learnable) interface and a powerful (and ideally learnable) programming language (*ibid.*, p 181). It is this gain in expressive power which is also at the heart of diSessa's Boxer idea: in this case, the interface and the language are so tightly integrated that it is sometimes difficult to know whether one is programming are 'just' editing text.

We want to draw attention to two facets of Mathsticks that we regard as important. In the first place, *we provided tools to focus on what mattered to us* — just enough of the relevant structures were embedded in the tools so that they could be thought and talked about, explored and manipulated, at a level 'above' that of Logo. There is an important methodological point here. In order to study the forms of expression employed by learners to mediate their understandings of mathematical ideas, we must tread a careful path between allowing free range to that expression and constraining it too tightly. In the former case, it might be difficult to capture any traces of the learner's thinking — let alone any traces that could be used as resources for mathematical learning. The latter approach runs the risk of constraining thinking to the point of total predictability, in which case we would merely be studying our own preferences (this phenomenon is closely connected with the idea of the didactic contract, Brousseau, 1984, although here we are concerned with constraints built into tools while Brousseau's construct concerns teachers' constraining behaviour). As researchers, let alone educators, we have to respect *and* constrain diversity.

The second point is that at *the same time, nothing was ruled out*: the students had a working knowledge of Logo, knew how it worked in the sense of editing and writing programs, changing interface objects, and were free to adopt any method of construction. Thus the tools could be manipulated by programming, as well as by direct manipulation in a way that maintained connections between the Mathsticks interface level and the level of the language below.

In trying to make deeper sense of the microworld idea, we will now attempt to clarify some of the casual references we have made so far about levels, and introduce the distinction between *platform* and *superstructure*. By platform, we mean the base level at which it is possible for users (rather than professional programmers) to interact. A platform would include high level programming languages but not for example machine code. In most cases, users interact with the platform because the designer *expects* them to do so. Most software (including educational software and, as we mentioned earlier, even some applications in Logo) takes pains to make the platform level *completely* invisible, and, in general, make a virtue out of this perceived necessity on the grounds that only programmers need to know how to program.

Superstructure, on the other hand, describes the objects in the microworld and ways to manipulate them (the matches in Mathsticks). For the moment, it doesn't matter whether this form of manipulation is direct (e.g. via a mouse or speech input), or by entering lines of text which may or may not be the same as the language of the platform. The point is that the kinds of interactions which users' experience and the HCI tools they employ are a subject for the designer, and determine to a great extent what activities and experiences the user has as she interacts with the program. For most users with most software, there is only superstructure.

The idea of superstructure raises new dilemmas. How visible is the platform level? How easy is it for the user to 'descend' to the platform level? How permeable is the barrier that separates superstructure from platform? How rich is the potentiality of modifying tools at the superstructural level along with the interactions that go with it? How familiar can and should the user be with platform tools?

The answer, of course, depends on the pedagogical aim. The issue is not, of course, whether the mechanisms should be open to the user, but *how much* of them should be. But there is a problem, and this time it *is* primarily technical. Whereas it makes sense to maintain a certain eclecticism regarding user interface at the superstructural level (as we did with Mathsticks), it is very difficult to adopt the same approach as regards the platform, where text is all there is.

We should not underestimate this distinction — non-linguistic (iconic, GUI) interfaces are now *the* way that people expect to interact with machines. They do not, in general, expect to program in any conventional sense. There is therefore an interaction barrier[2]: the things you have to do to gain a sense of the microworld's mechanism may be substantially different from the things you do within the microworld. It is not just the things on the screen that are different; it is the ways you move them around that conventionally distinguishes platform from superstructural interaction. *Finding ways to break down this distinction may turn out to represent a significant advance for mathematical learning with digital technologies.*

This issue raises a second major question of microworld design, which is related to but separate from the interface issue above. What is the appropriate grain size for objects and relationships at the superstructural level? What level of complexity is appropriate for users; how far, in other words, should be the distance between superstructural elements and the platform on which they are built? We cannot hope to answer this question in the abstract; it is a research question which we (and many others) are currently addressing, and which we will discuss below in the context of our current work.

[2] We use this term by analogy with the idea of *abstraction barrier* in computer science, the means by which programmers can wrap up pieces of programs to use as tools whose workings need not be visible in the larger program. This is a key means of controlling complexity: 'By isolating the underlying representations of data objects, we can divide the task of designing a large program into smaller tasks that can be performed separately" (Abelson and Sussman, 1985, p.126).

