A NEW APPROACH TO INCREASING CBT DEVELOPER PRODUCTIVITY WITH AN INSTRUCTIONAL DATABASE

Dr. Jesse M. Heines
University of Massachusetts Lowell
Lowell, Massachusetts, U.S.A.

Dr. Robert S. Becker
TW Design, Inc.
Atlanta, Georgia, U.S.A.

Abstract

The development of computer-based training is a highly labor-intensive activity. Many CBT authoring systems have been created to increase CBT developer productivity, but many become unwieldy when applied to large courseware projects. The authors have worked to increase development productivity by providing instructional designers with a way to build a database of lesson content that is presented by a lesson "engine" cognizant of the course's overall lesson logic. The engine takes advantage of repetitive interaction strategies to present students with a highly consistent user interface that is easy to update when course requirements change.

Authoring Systems and Productivity

In 1986 Heines and Israelite wrote, "the major purpose of computer-based training (CBT) authoring systems is to increase CBT developer productivity," and that purpose remains unchanged in 1994. Over 50 major authoring systems have been on the market at one time or another since that writing (*CBT Directions*, 1992), each with its own approach to simplifying the process of converting lesson storyboards into interactive learning programs. As one reviews this myriad of systems, it is easy to paraphrase Will Rogers and say, "I never saw an authoring system that didn't have some unique feature I really liked."

While today's authoring systems certainly simplify the job of producing individual courseware components (especially those involving multimedia), many of them become unwieldy when used to produce courses requiring thousands of such components. Perhaps the clearest examples of this may be seen with visual programming systems like Authorware and IconAuthor. These systems are truly excellent tools for producing a wide range of instructional programs, but

---

This paper was presented at the Asian-Pacific Information Technology In Training and Education (APITITE) Conference in Brisbane, Australia, on June 29, 1994.

anyone who has used them to produce courses with thousands of icons will tell you that the paradigm that produces quick, snazzy demos on exhibit floors can be difficult to manage when applied to courses of any realistic size.

One major problem with such authoring systems is that they lack true subroutining capabilities and the ability to access sophisticated databases. There is much repetition in CBT courseware from lesson to lesson, and when the structure needed to implement typical sequences must be repeated over and over rather than passing parameters to a single subroutine, the size of a course can grow out of control. Such multiple instances of logical sequences can be a nightmare when the course must be updated. In this situation, one seldom finds *all* occurrences of a specific instance, leaving dangerous inconsistencies in the course's user interface. When RFPs for major courseware development efforts call present parameters like 16,500 screens, 800 graphics, 4000 pages of text, etc., and estimates of hundreds of man-hours to produce a single hour of "moderately sophisticated instructional or training interaction," (Fairweather and O'Neal, 1984) the ability to reuse lesson logic and access courseware components from databases becomes critical.

The more pressing problem, however, is that these authoring systems address the *implementation* of CBT functionality, but do little to address the even more labor-intensive CBT *design* process. In *Making CBT Happen*, Gloria Gery (1987) outlines the following five-step "software development process" for CBT, with the indicated deliverable(s) for each step.

- Project Definition, which yields documentation specifying the audience(s), learning objectives, course topics, interactivity levels, course standards, and design schedule
- Design, which yields a course design document
- Development/Scripting, which yields storyboards and/or scripts
- Programming/Entry Into Authoring System, which yields runnable courseware
- Evaluation, which yields revised courseware

In this scenario, not only does the authoring system make no contribution to the first three steps, but the deliverables of these steps – documentation and paper mock-ups – cannot in any way be used to automate the programming or data entry task in the fourth step. Even with on-line storyboarding tools, Alessi and Trollip (1991) recommend doing storyboarding on paper first. It is important to note that adding more people to the definition, design, and storyboarding steps does not necessarily increase productivity. As Frederick Brooks (1975) taught us in *The Mythical Man-Month*, as one puts more human resources on these types of tasks, the time needed to resolve the inevitable communication bottlenecks between workers quickly offsets any increases in the actual number of man-hours worked.

The Instructional Database Approach

We are approaching these problems by writing courseware presentation engines that read course content from databases and present it with a consistent user interface. This approach addresses the implementation issue for large courseware projects by providing a near total separation of lesson content and lesson logic, thereby allowing instructional sequences (logic) to be reused with different data (content) with no increase in programming overhead. It addresses the productivity issue for these projects by integrating the storyboard process with the building of the instructional database, therefore realizing significant productivity gains by obviating the need for reentering content data specified in the design and storyboarding steps. A similar approach has been discussed by Sampath and Quaine (1990).

