# Runtime Revolution: An easy to learn programming software for educators

The public has high hopes for computer technologies in the public schools. An issue to be addressed is how best to utilize this investment. Computer technologies utilizing multiple media and interactivity are beneficial in teaching and learning. The production and sharing of learning objects demands that the educators possess the ability to create them. Numerous studies[1] reveal that most instructors do not venture beyond such "basic" computer usage tasks as email, the web and word processing. This article briefly examines novice/non-programmers and the multimedia authoring program Revolution concentrating on its programming language, visualization, and the possibility for meaningful module production/reuse.

## Introduction

Computer scientist and educator Donald Norman (Norman, 1996) has noted that computer technologies, especially those utilizing multiple media and interactivity, are beneficial in three key areas of human learning, namely, engagement, effectiveness and motivation. Additionally, the excitement within the educational community with respect to the production and sharing of learning objects demands that the average educator possess the ability to create meaningful learning objects. This demand, however, presents the problem of learning the art of computer programming, an activity which suggests spending a good deal of time learning a cryptic computer language. This is not necessarily the case, however.

Some of you may remember *Hypercard*, an application program and programming development environment produced by Apple Computer in 1987. For those who never had the opportunity to use *Hypercard,* it was a graphically-oriented development environment coupled with not only an easily comprehensible, English language-like scripting language, but also a set of pre-built modules that the user could either utilize as-is out of the box *or* customize for a particular use. As such, it was in a way an embodiment of Ben Schneiderman's very articulation of the direct manipulation interaction paradigm that made graphical user interfaces such as Apple's Macintosh operating system easy to both learn and use by ordinary individuals: *Hypercard* provided (a) visual objects that (b) could be easily manipulated in (c) a series of rapid, incremental and reversible actions that resulted in the crucial feedback necessary for learning to use the program. As such, it was often used for rapid application development and quickly developed a reputation as a development environment which empowered ordinary persons to create their own software without enrolling in a computer science degree program.

[1] Green, K.C. (2003). The 2003 Campus Computing Survey, http://www.campuscomputing.net (accessedApril 2, 2007).

*Hypercard* was not, however, without its disadvantages. Color support was weak; the only graphic file format directly supported was Apple's proprietary PICT; and standalone executable files could run only on Apple machines. These disadvantages, Apple's since-discontinuance of the program, the introduction of Microsoft's *PowerPoint* presentation software, and the development and popularity of the world wide web have seemingly introduced a shift among educators away from the production of interactive, desktop-based digital educational wares in favor of static web pages, *PowerPoint* presentations, and a few interactive digital modules produced by those increasingly fewer educators with the skills to embrace the hypermedia creation tools left on the market. Or, as Moser (2005) put it, "roughly ten years into the e-learning age, educational technology has made only modest inroads into changing teaching."[2] The question, therefore, is whether there exists a market for a *Hypercard*-like product or whether its paradigm-heyday should remain an education historical footnote.

## The Rationale for Interactive Learning Modules and Problems with their Production

David Staley (2004) has suggested that discussions involving the implementation of digital technologies in the classroom begin by asking the question *Why is this technology here?*[3] "Ubiquity of technology is an insufficient rationale for inclusion in a classroom," he advises.[4] Instead, as Claudia Perry (2004)[5] suggests, digital technologies in the classroom are best utilized when they "incorporate interactivity, self-paced and self-directed learning options... and varied presentations of information (text, audio, visuals, multimedia, simulation)."[6]

These qualities tend to be strikingly absent from most webpages and *PowerPoint* presentations, the former due to the need to learn something akin to a formal structured programming language, the latter by definition of the abilities of the software itself. Moreover, merely adding *and simply reading* a basic *PowerPoint* presentation, while technically meeting the average person's definition of using computer technologies in the classroom, is a tragic disservice to the public's investment in digital educational technologies: they lack the interactivity that makes computers in the classroom a useful pedagogical addition. Even students themselves find such presentations only marginally useful: one small study performed at the University of Washington suggests that students desire that such presentations be utilized more effectively,[7] which does little to suggest that said presentations are an effective learning tool in terms of engaging student interest.

[2] Moser, F. (2007), Faculty Adoption of Educational Technology. *Educause Quarterly (1)2007*, p. 66. This particular article deals with faculty adoption of digital tools for learning in higher education.

[3] D. Staley (2004), Adopting Digital Technologies in the Classroom: 10 Assessment Questions, *Educause Quarterly (3)2004,* pp. 20-26.