The Playground project: aims and methodologies

In our ongoing research project we have moved to a new arena for constructionism — that of computer games. In this, we are building on the work of Harel and Kafai (Harel, 1988 and Kafai, 1995) who have pointed out that we should endeavour to tap in to children's games culture by adding a new dimension whereby they build their *own* games. Our project is to design and try out computational worlds — playgrounds — in which the objects in a game *and* the means for expressing them are engaging; where the programming of a game is itself a game. We have set ourselves the task of working with young children (aged as young as 4 and at most 8) where it is obvious that we cannot rely on the written word as a means of communication. This has challenged us considerably and forced us to take seriously other modalities of interaction, such as speech as well as direct manipulation. We want to give the children the opportunity to construct creative and fun games, and at the same time, offer them an appreciation of — and a language for — the rules, which underpin them.

In real playgrounds, children play with the objects they find, like swings and roundabouts, balls and sticks. They create new games with new rules (stand on the swings, have three people on the seesaw), make up new games with the existing objects (who can go fastest around the seesaw, climbing frame and slide?), and ignore the objects they find to create new ones (like hopscotch). And in a real playground, kids engage in a wide range of personal styles — through talking and shouting, moving, hearing seeing and so on. All their senses are engaged.

Our virtual playgrounds aim at a similar level of engagement. Unlike most computer games, in which the metaphors are simple, and the tools immediate to provide entertainment, our games are being constructed for learning. And to achieve this, we have had to resurrect the notion of programming (however unfashionable that may be), and reject systems which only allow direct manipulation of the interface but are severely limited in their expressive power.

Using the terminology introduced earlier in this paper, games are played at the level of superstructure, but to change or rebuild them requires an appreciation of and some access to the platform. It is managing this access — both in terms of what can be seen and what can be done — in order to build a game of one’s choice that enables us to help children become aware of how rules operate and the implications of rules. We want to blur the boundaries between user and programmer, between fun at the superstructural level and engagement with the semantics of the platform. In short, we are trying to build a system for children to *play with the rules*.

Just what we end up building and just what the children learn is a matter of ongoing investigation but at its heart lie many classical questions of microworld design. From all the foregoing discussion, it should be clear that we expect the platform to be *intrusive*, that is, we anticipate that children will, at some level at least, know what is happening below the superstructural level, understand how to interact with it, and appreciate the mechanism which makes it work. Actually we are coming to believe that the most important appreciation of mechanism is the idea of mechanism — knowing that there are rules which makes things work represents quite a few steps towards understanding precisely how something works.

But wanting children to have access to the platform has raised problems of choice of a platform that affords access to very young children. We decided to base the design and construction of playground on ToonTalk (Kahn, 1999). Its appeal for our purposes is in its "concretizations" of computational abstractions based on the animated metaphors of the computer game. The new concepts of ToonTalk design are twofold: the provision of powerful high-level constructs for expressing programs at the platform level and the provision at the interface level of concrete, intuitive, easy-to-learn, systematic game analogues to every construct.

The fundamental idea behind ToonTalk is that source code is animated. (ToonTalk is so named because one is "talking" in (car)toons.) This does not mean that it takes a visual programming language and replaces some static icons by animated icons. It means that animation is the means of communicating to both humans and computers the entire meaning of a program. Program sources are not static collections of text or even text and pictures, but are animated, tactile, enhanced with sound effects, and clearly physical. The programs of ToonTalk are encapsulated in the actions of robots which are trained by example to perform a role. The conditions, which determine subsequent performance of the actions at run time, can be generalised or specialised after the training has taken place. Details of how ToonTalk works, its design principles and some applications can be found at http://www.toontalk.com/

We are aiming to design our prototype games in ways that make it straightforward to interact both with the objects in the games and the mechanisms behind the objects which give them their behaviours. To achieve this we are engaged in iterative designing and prototyping with children to find the appropriate level of expression for these mechanisms. Unsurprisingly, we have found that if the level is too low, (e.g. raw ToonTalk at the platform level) such young children are typically unable to grasp what is happening; if it is too high (e.g. all the mechanisms are wrapped up in black boxes) the children might be able to play the games, but are not even aware of the rules that make them work, let alone able to inspect and modify.

Our solution to this challenge was to consider the mechanism in levels, thus using and developing the theoretical constructs presented earlier. Figure 2 presents an example of a game and the different levels of mechanism that are accessible to the user. This example will be helpful in both focusing the discussion and allowing us to present some illustrative work with children.

Figure 2

Three levels of mechanism in a Pong game: interface, superstructural and platform

In games, there are a range of components we shall call play objects: balls that bounce and make noises, wizards which turn into frogs, or dice that control a move. Figure 2a presents a simple Pong game that we built in ToonTalk. It consists of a background, two paddles, a ball and a score. There is a scoring connection between the top paddle and the score, so that when the ball is hit by the top paddle the score increases by ten points. Changes can be made at the interface level by direct manipulation and by using ToonTalk tools (for example, there is a simple way to change the colour of the background, or make copies of any play objects using the 'magic wand').

