ABSTRACT
The subject of children’s programming has long been a vexed and controversial one in the field of educational technology. Debates in this area have typically focused on issues such as how to create a child-friendly programming language; or whether children can learn particular topics (e.g., recursion) in programming; or indeed, whether it is worthwhile for children to encounter programming at all. For the most part, these debates have taken place against an implicit background of assumptions about what children’s programming looks like—namely, an activity focused on creating effects on a desktop screen or, occasionally, robotic toy. This paper argues that the cultural and anthropological contexts of children’s programming are now poised to change: that new programming materials, physical settings, and unorthodox display surfaces are likely to shift the nature of the children’s programming debate in profound ways, and to make programming a far more informal, approachable, and natural activity than heretofore. We illustrate this argument with projects underway in our own research.

Keywords
Children’s programming, paper computing, ambient computing, spherical surface programming

INTRODUCTION
The notion of children’s programming—essentially, of children learning to write working computer programs—has long been a subject of impassioned debate in educational technology. Perhaps it would be more accurate, in fact, to say that there have been multiple debates. One of these has focused on the “why” question: that is, why children should want (or need) to learn programming at all. Broadly speaking, the “advocate” position here [5, 11] has argued that programming is an empowering, creative, and fertile medium for children’s activities; while the “opponent” position has argued variously that programming is unnecessary, overly difficult, or (in the extreme) cognitively harmful for children. [8, 14,18]

Even among the community of children’s-programming advocates, there are smaller-scale but no less ferocious debates. Some of these have focused on appropriate language design: the arguments here involve issues such as how to make it easier for children to create working programs through (e.g.) rule-based programming systems [12], novel editing or compositional tools [6, 16], programming by example systems [17] and so forth. It is fair to characterize these as software-based efforts (and debates): the question is how to create software systems (languages, editors, operating systems) that children will find easier to learn or use, and the merits of the various attempts to do so.

Still other debates have focused on what particular ideas are fundamental (interesting, fertile, essential) to children’s programming. Illustrative questions here might include: Should children learn recursion? (And can they?) Should they learn about procedures and variables; and if so, when? Is “graphical” programming (such as the creation of a digital logic diagram) really programming? Are (e.g.) programmable robots or toys [21] a meaningful introduction to programming?

Although we confess to having strong views of our own on these various questions, the purpose of this paper is not so much to enter into these historical debates directly. Rather, our purpose is to argue that the foundations of these debates—the abiding images of what “programming” looks and feels like as an activity—are poised to change in profound ways over the next decade. These changes in the nature and style of programming are in turn, we believe, likely to alter irrevocably the terms of the longstanding debates sketched above.

There are novel technologies emerging that promise to introduce a “new look”—really, several new looks—to programming. On the one hand, a variety of traditional materials—fabric, paper—can now be employed as the background substrate for programmable artifacts and displays; that is, it is possible to work with programmable materials. In a similar vein, one can devise means of placing small, informal “chunks” of programs within physical environments, where they may be read or executed by mobile computational devices—a notion that we refer to
as _ambient programming_, or, in effect, making extended settings the background against which programs are created. Finally, there are novel types of _display surfaces_ that may be used as the backdrop for relatively unexplored styles of programming, taking advantage of unorthodox geometries and public settings.

The remainder of this paper is an extended discussion of these novel themes in programming, with particular attention to their effects on the longstanding debates concerning children’s programming. The following section will begin by elaborating on these themes (namely: “stuff”, “settings”, and “surfaces”), basing the discussion on current projects undertaken by the authors. The third section will then use these projects, and the themes they exemplify, as the foundation for a discussion of new directions in children’s programming. We close with a brief discussion of the many promising directions for future design and research associated with “new-look children’s programming”.

"NEW-LOOK" PROGRAMMING: STUFF, SETTINGS, AND SURFACES

In this section we briefly elaborate upon the three themes in programming mentioned above. For each of these themes, we illustrate the idea through prototype projects developed in our labs.