[4] Ibid, p. 23.

[5] C. Perry (2004), Information Technology and the Curriculum: A Status Report. *Educause Quarterly (4)2004*, pp. 28-37

[6] Ibid, p. 30.

[7] Ibid, p. 31. Perry citing K. Gustafson, The Impact of Technologies on Learning, *Planning for Higher Education, 33(2),* pp. 37-43.

One reason why such presentations are perhaps something less than an optimal example of computers in the classroom can be glimpsed in an observation Kendall Whitehouse (2005) made regarding the paradoxical earlier failure of television in the classroom as opposed to the well-known successes of educational television programs like *Sesame Street* and programs provided on the History Channel: "They do not use television to replicate the experience of the classroom. They provide a different type of learning, driven by the particular characteristics of the medium."[8] Indeed, a large part of what distinguishes computer software from, say, a book, is its interactive nature. More pointedly, as Marshall McLuhan put it, "the medium is the message,"[9] and, in the case of the computer and its related technologies, the medium/message is interactivity. Thus, when designing computer tools for learning, it is imperative that one capitalizes on the computer's ability to provide interactivity, not only because it assists with engaging the learner but also because it leverages the characteristics of the computer's medium.

Armed with this insight, one might wonder why educators sometimes place such emphasis on *PowerPoint* presentations as an effective educational tool given the wide variety of digital creation tools that currently exist. The Wharton School (University of Pennsylvania) has developed and implemented what they feel is a fairly successful usage of web-based gameplay learning and simulation in the teaching of business and economic concepts.[10] The very nature of game play and simulation must needs require interaction with the end user. To achieve this need, the Wharton School utilizes a full-time IT staff to create these interactive learning modules in consultation with its faculty, and uses industry-standard development tools, including Macromedia's (now Adobe) *ColdFusion MX, Flash* and *Dreamweaver,* and Microsoft's *SQL Server*.[11]

The fact that the U of P requires a full-time IT staff to develop these programs as opposed to the faculty members themselves developing them is telling: industry standard tools tend not to be embraced by the average instructor. There exist entire certification processes for learning *SQL*; providing interactivity in *Flash* requires learning the close cousin of the *JavaScript* scripting language which was primarily designed as a "lite" version of the formal programming language Java (and which itself is the topic of numerous university-level classes and programs). *Dreamweaver* likewise requires the use of a relatively unintuitive scripting language or environment in order to provide web-based interactivity (such as asp or php-based solutions).

These, clearly, then, are not exemplary of programming solutions 'for the rest of us.' The reasons are well-known to those who study the psychology of the novice or non-programmer. Novice/non-programmers, not unlike the majority of the consumer (i.e., non-programmer) population are mystified by the computer's operations and see them largely as a mysterious black box (DuBoulay, as cited by Mayer, 1981).[12] Adding to this obstacle is the fact that computer languages have a distinct epistemology of

---

[8] Whitehouse, K. (2005). Web-Enabled Simulations: Exploring the Learning Process, *Educause Quarterly 2005(3)*, p. 20.

[9] McLuhan, M. and Fiore, Q. (1967), *The Medium is the Message: an inventory of effects.*

[10] Ibid.

[11] Ibid, p. 23.

[12] Mayer, R. (1981). The Psychology of how novices learn computer programming. *ACM Computing Surveys 13(1),* 121-142.

computer data structures and algorithms that differ radically from the epistemology of natural human languages (Smith, Cypher, & Schmucker, 1996).[13] To put it briefly, as Solloway (1983) has noted, "even at a simple level, [programming] is a difficult activity to learn"[14] (one must understand that, in 1983, programming involved a strictly command-line environment). Indeed, it is estimated that fewer than one percent of computer users have the ability to engage in programming activities.[15]

The nature of the programming language itself presents a major obstacle for the novice/non-programmer, and it has been found that this audience has difficulty in parsing pseudocode [that is, natural-language] representations of the programming problem into the development environment's syntactical language (Green, 2001; Barr, Holden, Philipps, D. & Greening, 1999).[16] Additionally, programming language reference materials, especially language dictionaries, are of little assistance in that resources targetting the programming community tend to present code examples in isolated, small examples which are focused on a single concept or a single language construct.[17] This isolation removes code from context as well as code from feedback, and thus further fragments the programming/learning to program process by resulting in a critical lack of understanding of how intentions become pseudocode, and how pseudocode is translated into valid but rigidly syntactic computer language. Hence, both the traditional programming references as well as the nature of the programming language itself can result in the novice/non-programmer failing to develop an understanding of how the subcomponents of computer programs relate to one another and to the program's overall objectives.[18]