Our effort in microworld design has been to build a superstructural level on top of ToonTalk which allows children to manipulate the 'things that matter' in the game — in this case the *behaviours* of the play objects. We have created a class of playground objects called ‘behaviours’ which are portable components packaging the functionality of robots into manageable pieces. Functionality for play objects is realised through adding trained robots to their flipside. A key design principle of the behaviours is that we judge them (sometimes correctly!) to be the right grain size for children: that is, they provide an appropriate level of complexity and functionality so that they can be appreciated and used in the design of new objects and rules. Figure 2b shows how flipping the top paddle in Pong exposes its behaviours which comprise robots that ‘`make sound when hit`’ and ‘`when hit send message to score`’. At present, the descriptions of the behaviours are mainly in the form of natural language (i.e. text), although we are currently adding graphical and audio descriptions of a behaviour, so that its functionality becomes evident in its (dynamic) representation.

At the platform level, as we have said, there are robots dealing with the interactions and events. For example, the robot shown in Figure 2c represents one mechanism belonging to the top paddle. The robot has been trained to play the sound `Be Yaw` when hit by a ball: There are two inputs to this program in the box in front of the robot, and the conditions under which it will act are shown in its thought bubble. These initial conditions are set in the training stage but can be modified subsequently. The ‘`hit who?`’ hole of the input box is a dynamic sensor which shows what is currently colliding with the paddle. In order for the robot to be triggered into action (i.e. its condition satisfied), a picture of a ball needs to appear (this will happen when a ball collides with the paddle). In the second input box, the only requirement is that some sound is present. In figure 2c, the '`hit who?`' sensor is black showing that there is no current collision. However, when the paddle is hit by a ball, all the robot’s conditions will be fulfilled and it will play the sound in the second hole of the input box.

We have found that children are able to understand the principles of programming by example, the role of input boxes, the generalisation of conditions etc. However, we have found the robots most useful when used as part of the superstructural level, that is when embedded in behaviours used for carrying out a specific task, such as the `‘make sound when hit’` behaviour. We shall illustrate this conjecture in the following section.

We are unable in this paper to provide further details of either the platform or the playground/microworlds we are building: these and details of the research objectives can be found at http://www.ioe.ac.uk/playground. Instead, we will illustrate our general line of reasoning by reference to a case study of two children interacting in out Pong microworld.

The relationship between platform and superstructure

Two girls, Rachel and Heather, both aged 8, started with a two-player Pong game similar to the one in Figure 2a. One of them used the SHIFT and CTRL keys to control the left and right movement of the top paddle, while the other (they took turns) used the mouse to move the bottom paddle. The ball bounced around and the girls each tried to hit it with their paddle. The score (bottom right hand corner) increased by 10 points whenever the top paddle hit the ball and there was also a `‘be yaw’` noise every time the ball hit the top paddle. At this level of playing the game, the mechanisms which drove these actions were largely invisible — but, as we shall see, they were not inaccessible.

At first Rachel and Heather simply treated the game as a closed system during which they noticed that the score was changed by the top paddle only. They invented a new twist — they took turns to play against the clock, trying to get the most points in 30 seconds. However, after a short while they both pronounced the game as 'boring'.

Changes at the superstructural level

Because of the culture we had developed in our classrooms of changing games, the girls began to think about how they could change the game. The simplest changes they could make involved changing colours and sizes of objects at the interface level.

Heather: “Make it more colourful… it’s a bit dark!”

They made the background light blue and the bottom bar brown. (The programming environment allows colour changes to certain objects simply by pointing at them and pressing keys). Interestingly enough, these apparently trivial modifications immediately changed the look and feel of the game, and generated some new suggestions.

Rachel: “[We] could have two scores, one for bottom one for top”

Heather: “…you could have like the paddle as a fish”

Rachel: “I’ve got an idea … Bammer hits the thing down and hits the ball.”

Rachel's idea was that the ball should be changed into a picture of Bammer the mouse, one of the creatures who inhabit the ToonTalk platform (Bammer's jobs include adding numbers, merging pictures and concatenating strings of text). The new object (a picture of Bammer) needed to retain the functionality of the old object (the paddle) but have a different face. This, Heather and Rachel realised, could be achieved by transferring all the behaviours — something the girls knew how to do.

Heather: “I know — you stick the paddle on the back.”

The girls transferred the functionality of the paddle to a picture of Bammer, by turning both over and placing one on the other. This operation at the superstructural level has platform level functionality because of ToonTalk's object oriented design. Similarly the behaviour objects on the back of the paddle transfer their functionality, giving Bammer the right behaviours. This is illustrative of the delicate relationship that has to exist in our playground design: while the platform provides the means to effect the behaviour transfer, it is having the right things in the right place at the superstructural level which enables the children to make use of this functionality.