**Paper Computing: An Example of Making a Material Programmable**

Traditionally, children’s programming has been limited to a relatively select variety of computational devices. In the early years of personal computing, the assumption was that children would work with desktop computers; over the years, that image has been augmented through the inclusion of mobile and handheld devices, as well as robotic systems such as Lego Mindstorms. [9] Still more recently (and excitingly), computational construction kits have permitted individual programmable pieces to be assembled into larger constructions endowed with interesting behaviors. [15]

These latter developments make use of the fact that computers, and their associated actuators and sensors, can now be made sufficiently small and affordable to allow them to be placed inside child-sized play objects. Making use of the same developments, we can now augment certain types of traditional materials—fabric, paper—so that they can be freely decorated with computational elements. The essential idea here is that through creating a “kit” of small computational pieces (processor, battery, sensors, motors, and so forth) we can add those pieces onto particular materials, allowing them to run programs. A textile artifact—a shirt, a hat, a book cover, a tapestry—can be turned into a working computational display, as we have discussed in earlier publications [cf. 4].

Here, we will describe more recent work (see also [3]) that permits paper surfaces to be used as the backgrounds for running programs. The basic idea behind this notion of _paper computing_ is that we can construct small, flexible pieces as shown in Figure 1; and these pieces may then be easily attached to paper treated with conductive paint to create full-fledged working programs.

![Figure 1 (top): A paper-based Arduino processor.](image1)

![Figure 2 (bottom): A set of paper computing elements to accompany the processor in Figure 1. From top left, clockwise: battery, motor, speaker, switch, and LED.](image2)

Figures 1 and 2 show photographs of the computational elements of our “paper computing” kit. These include a microprocessor (Figure 1), battery, motor, speaker, switch, and LED (Figure 2). All of these elements are created using a technique similar to one that we have used for placing elements onto fabric; here, the elements employ conductive fabric as the basis of electronic connections, and a paper backing. The microprocessor unit is programmable using the popular Arduino development environment [2], and may indeed be reasonably regarded as a paper-based Arduino.
While Figures 1 and 2 depict the individual computational elements, we have not yet shown how these can be used to create paper-based computing. Figures 3 and 4 show how this is done through an example. Here, a variety of elements have been attached to a sturdy sheet of sketchbook paper that has been treated on its opposite side with magnetic paint. (The elements themselves have small magnets on their bottom side to provide a reasonably stable attachment.) On the front side, the paper has been decorated with both conductive paint (the gray areas) and standard decorative paints. By painting the paper with conductive regions, then, one can create a decorated “conductive backdrop” against which the kit pieces can be placed, rather like easily-removable computational “stickers”. In the example shown in Figures 3 and 4, the user has created a working paper program in which she can control light or sound by touching a skin-conductance sensor that, in this instance, has been painted directly onto the page.

In summary, then, by combining several powerful novel materials (conductive fabric, magnetic and conductive paints) with accessible computational elements, we have created a system through which paper sheets may be “decorated” with running programs. The user, in a typical scenario, writes a program using the Arduino system on a desktop machine, downloads it to the microprocessor, paints the background for the running program by hand, on paper; and then places the appropriate kit pieces on the paper to create the running program. This need not be the only scenario for paper-based computing, however: for example, a pre-painted background might be supplied to a child, and the child’s job would be to write different programs to operate on that background. Or, conceivably, the child’s job might be to take an existing program and to create different painted paper backdrops against which that program could be made to run. We will return to these various scenarios in the next section of this paper.

Ambient Programming: Spreading Program Throughout a Physical Setting

Traditionally, one thinks of a “program” as a collection of symbolic (generally textual) instructions stored in computer memory or printed out in paper form. Increasingly, however, it is feasible to use small mobile computational elements as “portable program readers”: devices that read elements of programs that have been “written out” in various forms around one’s physical environment. In effect, one can think of this idea as having program “chunks”—routines, commands, variables, and so forth—spread around one’s setting in forms that are readable by portable devices.