As an example, to set the label of a button, a traditional [object oriented] programming language would require the user to write

```
firstButton= new button

firtButton.label = "push me"
```

Whereas an authoring system/programming environment which utilized a language with natural-language properties would require the user to write

```
set the label of button 1 to "push me"
```

Two things should be immediately apparent when comparing the two sets of code:

---

13 Smith, D., Cypher, A. & Schmucker, K. (1996). Making programming easier for children. *ACM Interactions 3(5),* 58-68.

14 Bonar, J. & Soloway, E. (1983). Uncovering principles of n ovice programming. *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 10,* 10-13.

15 Smith, D., Cypher, A. & Schmucker, K., op cit.

16 Green, T. (2001). Instructions and descriptions: some cognitive aspects of programming and similar activities. *Proceedings of the working conference on advanced visual interfaces*, 21-29.; Barr, M., Holden, S., Philipps, D. & Greening, T. (1999). An exploration of novice programming errors in an object-oriented environment. *SIGCSE Bulletin 31(4)*, 42-46.

17 Neal, L. (1989). A system for example-based programming. *CHI'89 Proceedings*, 63-68.

18 Guzdial, M. (1995). Centralized mindset: a student problem with object-oriented programming. *SIGCSE'95, 182-185.*

first, the traditional language required two lines of code whereas the natural-language language required only one, and, second, the natural-language code style mimics the way humans think (and, additionally, would be rather similar if not altogether identical to the pseudocode describing said action). This short example is indicative of the power of a non-traditional programming language using natural-language properties to capitalize on the novice/non-programmer learner's innate capacity for natural human language as an anchoring or scaffolding strategy in learning a development environment's programming language.

A second large obstacle facing the novice/non-programmer is the absence in many development environments to provide visual or otherwise obvious "one-to-one mapping between what they write or see (in the code) and what the system is doing as a result"[19] in the midst of a programming activity. This suggests that visual and visually-oriented programming environments such as *Hypercard*, *Flash*, *Dreamweaver*, *VisualBasic* and various iconic-flow programs would lend themselves well to the non/novice programmer by providing visual and metaphoric mappings or models of the programming environment which link code and output.[20] A wealth of literature exists documenting the importance of appropriate metaphor, visualization and natural-language environments in making programming an embraceable opportunity for the novice/non-programmer and will not be repeated here. The question here is whether or not Revolution is an example of that desired category of programming environments.

**What is Revolution?**

Revolution is very similar to Apple's Hypercard in that it is an object and media-rich, event-driven development environment which utilizes a natural language-like scripting language at its core. As in *Hypercard,* the metaphor in use for application development is the one of a 'stack' of cards. A 'stack' is a series of cards presented in a window. A stack can contain other stacks; additionally, two or more stacks can be deployed simultaneously in two separate windows. If the "stack-card" metaphor seems a bit dated, feel free to think of a "stack" as an application, and a "card" as a particular screen or window view (indeed, the need for retaining the stack-card metaphor and language elements is due to the company's wish to ease the transition of previous users of *Hypercard* and *Supercard* to allow them to import pre-existing "stacks" made with those products with only minimal scripting changes needed). Additionally, Revolution provides a wide variety of pre-made interface elements, from various button types (including menuing elements)to different text fields to graphics, movie objects (linked to QuickTime movies), image areas (for bitmapped graphics) and, within variants of *nix, vector graphics which the user may add to their project by a drag-and-drop methodology from Revolution's tools palette.

In sharp contrast to web-based documents, it is very easy to bring the application to life. Simple actions can be assigned *directly* to any of those elements with Revolution's language scripting language, thus reinforcing the concept of objects which exhibit event-driven behavior via the pedagogically sound method of linking code and output.

[19] Ramadhan, H. (1992). An intelligent discovery programming system. *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: technological challenges of the 1990's*, 149-159.