The two girls could effect other changes at this intermediate level. They changed the ball to a bird and transformed all its functionalities to give the game the appearance shown in Figure 3, in which paddles and ball are replaced by bammers and bird.

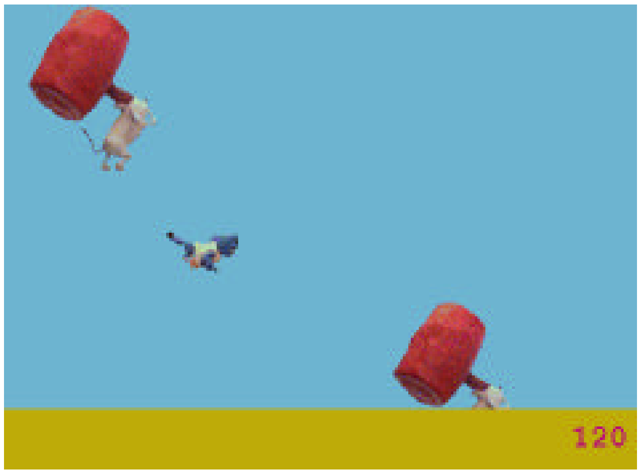


Figure 3

Pong with changed appearance but same functionality

Now the changes to colour stimulated more ideas and supported the girls’ inclination to build an underwater narrative: watching this, we suddenly recalled that Rachel had mentioned earlier that she wanted to change the paddle to a fish!

Rachel: “I know that’s like the sea and he’s [Bammer] running down into it! Cos that’s like there’s a hill and there’s sand going down.”

Heather: “There’s a problem! He’s walking on… the water!”

Rachel: “It doesn’t matter”

The new game was structurally very similar to the original, but to the girls it had suddenly become far from boring! On the contrary, it was now a compelling game, not least because they had made it themselves.

In the next session, we gave the girls some new pictures of fish and sharks to help them in their objective of making their game ‘go underwater’. They made two copies of the shark picture — one for each of the paddles, and discussed further changes: Rachel wanted to have lots of fish bouncing up and down, an idea she had picked up from other children who had made multiple balls in their games. They then started to change the paddles to sharks, an easy enough task involving essentially the same procedure of behaviour transfer as they had used in changing the paddle into Bammer. At this point, they similarly changed the ball to a fish and made copies of it to place underwater (see Figure 4).

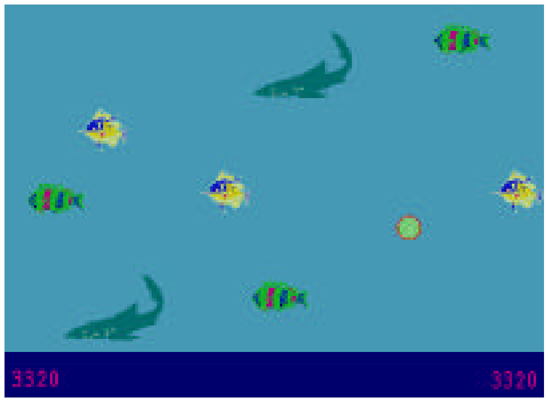


Figure 4

Sharks, fish and two copies of the same score

How much programming were Heather and Rachel doing? It is tempting to say that at this point, they were simply replacing pictures with other pictures. But in the process, they exposed the static representation of the programming platform. While the grain size of their actions was large when programming at the behaviour level, a finer grain size was made visible —the mechanisms were visible even if they remained intact. Heather and Rachel were also familiar with robots and had programmed at this level in some simple cases. They therefore had an idea of what it was that was inside the behaviours and caused them to work.

At the beginning of the session only the top shark scored points — as in the original Pong game. The girls wanted to make their game competitive by adding the same functionality to the bottom paddle, so they could play against each other. They copied the score object on the bottom right of the game so it also appeared on the left (see Figure 4). They also realised that they had to add some functionality to the bottom shark so copied the `‘when hit send message to score’` behaviour on the top shark (as shown in Figure 2b) and put it on the flip side of the bottom shark. These two changes, the first at interface level and the second at the superstructural level, revealed a bug in their thinking about the mechanism of the scoring system and required an intervention at the platform level as we shall describe later.

Changes at the Platform level

The sight of the sharks in an underwater context with fish, provoked the two girls to make more suggestions for changes:

Rachel: “The sharks are the paddles. And if one of those hit the sharks- any of them …”

Heather: “it goes like this … ‘chomp’!”

They wanted a different sound that played whenever the sharks hit the fish. At this point, only the top shark had a sound behaviour (left over from being a paddle in the Pong game). To change the sound they removed the behaviour labelled `‘make sound when hit’` (see Figure 5) and dug down towards the platform level to investigate the mechanism. They found that the robot was trained to play the sound `‘be yaw’` every time its conditions were satisfied. So in order to reconfigure this behaviour they had to remove `‘be yaw’` and replace it with `‘crunch’`. This they managed easily, demonstrating how the two children interacting at superstructural level could, in simple cases when it mattered to them, move into the platform and modify the program.