An example from our lab may help to make these ideas more concrete. Consider, then, the small Lego car shown in Figure 5. The car includes a PIC microprocessor, and its undercarriage contains a “program reader”—essentially, a light source that can read the reflectance of the surface underneath the car. Figure 5 not only depicts the car and (at right) a view of the program reader, but it also shows something else of crucial importance in the picture at left: namely, a view of a program instruction (as it happens, a “turn to the left” instruction) written out as a bar code. Thus, if one were to start the car from the position shown in Figure 5, it would roll over (and thus read) a “turn left” instruction, after which it would turn and continue on its way.

The car in Figure 5 (a schematic is also shown in Figure 6) may thus be viewed not simply as a “programmed robot”,

---

1 A video clip showing this example in operation may be found at: www.youtube.com/user/craftvid
but as a device that moves about in accordance with instructions written out on the surface beneath it. By creating a small set of distinct commands that the car can read (Figure 7) and laying them out on the floor, one can in effect “write a program” on the floor of one’s room.

Figure 5. At left (1a), a view of the ambient program-reading car about to encounter a “turn left” pattern on the ground. At right (1b), a view of the underside of the car, showing the program reader itself.

Figure 6. A schematic of the car in Figure 5, showing the placement of the microprocessor, Lego motor, and “program reader” sensor.

Figure 7. Four readable instruction patterns (the actual patterns read by the car are the bars at bottom; the symbols above are included for convenience). From left to right: a left turn, a right turn, a “flip-a-coin” (left-or-right) turn, and a pause.

Figure 8. A simple program, laid out for the car to traverse: a right turn, followed by two left turns, and (finally) a pause.

Figure 9. The program shown schematically in Figure 8 is here "written out" on the floor in front of the actual car. The video in www.youtube.com/user/craftvid shows the car following this program.

Figures 8 and 9 depict a particular program (the same one), both in diagrammatic form and in a still from a video of the working prototype. The car, once started, will first pass over the “turn right” pattern; then two subsequent “turn left” patterns; and finally pauses briefly after passing over the farthest card toward the top of each figure.

The purpose of this particular example, entertaining as it is, is to illustrate a larger theme in programming: namely, that programs can be symbolic artifacts placed in one’s physical environment in all sorts of interesting ways. In our example, the “program” for the car’s behavior is represented as cards spread about the floor; one might imagine other types of scenarios in which a “program” is represented via color patterns on wallpaper, or a sequence
of flashing lights, or a tactile pattern of textures on a wall, to name just a few.

The idea of “ambient programming” suggests that programs can be constructed in informal, moment-to-moment ways. One might alter the “program” shown in Figure 9 by physically messing about with cards upon the floor, changing positions and putting down new cards. Indeed, one can simply draw out (using a felt-tip marker) patterns of the type shown in the figure; so one needn’t have a computer (or keyboard) to “write a program” in this case, but can actually write out a readable sequence of cards by hand.

The key point for now is that ambient programming implies a type of programming activity that looks and feels quite different from the traditional method of desktop composition. Programs may (depending on the example) be placed around a room, drawn by hand, scrawled onto a wall, changed by whistling particular tunes, and so forth. We will return to the ramifications of this idea in the next section.

Programming on a Planetarium Sphere: Making Use of Unusual and/or Public Surfaces for Displays

Traditional children’s programming (particularly graphical programming) has usually been imagined as an oddly solitary and private affair: by assumption, a child writes a program that runs on the computer screen directly in front of her. Occasionally, such programs are shared with teachers or other students, who sit beside the child or lean over her shoulder; but the very design of most computers implicitly leads to a style of activity in which programs are imagined as single-viewer affairs displayed on flat computer screens.

This tradition, too, is rapidly changing in the face of novel techniques for display projection. In our lab we have developed a relatively dramatic example illustrating the move away from the flat, solitary screen: a prototype programming system for children that makes use of a giant spherical display screen at our university’s planetarium. The basic idea behind the system—which is still in development—is that the child may write programs similar in spirit (and syntax) to traditional Logo turtle programs [1], in which a programmable “pen” moves about the screen in accordance with programmed commands. In this case, however, a program command that tells the turtle to move “forward 1 unit”, rather than producing a straight line on a flat screen, now produces a portion of a great circle on the planetarium sphere.