Early Presentation Engine

In 1986 the first author and his colleague Larry Israelite demonstrated a very small lesson presentation engine (less than 40,000 bytes in compiled form) that read course menus and lessons from an instructional database and displayed them to students with a consistent user interface (Heines and Israelite, 1986). Lessons consisted of both text and graphics, and specification of this content was coded manually using a standard text editor and a tiny language (approximately 10 commands in all). A simple text and graphics screen from that early work is shown in Figure 1, while the same screen with an overlaid help window showing user interface options is shown in Figure 2 (Scientific Systems, Inc., 1986).

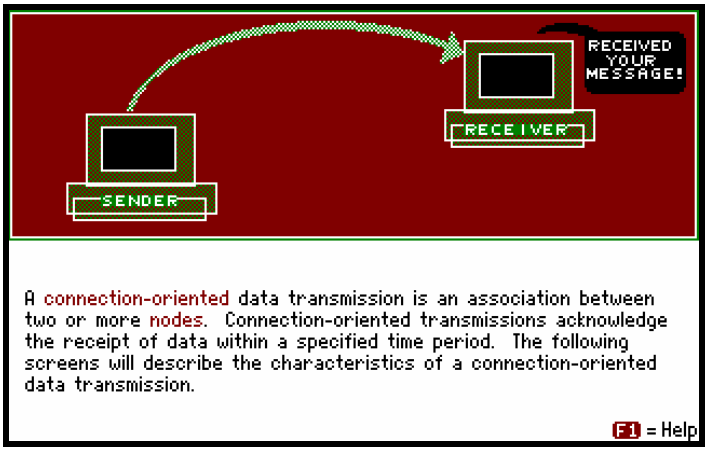

Figure 1. Lesson display from an early presentation engine.

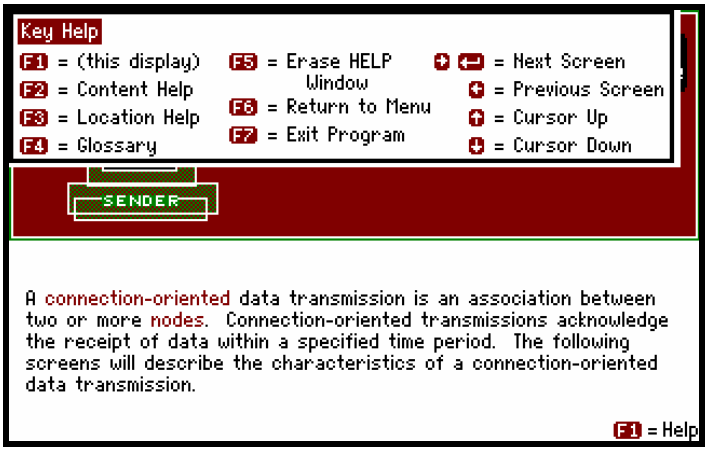

Figure 2. Lesson display with overlaid help window and consistent user interface.

The 1986 lesson presentation engine had two drawbacks. First, the displays and interactions we were able to present with this engine were relatively simple by today's standards. Second, the lessons were coded using a text editor and thus authors did not have a WYSIWIG "feel" as they developed their lessons, although the interpreter automatically produced storyboards from the author's input. The approach's main advantages were that it produced a course with a highly consistent user interface in spite of the fact that about a dozen instructional designers were involved in creating materials, and that it allowed faster lesson production than could be achieved by the same team working in the underlying authoring system (TenCORE®).

Current Presentation Engine

We have made considerable effort to build on that early work, and we are in the process of building a much more sophisticated lesson presentation engine that reads menus, lessons, glossaries, help texts, and navigational control from an instructional database and presents them using a consistent graphical user interface. In trying to maximize productivity, our main goals in this effort were:

- to provide instructional designers with a WYSIWYG, on-line storyboarding tool
- to increase the speed with which lessons can be developed
- to automate the process of creating finished lessons from the on-line storyboards
- to create a system that allows instructional designers to update their lessons easily

The first goal is being achieved using TenCORE Producer®, a product of Computer Teaching Corporation in Champaign, Illinois. Readers experienced with this tool may immediately ask why, if we use TenCORE Producer, don't we just run our lessons using the Producer execution engine? The answer is that we need far more control over screen elements and interactions than is possible using the Producer executor. Like many such point and click authoring systems, Producer simplifies course development by making many assumptions about student/computer interactions that we simply were not able to accept for this project.