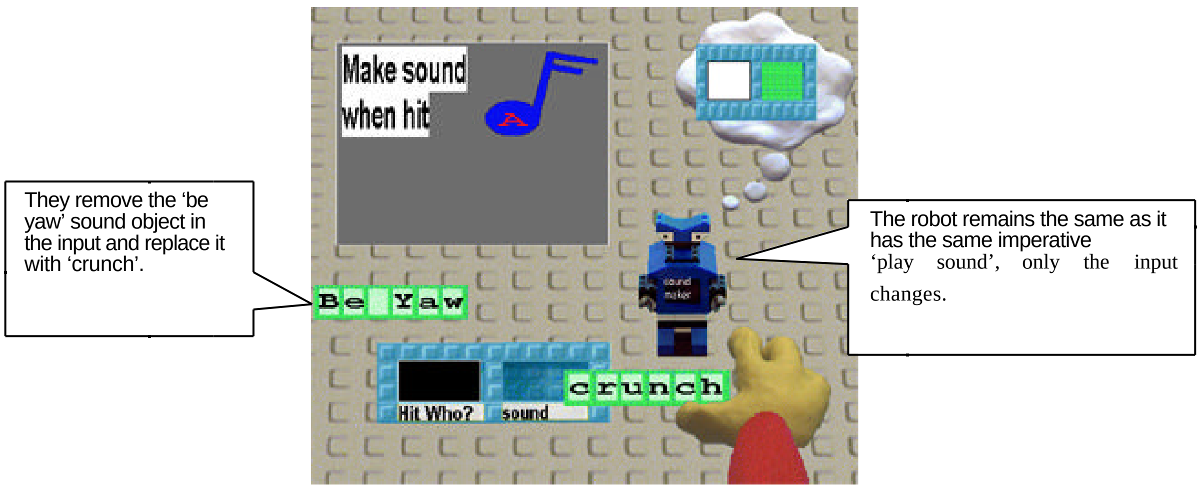

Make sound when hit
Be Yaw
Hit Who?
sound
crunch
sound maker
They remove the ‘be yaw’ sound object in the input and replace it with ‘crunch’.
The robot remains the same as it has the same imperative ‘play sound’, only the input changes.

Figure 5

The robot that makes a sound when hit

While playing this new version, Heather controlled the top shark and Rachel the bottom one. It took them some while to notice that both scores ‘belonged’ to both sharks and were always the same.

Heather: “You see! I said we shouldn’t have copied it!”

Heather knew there was a problem and explained again that they were planning to have a *separate* score for each shark:

Heather: “We want to…change it so that each one has its own score… so each shark has its own score.”

There had been a bug in their thinking, but to fix it they had to find out more about the mechanics of the score system: they needed to see the nature of the connection between the sharks and the scores. This is quite a complicated business. Any object oriented language needs a mechanism by which objects can send messages to each other — this is one of the powerful ideas of object oriented programming. ToonTalk is no exception. But it is different in one important respect — the message passing is instantiated in a concrete metaphor; quite simply, birds take messages to nests, (see Figure 6) so by programming a robot to give something to a bird whose nest is on another object, a message is passed to that object and the bird returns to its starting place. This gives a one-to-one communication channel.

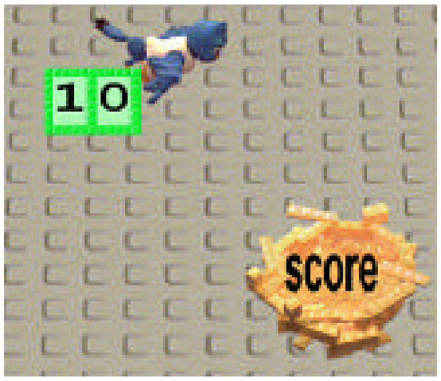


Figure 6

A bird takes a message to its nest: the communication metaphor in ToonTalk

Heather and Rachel took off the top shark and looked again at its behaviours by flipping the picture over. They found the scoring behaviour, `“hit picture send message to score”` and took it out. We explained that the robot had been trained to give the number 10 to the bird each time the shark hit a picture. This 10 then ‘magically’ was added to the score. How did it get there?

In the case of the scoring mechanism, the bird in the `“hit picture send message to score”` behaviours on a shark had a corresponding nest in a behaviour on a score. An understanding of this platform level metaphor would be crucial if the girls were to implement their competitive shark game successfully.