Figures 10 and 11 show the results of running two simple turtle programs on the sphere; one, a program that creates a “spiraling pattern” inward toward a point on the sphere, and the other a pattern of great circles. (Both photos, incidentally, indicate the size of the spherical screen in the display area.)

Figure 10 (top). A multicolored spiral pattern drawn on the planetarium sphere.

Figure 11 (bottom). A pattern of great circles drawn on the sphere. A video of the system in operation may be seen at: www.youtube.com/user/craftvid

Our programming system, again, is just an example of a much larger landscape of possibilities involving novel display devices and projection systems. One might imagine (e.g.) displaying programs on a ceiling dome, or on the walls of a child’s room, or on the exterior of an office building. Displays may, in certain circumstances, be large public surfaces; or they might be spaces shared between multiple programmers (e.g., several children might program the planetarium sphere at once); or they may be made visible worldwide over the internet (e.g., the sphere might be seen by the use of a webcam placed in the planetarium); or they might employ a variety of still-early techniques for three-dimensional or volumetric display. [7]

The upshot of all these developments is that children’s programming can soon be associated with an endless variety of display surfaces at multiple scales and exhibiting multiple levels of private or public viewing.

THE CHANGING LANDSCAPE OF CHILDREN’S PROGRAMMING

The previous section described several types of developments in programming—programmable materials,
ambient programming, and novel display and projection capabilities—what we believe are collectively poised to radically alter the traditional appearance of children’s programming, and the traditional debates around the subject. In this section we develop these arguments further, using the working prototypes of the previous section as occasional examples-to-think-with.

The Anthropological Context, I: What Programming Looks Like

One of the interesting aspects of the many debates about children’s programming—and particularly the question of whether children should program at all—is that the terms of the debates often hinge on what can only be called aesthetic considerations. Repeatedly, one sees objections to the very look of programming—the glowing screen, the absence of sensory stimulation, the emphasis on abstraction as opposed to “real-world” experience or tangible materials. Consider, for example, the following passages taken from prominent books written by educational computing “skeptics”:

> From birth, sensory areas in the back of the brain refine their ability to perform basic functions effortlessly: listening, looking, touching, and moving…. This “intersensory integration” is critical for good learning, and it takes lots of practice. Open-ended computer use—such as a drawing program—offers some combining of sensory abilities but differs qualitatively from nature’s programming of whole-body, three-dimensional sensory experience. [8, p. 208]

> Logo grants the children control over an experimental “microworld” in which to do their programming; but the microworld is not the full terrain of the human imagination. It is, at last, a two-dimensional computer screen which can only display the capacities of the program. Logo has a repertory of just so many colors, so many shapes…. It is well suited to geometrical play but not to fantasy that oversteps those narrow boundaries…. I found myself haunted by the image of the prisoner who has been granted complete freedom to roam the “microworld” called jail: “Stay inside the wall, follow the rules, and you can do whatever you want.” [14, p. 75]

> There is much to say about these arguments, both in support and opposition. For our purposes, however, the crucial point is that such observations assume a portrait of programming that is likely to become less and less representative over the next decade. Programming is not (or need not be) a physically impoverished activity: one could “program” the ambient-programming car of the previous section through strenuous physical activity, by running about a room to slap down “instructions” ahead of the oncoming car. One can combine the feel of paints and papercrafts with programming, as in the paper computing example shown earlier. One is not at all limited to “two-dimensional screens” (or at least flat ones) as indicated by our discussion of spherical (and ultimately volumetric) programming displays.

The aesthetics of programming are poised to change. While it is unlikely that “classical” programming will (or, in our view, should) disappear, that style of programming will ultimately be seen as one spot in a much larger landscape of programming styles—physical, tactile, sensually rich, athletically demanding, and so forth. And as “programming” comes to suggest a different type of activity, the stereotyped portrait of “the programmer” likewise evolves, as the next subsection will argue.