[20] Smith King and Barr, 1997

For instance, in a simple navigation system within a stack, the stack developer might wish to provide forward and backwards navigation buttons. Using Revolution, the procedure would be to drag-drop the button type of choice to one's open stack, then double-click the button itself directly to access a script window allowing the developer to assign the following scripted behavior (note: all text following "---" is commentary to the preceding code; the "---" tells the underlying engine to ignore anything following it on any given line):

```
on mouseUp --the event precipitating the action
  go next card -- the behaviour to be performed by clicking
end mouseUp -- the end of this particular action
```

Thus, in the above example, we have visual objects that the learner can manipulate, a direct object-action mapping paradigm linking code, object and outcome, and, additionally, by switching mode within the development environment from development to user/test mode (Revolution does not require a program to be compiled for testing, or, indeed, even end user use purposes), the learner can immediately test his/her efforts and receive valuable feedback which can serve to reinforce learning.

This is in stark contrast to authoring environments which utilize more formal programming languages: Richard Decker (of *Analytical Engine* fame, 1990) has noted that "even students with very good quantitative skills often expend more energy learning where semicolons belong than they do mastering the concepts."[21] Decker's initial efforts involved a "best approaches" look at the single, terminal, university-level course in computer science which targets the non-computer science major. His findings suggest that visually-oriented authoring environments such as *Hypercard* or Revolution with natural-language like scripting languages support learning in that novice/non-programmers "want results, and we feel that at the introductory level this is an entirely appropriate point of view."

Furthermore, by engaging in even such a simple programming activity as that noted above, the direct and immediate feedback can provide a sense of accomplishment[22] that not only is encouraging for the new/non-programmer, but also is an example of "actually completing assignments... that require technology skills" that Efaw (2005)[23] has noted as being one of two critical elements for successful implementation of technology in the classroom.

Meaningful interactivity, however, requires more than simply creating "go next" buttons that mimic using the space bar in a PowerPoint presentation. More complex interactivity can be created in Revolution by using what is called "branching constructs"; these provide different paths of action for the end user/student and can

---

[21] Decker, R. and Hirshfield, S. (1990), A Survey Course in Computer Science using Hypercard. Proceedings of the twenty-first *ACM-SIGCSE technical symposium on computer science education 22(1)*, pp. 229-235.

[22] Decker, ibid.

[23] Efaw, J. (2005) , No Teacher Left Behind: how to teach with technology. *Educause Quarterly 4,* 28-29.

also be achieved  using, again, natural-language constructs:

```
on mouseUp
  answer “What do you want?” with “Coffee”, “tea” and “me”
    if it is “Coffee” then -- user clicked Coffee
      answer “Bad for your health”  -- new dialogue box
    end if
    if it is “tea” then  -- user clicked tea
      answer “High in antioxidants!”
    end if
    if it is “me” then  -- user clicked me
      answer “I'm taken!”
    end if
end mouseUp
```

In the above example, if-then and if-end if structures are used to provide branching for interactivity.  The `answer- with` command produces a dialogue box with the button choices specified in the remainder of that line of code (up to 7 such choices are supported) whereas the `answer` alone command provides a dialogue box with the text specified.

Thus, on the surface, Revolution works much like the easy-to-use interface and application builder that *Hypercard* was. It is very similar. However, Revolution takes the *Hypercard* paradigm several steps forward.  Recall some of the disadvantages of *Hypercard*:  limited graphics support, practically  nonexistant color support and the inability to deploy creations to the Windows operating system. Conversely, Revolution supports the major graphics file formats (TIFF, PNG, JPG, GIF etc.) as well as provides modern color support.  Additionally, Revolution breaks the platform barrier by allowing a stack developer to deploy his or her creations onto the current major operating systems – Mac OS 9, Mac OS X, Windows Vista and XP, and a number of unix variants. Moreover (and very unlike *Hypercard*, which provided user interface elements that in some instances weren't even in compliance with Apple's Human Interface Guidelines -- HIG -- for the time), the Revolution engine creates interface elements that are HIG-compliant for every platform supported *without any work required by the stack developer*.

It sounds  simple, and, for you, the potential developer of educational learning objects, it *is* simple,  but what it means is quite extraordinary. You could even start making shareware applications that not only run in the Mac OS, but also in Windows, IRIX, Solaris, and more. And in case you prefer to impose your own style rather than follow the OS native ones, you are free to create windows of any size and shape or create widget-like applications translucent backgrounds. Moreover, with respect to the issue of distribution and reuse of learning objects, Revolution provides the free and easy ability to upload your stacks to a common and freely accessible server from within the program's IDE (integrated development environment), another successful strategy for assisting educators to infuse technology in the classroom.[24]

Revolution is thus a write-once, run-anywhere format. For the capital outlay of less than US$100, users can distribute their application in a format that requires a player. However this player comes for free and exists for any of the most common operating systems. For a heavier price tag, you can compile your stack and transform it into a native executable application that runs on those same operating systems, thus eliminating the need for a player engine. All you need is a license that allows you to compile for the appropriate OS. In other words, the potential audience for any of your applications can be almost infinitely expanded, literally at the click of a button.