To make the platform visible in a dynamic way, we suggested that the girls took off one score and put it next to the game. The game worked as before, until someone scored. At this point, the score was incremented: but the mechanism of the increment had now become visible — when either shark was hit a bird flew out and delivered the message ‘10’ to the score, accompanied by dramatic wing-flapping sound effects (all this is part of the animation of the platform).

When the second score was removed and placed next to the game, the visibility of the scoring mechanism was even more dramatic. If the bird has been copied, both birds fly to the same nest giving many to one communication. This essentially is what had happened when the girls copied the `‘when hit send message to score’` behaviour. But if the nest is also copied (as the girls did when they copied the score), the bird makes a copy of itself and the message — two identical messages are delivered, one to each nest (one-to-many). We have tried to illustrate this in Figure 7, which shows two birds emerging from the flip side of the bottom shark after it had been hit by the ball.

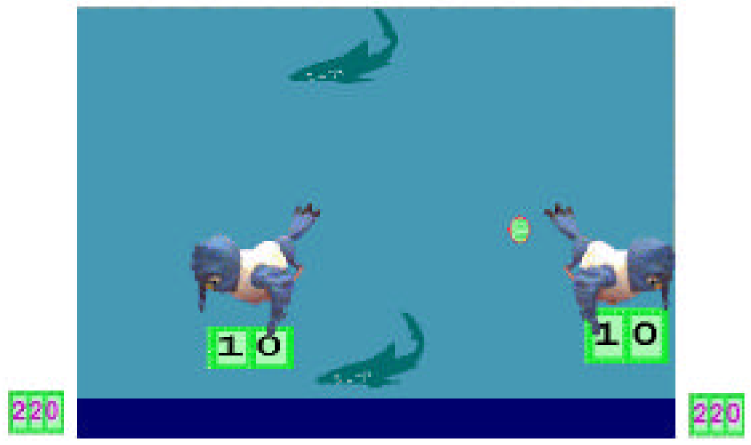


Figure 7

Two identical birds flying to their different

After the children had played their new game and watched the birds delivering their scores, we decided to check if the metaphor was transparent:

> Researcher: “What do you think is going to happen, Heather, when we start the game?”
>
> Heather: “Both sharks are going to have numbers coming out of them.”
>
> Researcher: “Where are they going to go to?”
>
> Rachel: “Into the Points.”

Now the girls could see two birds fly out each time a shark was hit, one bird flying to each score. Although still playing the game, a change at the superstructure level (removing the scores) had led to an exposure of the platform, an interplay between superstructural and platform levels. At the simplest level, a mechanism built into the platform revealed the bug in the girls’ game design and showed Heather and Rachel why the scores were not responding as they wanted them to.

At a deeper level, revealing the mechanism in this way gave an entry into the bigger ideas of using the generative power of programming. Constructing further insight into the mechanism now means another connection into the scheme of the code that can be used for construction later.

Finally, to complete the description of the evolution of Heather and Rachel's game, we explained that if two separate scores were required we needed to have one bird per shark. We helped them replace the bottom shark-score connection with a new bird-nest pair. Their two-player game was now complete as illustrated in Figure 8.

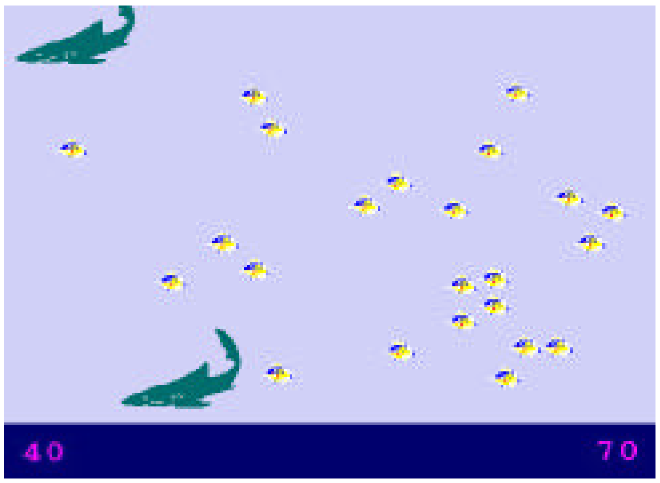


Figure 8

Shark game with two separate scores

Discussion and Conclusions

Our episode is not so divergent from Logo microworlds at their best: the sense of engagement, the need for powerful ideas that arises naturally, the explorability and engagement without sacrificing extensibility and expressive power.

There are differences, which we think delineate an evolution of the microworld idea. Comparing Heather and Rachel's activities with the Mathsticks microworld, the most obvious surface-level change is in the substitution of directly manipulable (animated) elements for text-based programs. This difference is, however, a symptom of evolutionary change. The key difference, we believe, is not only in the new kind of language (certainly important) but also in the interplay between superstructure and platform.