The Anthropological Context, II: What the Programmer Looks Like

One of the most famous descriptions of the stereotyped “hacker”—an unkind, though not completely inaccurate passage—was written by the late Joseph Weizenbaum over thirty years ago:

> Wherever computer centers have become established…bright young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the buttons and keys on which their attention seems to be as riveted as a gambler’s on the rolling dice…. Their rumpled clothes, their unwashed and unshaven faces, and their uncombed hair all testify that they are oblivious to their bodies and to the world in which they move. [20, p. 116]

Weizenbaum’s portrait was an unfair stereotype when it was written—at best only partially true—and it remains so today, as anyone familiar with computer programmers and students could attest. (Indeed, even if one accepts the portrait at face value, one might respond that it is somewhat refreshing to see any young people so fully mesmerized by an intellectual activity—it happens, after all, so rarely.) Nonetheless, there is sufficient truth to Weizenbaum’s image that it, or subsequent incarnations of it, have taken hold in the culture at large. A crucial theme in this portrait is the notion of obsession: one cannot be a true programmer, according to this image, without being consumed by the activity to the exclusion of everything else. (A similar, though more sympathetic, portrait of the “hacker” image and ethos is discussed in Chapter 6 of Turkle’s book The Second Self: [19])

This image—however true or false it may ultimately be—has, in our experience, done tremendous damage to the popularization of programming among young people (particularly adolescents). Again, the issues are aesthetic: programming (on this view) is antisocial, sensorily and physically deprived, overwhelmingly male in composition, and so forth. And again, whatever truth this portrait may
have had in the past, it is likely to have progressively less so in the near future.

One can be, for example, a dedicated fabric artist or papercraft designer and be an avid programmer: this activity may develop into an obsession, but it is one that looks very different from Weizenbaum’s portrait. That is, the material dimension of programmable materials naturally leads to an integration of the traditional styles of artistic dedication and intensive programming. In a similar vein, one can participate in ambient programming through occasional, informal means—adding a bit of program to one’s environment here or there, altering the symbols or meaning of a small chunk of program text—without making this activity the sole focus of one’s day. Ambient programming, as an idea, is indeed predicated on the notion of informal, occasional, exuberant-but-not-obsessive participation. Just as programming is poised to change, then, so is the accompanying portrait of the “programmer type”. As before, we do not believe or hope that the older style of programming (or programmer) will disappear; but it will become far less archetypal.

The Anthropological Context, III: Programming as Public and Shared Activity
An implicit element of the traditional portrait of programming is that it is a relatively solitary activity, best framed as an intense intellectual encounter between the programmer and machine. There is a strong potential value, in our view, to just this sort of activity, requiring patience, concentration, and what might be called an almost zealous level of meditation. It is also true that this portrait does have a social, human dimension: to the “hackers” described in [19], for example, the support community of their peers is a source of encouragement and comfort.

Nonetheless, this is not the only social style compatible with “new-look” programming. In the traditional portrait, there is an element missing—one in which programming is done almost in the spirit of mass performance, like the activity of many dancers on a ballroom floor. While the results of programs may run in public settings, the activity of programming itself is rarely visible. Consider, in contrast, the activity of laying out the components of a program for the ambient-programming car discussed in the previous section. This is a type of “open programming”, in which the creative thought process is woven into visible physical activity, almost as it might be with a jazz musician in the act of improvising. Likewise, the programmer of the giant spherical display might well devise, interactively and in real time, portions of code that can run in public, for all to see; or multiple programmers might use the same public display device in a kind of “conversational program”, like multiple painters working on a mural.

Programming Languages and Their Elements
The reader might have noticed, to this point, that there has been very little discussion in this paper of what is often thought to be the most contentious question in children’s programming: namely, how to devise a suitable language. Among those who accept the value of programming as an activity, the fiercest debates have been over issues like whether computer languages (individually or collectively) are sufficiently learnable, expressive, engaging, and so forth. Historically, these debates have even manifested themselves in what, to outsiders, might seem like squabbles over relatively low-level details—e.g., whether a language should encourage the use of recursion, or the syntax of variables.