Moreover, and important with respect to the popularity of learning objects and building collaborative repositories for the same, in addition to the ability to upload learning objects to Revolution's publicly-accessible server, uncompiled Revolution stacks allow new users or fellow developers the possibility to modify the existing learning object *or* repurpose useful code modules used within the learning object. This results from the uncompiled nature of the Revolution stack which provides not only full access to underlying code attached to specific objects, but also the ability to simply copy-paste useful objects and code between learning object stacks. Thus Revolution is not only learnable, but its IDE or integrated development environment actually supports as well as encourages learning and the ability for code modification/reuse directly.

**Advanced Features**

Revolution also provides solutions for more complex visual and audio representations of information. It has functions that give you formatted display of HTML or RTF content; spreadsheet/table fields; MIDI music file creation and playing; new sound-recording architecture; support for the parsing and creation of XML documents; Unicode text entry and text manipulation; instant access to web protocols like HTTP or FTP, and TCP sockets; almost instant access to SQL databases; and calls to the system shell. As an example, it has the ability to read a web document using the simple single line of code `get url "google.com"`. Similarly, external web files can be linked to within Revolution using the simple bit of code `go url "http://google.com"`, which launches the end user's default web browser and, if the computer has an active internet connection, directs the web browser to the specified site.

Revolution is also adept at text handling, largely because it does not utilize typed data that most formal programming languages use. For example, in a formal programming language, data must be declared to be boolean, integer, floating and strings. Failure to correctly indicate the data type can lead to the program not working. However, in Revolution, there is no need to declare data types: Revolution simply examines the data in context and chooses the correct form of treatment. Hence, in Revolution, "two" is the same as "2". To put in a scrolling list field the data corresponding to the fourth column of data of a csv file exported from Excel, nothing more is needed than

```
repeat for each line l in file url "my.home.page—my_data.csv"
    put item 4 of line l after field "the data"
end repeat
```

[24] Efaw, op cit., 30.

Revolution can also handle regular expressions. All tag names in a XML document can be found with the instruction

```
get matchtext(the_text, "<([^> ])+", the_match)
```

**Regex and CSV and XML, oh my!**

What's all this incomprehensible stuff about csv formats and regular expressions and text handling? Don't worry – if you don't need it, you don't need to know about it to use Revolution. But it's nice to know that, should you, the now novice/non-programmer, ever decide to spend a half-decade pursuing a degree in computer science, you won't necessarily feel the need to pitch Revolution overboard because it can't do "real" programming. For instance, recall the earlier example touting the simplicity and intuitiveness of Revolution's use of the if-then and if-end if constructs? Here's a secret: "real" programmers largely hate these constructs for being too verbose; they like "case" and "switch" statement constructs. Revolution isn't particular about which you use; use whichever is most within your comfort zone. Just keep in mind that being able to build something functional, polished, and impressive on, say, your Mac and hand it over to your, say, Windows or Linux-using (or, vice-versa) students and colleagues comes at the educationally-attractive price of ~US$50 (for Revolution *Media*, which also comes with pre-built templates, including games).

## Conclusion

When Apple stopped supporting *Hypercard*, Educators were forced to moved on. Some moved to Macromedia *Director* or *Flash*, others to *REALbasic,* and, increasingly, many others to Microsoft's *FrontPage* and *PowerPoint*. But of the latter two, one still requires the mastery of complex language solutions and the second is lacking in interactivity. Somehow, the complexity or limitations of the "solutions" currently in use by and for educators seem to have put an end to educator's efforts and abilities to develop clever applications for use in the classroom. We sincerely hope that Revolution will re-energize them. The Revolution development environment is a breakthrough for anyone who writes and designs computer software. Revolution enables developers to easily and quickly create powerful Internet-enabled applications and solutions which can be delivered on Linux, Mac OS X, classic Mac OS, Windows, and popular UNIX systems. This makes it ideal for the education market.