The search for replacements to old kinds of formalism based on textual strings is certainly going to be a defining difference of the new century's mathematical expression (see Kaput, 1994, for a thought-provoking discussion). In general, its potential lies in the broadening of expressive power, to include more immediate, graphical, dynamic and expressive entities and the exposure of relationships between them. In programming terms, therefore, this will allow a richer set of metaphors which provide mappings from abstract computational entities and actions to the concrete world of objects, sounds and even gestures.

Using ToonTalk as a platform has shown us that even quite young children can recognise that they can build objects from scratch (by training robots), combine, generalise and debug. More generally, they can come to think about mechanisms — how things work, why they work, how they can be rebuilt. In the twenty-first century, where the opportunity to strip down working systems with spanners and screwdrivers are much more limited than they were, we will need to consider virtual alternatives, ways for children to take things to pieces, look at what makes them tick, and put them back together (see Noss, 1998, for a discussion of the implications of this view for mathematical learning).

There are also prices to be paid. While our case study illustrates the expressive power of a non-textual programming language, and the opportunities it affords for building superstructures which are appealing and powerful, the absence of a textual description frequently renders the programming to be cumbersome and time-consuming for designers and seriously limits communicability to peers and teachers. This latter issue resonates with the heated debates over the advantages and disadvantages of direct manipulation and text-based interfaces for the learning of mathematics (see, for example, diSessa, Hoyles, & Noss, 1995 for a report of one such debate in the context of dynamic geometry): proponents of the former point to a sense of engagement developed with screen objects; advocates of the latter stress the importance of a language for description, reflection and communication. Perhaps our criticism at that time of direct manipulation missed an essential point — that point and click is precisely the right mechanism for building an expression of the relationships, but precisely the wrong one for reflecting on it and communicating it. There is a duality between expression and reflection, and we must find new ways to play one off against the other. We are currently undertaking a more controlled comparison between ToonTalk-based and Logo-based playgrounds, which will allow us to explore the relative strengths and weaknesses of the two approaches, hopefully leading us to design more successfully in both.

Our superstructural level of behaviours seem to have hit on a way to offer young students some of the power and manipulability of programming and, at the same time, some awareness of the mechanisms of the platform. They are of about the correct grain-size for children: But they also open a window on to deeper issues. The different representations of the behaviours afford a means to think about a rule, what makes it true, and the limits of its validity (e.g. is A hits B the same as B hits A?). In our case study we described the simple rule '`make sound when hit`' which later became transformed into the more complicated `'if hit, score 10 points'`. Our contention is that by looking at how these mechanisms work, and changing them, children are better able to be appreciate inference and conditionality and even make these relationships explicit.

In retrospect, it seems that these metaphors of grain size, levels and permeability between levels are crucial for incremental learning. Perhaps this is the reason why there are not many examples of computational worlds which afford interesting and creative directions for children to learn mathematical ideas *and* which provide an entrée into the world of formal systems which child-programming had always claimed to offer: the set of tools and metaphors appropriate for navigating around at the interface level were *not* functional below that level — to get below, one had to enter a new world of arcane (usually textual) difficulty. Reflecting on our earlier position, we see that we identified platform with text. Now we can explore how new platforms offer new opportunities which preserve what is essential about microworlds, but which increase the functionality at the programming level for learners.

Our purpose is not to claim that we have found a 'solution'. In any case, there are new difficulties emerging: we currently have no mechanism for abstraction — which is combining a collection of programs into a new program and naming this. And the directly manipulable animated elements of programming robots come at the price of no sensible editing mechanism. The case of Heather and Rachel is, however, indicative of possible futures for microworld design. Exposing the birds to reveal the mechanism of message passing feels like an example of a more general possibility: ought we not to be thinking about building systems more generally whose mechanisms are visible and accessible at some level? Mathsticks showed us the importance of linking actions, visual and symbolic representations whilst maintaining the symbolic as a programming tool. It pointed to the importance of permeability between levels of mechanism. Our hope is that our playgrounds are taking a further step in this direction.

Acknowledgement

We wish to thank Ken Kahn for his inspiration in game development and Miki Graham for her assistance in data analysis.

References

Abelson Harold, Gerald Jay Sussman and Julie Sussman (1985) *Structure and interpretation of computer programs* Cambridge, Mass.: MIT press.

Brousseau, G. (1984). The Crucial Role of the Didactical Contract in the Analysis and Construction of Situations in Teaching and Learning Mathematics. In H. Steiner (Ed.), *Theory of Mathematics Education* (Vol. 54, pp. 110-119). Bielefeld: Institut für Didaktik der Mathematik (IDM), Universität Bielefeld.

Clements, D. H. & Sarama, J. (1995). Design of a Logo environment for elementary geometry. *Journal of Mathematical Behaviour*, 14, 381-398.