Such issues are not going to go away; in any imagined scenario for children’s programming, the child will, ultimately, have to work in some symbolic medium. Still, in some respects at least, we believe that the overall question of children’s language design will itself have to be revisited in the context of “new-look” programming. As a first approximation, we do not feel that the question of language design is the overwhelmingly central question that it has traditionally been assumed to be. Indeed, in two of the projects mentioned earlier, we have arguably finessed the language question: the paper computing kit makes use of the Arduino language (hardly a revolutionary language design), while the spherical computing system is essentially based on the traditional Logo language. For our own projects, we are likely to return to the question of children’s language design, but we do not feel that is the alpha-and-omega question of children’s programming. To us, the question of the settings, purposes, and products of children’s programming represent issues of at least equal importance to the choice and design of the programming language itself.

That said, there are a few other observations to be made in this brief discussion of the subject. First, for many types of projects, the style and setting of the program lead naturally to certain types of programs (and program constructs) being highlighted. For example, in many paper computing projects (and in many embedded-computing projects more generally, such as those involving robotic toys), a typical program will tend to be rather short, often involving reading a sensor input and responding to that input with a relatively straightforward actuator output. In other words, for this type of program (though again, not for all) the need for large numbers of procedures, complex control structures, large data sets, and so forth is minimized. Conversely, other sorts of programs involving multiple embedded processors within some material artifact might necessitate an introduction to ideas of parallelism and distributed computing—ideas which are often imagined to be too advanced for children’s work (though see [13] for a classic discussion of ideas of parallelism in children’s programming).

Ambient programming leads to still other considerations in the development of children’s programming. Here, we might imagine novel types of symbolic elements to represent programming constructs strewn about a physical
setting. The bar codes of the example in the previous section are one, rather mild, example of an unorthodox symbolic programming “code”. Other possibilities might include patterns of flashing lights, color codes, codes embedded in sound, and so forth. In other words, in the world of ambient programming, what counts as a symbolic “program”, and what counts as a programming “language”, might be subject to radical reconsideration. This is a subject for a longer discussion at a later time.

Programming Within the Larger Intellectual Landscape

Finally, it is worth mentioning the potential role of “new-look” programming in the larger debates about the purpose and role of programming in children’s lives. Traditionally, advocates of children’s programming have argued (and we agree) that programming can be a profound addition to children’s creative repertoire, particularly in many areas of mathematical, scientific, and artistic endeavor. In our view, the growing presence of such phenomena as material programming, ambient programming, and expanded display surfaces will only strengthen this argument. Craft and materials-based projects will incorporate programming ideas in novel types of children’s art and engineering; ambient programs will allow children to weave easily expressed programming ideas into their day-to-day projects; novel display surfaces will enable children to create new types of public dynamic artwork and simulations. In short, we see these developments as leading to a much more visible presence of programming ideas in children’s lives both in and out of the classroom—and to a much more natural and varied presence of those ideas in children’s intellectual lives.

FUTURE RESEARCH: A (VERY BRIEF) SAMPLER OF POSSIBILITIES

Each of the individual themes discussed in this paper can serve as a foundation for future research and design. Materials programming might be extended to explorations in adding computational capabilities to (e.g.) ceramics, plastics, or metal artifacts; ambient programming might be extended with work on novel types of mobile program readers and data representations (see, for instance, the pioneering work by Druin and her colleagues in this area [10]); display efforts might explore the uses of handheld projectors for small, locally customized displays, or volumetric (3D) displays, in children’s programming. More broadly, these efforts collectively represent what we hope will be a revived, energized, and expanded view of children’s programming—one done more with an eye toward the anthropological contexts of programming (the reasons children program, the settings in which they do so, the relationships they form by doing so) than simply the curricular aspects alone.

ACKNOWLEDGMENTS

The work described in this paper has been funded in part by the National Science Foundation under grants EIA-0326054 and REC0125363, and by the University of Colorado Outreach Program. We wish to thank Mark Gross, Gerhard Fischer, and Clayton Lewis for helpful conversations.

REFERENCES

2. Arduino website: www.arduino.cc
9. Lego Mindstorms website: mindstorms.lego.com
16. Scratch Language website: scratch.mit.edu
17. Stagecast Creator website: www.stagecast.com