Clements, D. H., Battista, M. T., Sarama, J. & Swaminathan, S. (1996). Development of turn and turn measurement concepts in a computer-based instructional unit. *Educational Studies in Mathematics*, 30, 313-337.

diSessa, A. & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, 29/9, 859-868.

diSessa, A. (1985). A Principled Design for an integrated Computational Environment. *Human-Computer Interaction*, 1, 1-47.

diSessa, C., Hoyles, C. & Noss, R. (1995). *Computers and Exploratory Learning*. NATO ASI Series, Subseries F, 146. Heidelberg: Springer-Verlag.

Edwards, L. D. (1992). A Logo Microworld for Transformation Geometry. In L. Hoyles & R. Noss (Eds.), *Learning Mathematics and Logo* . Cambridge, Massachusetts: MIT Press.

Edwards, L. D. (1995). Microworlds as Representations. In A. A. diSessa, C. Hoyles & R. Noss (eds) *Computers and Exploratory Learning*, 127-154. Berlin/Heidelberg: Springer-Verlag.

Eisenberg, M. (1995). Creating Software Applications for Children: Some Thoughts About Design. In A. A. diSessa, C. Hoyles & R. Noss (eds) *Computers and Exploratory Learning*. Berlin/Heidelberg: Springer-Verlag, pp.175-196.

Harel, I. (1988). *Software design for learning: children's constructions of meanings for fractions and logo programming*. Unpublished doctoral dissertation. Cambridge MA: MIT Laboratory.

Healy, L. & Hoyles, C. (1999). Visual and Symbolic Reasoning in Mathematics: Making connections with computers? *Mathematical Thinking and Learning*, 1, 1, 59-84.

Hoyles, C., & Noss, R. (1987a). Children Working in a Structured Logo Environment: From Doing to Understanding. *Recherches en Didactique des Mathématiques*, *8*(1.2), 131-174.

Hoyles, C., & Noss, R. (1987b). Synthesising Mathematical Conceptions and their Formalisation through the Construction of a Logo-based School Mathematics Curriculum. *International Journal of Mathematics Education in Science and Technology*, *18*(4, July/August), 581-595.

Hoyles, C., & Sutherland, R. (1989). *Logo Mathematics in the Classroom*. London: Routledge.

Hoyles, C. & Noss, R. (1996). *Windows on Mathematical Meanings: Learning Cultures and Computers*. Pp. 275. Netherlands: Kluwer Academic Press.

Hoyles, C. (1993). Microworlds/Schoolworlds: The Transformation of an Innovation. In C. Keitel & K Ruthven (eds) *Learning From Computers: Mathematics Education and Technology*, 1-17. Berlin/Heidelberg: Springer-Verlag, pp. 1-17.

Kafai, Y.B. (1995). *Minds in play: Computer game design as a context for children's learning.* Lawrence Erlbaum Associates.

Kahn, K. (1999). Helping children learn hard things: computer programming with familiar objects and activities *in* Druin, A. (ed) *The design of children's technology*. San Francisco: Morgan Kaufman Publishers Inc. 223-241.

Kalas, I. (1997) Thomas the Clown. University of Comenius, Slovak Republic.

Kaput, J. J. & Roschelle, J. (1999). The Mathematics of Change and Variation from a Millennial Perspective: New Content, New Context. In C. Hoyles, C. Morgan & G. Woodhouse (eds) *Rethinking the Mathematics Curriculum.* London: Falmer Press, Chapter 12, Section 2, pp. 155-170.

Kaput, J. (1994). Democratizing Access to Calculus: New Routes using Old Roots. In A. Schoenfield (ed) Mathematical Thinking and Problem Solving, Erlbaum, Hillsdale, NJ, pp. 77-156

Noss R. (1998). New Numeracies for a technological culture. *For the Learning of Mathematics*, 18, 2, 2-12. 1998

Noss, R. and Hoyles, C. (1996) *Windows on Mathematical Meanings: Learning Culture and Computers.* Dordrecht: Kluwer.

Noss, R., Hoyles, C. & Healy, L. (1997). The Construction of Mathematical Meanings: Connecting the Visual with the Symbolic. *Educational Studies in Mathematics*, Special Edition, 33, 2, 203-233.

Papert, S. (1991). Situating Constructionism. In I. Harel & S. Papert (eds) *Constructionism.* New Jerseyt: Ablex Publisshing Corporation. 1-12.

Thompson, P. (1992). Notations, Conventions, and Constraints: Contributions to effective uses of concrete materials in elementary mathematics. *Journal for Research in Mathematics Education*, 23/2, 123-147.

Turcsányi-Szabó, M. (1999). Logo Connections: Some cunning aspects. In R. Nikolov, E. Sendova, I Nikolova & I Derzhanski (eds.) *Proceedings of the Seventh European Logo Conference*, 80-91. Sofia, Bulgaria.