The second goal is being achieved by developing display and interaction templates that users copy within Producer and modify to create the sequences presented by the lesson engine. Figure 3 shows a simple display template as it appears to instructional designers within Producer, and Figure 4 shows how that display appears to students in the CBT course.

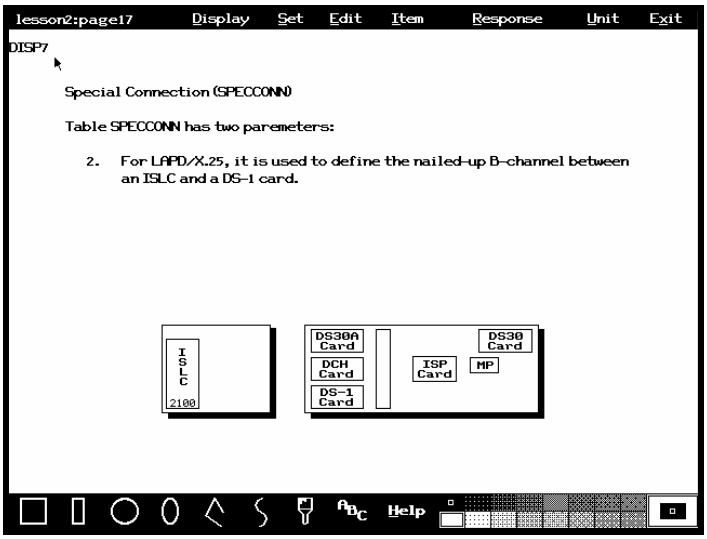

Figure 3. Lesson display as viewed by an instructional designer in the TenCORE Producer storyboard tool.

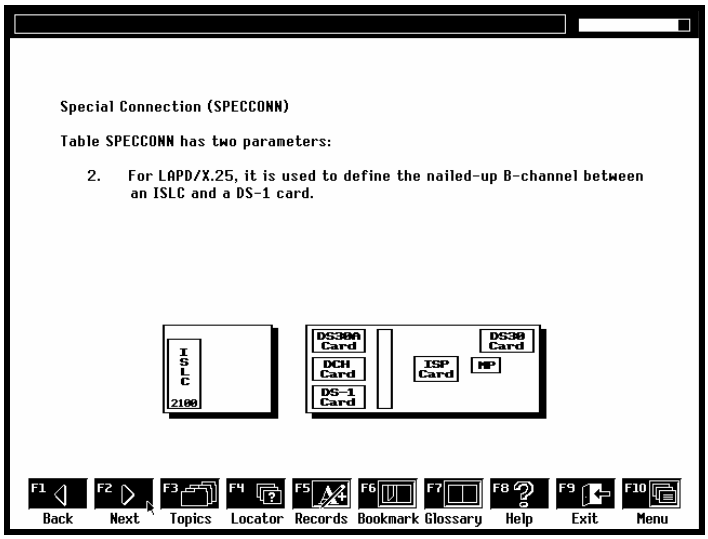

Figure 4. Lesson display as viewed by a student running the presentation engine (Northern Telecom, 1994).

A more sophisticated example is shown in Figures 5 through 7. Here two template screens are combined (Figures 5 and 6) to create a single display screen (Figure 7) with a twist. Based on the template identifiers that appear at the upper left of the screens during the storyboarding process (see the arrow in Figure 6), the lesson engine logic "knows" that the text in the second screen is to be overlaid onto that in the first inside a user-movable text box. Such power is made possible by separating lesson logic from content. Note also the large number of student options at the bottom of the screen in Figures 4 and 7. Such options would have been very difficult to implement using the Producer execution engine, but are a reasonable task for the TenCORE Language Authoring System in which our lesson engine is written.

The third goal is being achieved by using the raw output of the TenCORE Producer storyboard tool as our instructional database. Again, to readers experienced with this tool, this means that we read the .TPR file directly; we do not convert it to a .BIN file. The .TPR file is a TenCORE "nameset," a file format that is fully documented in the TenCORE Language Authoring System manual for version 4.2. It is interesting to note that we looked into using other database generation tools for this goal, most notably FileMaker Pro for Windows, but Claris, the software manufacturer, refused to give us documentation on the file format written by this excellent database tool, so we were unable to use it. (The exported ASCII file written by FileMaker does not contain graphics or text formatting information, so it is useless for our application.) The format of the "nameset" written by TenCORE Producer is documented in the TenCORE Language Authoring System manual, so we have been able to process it easily.

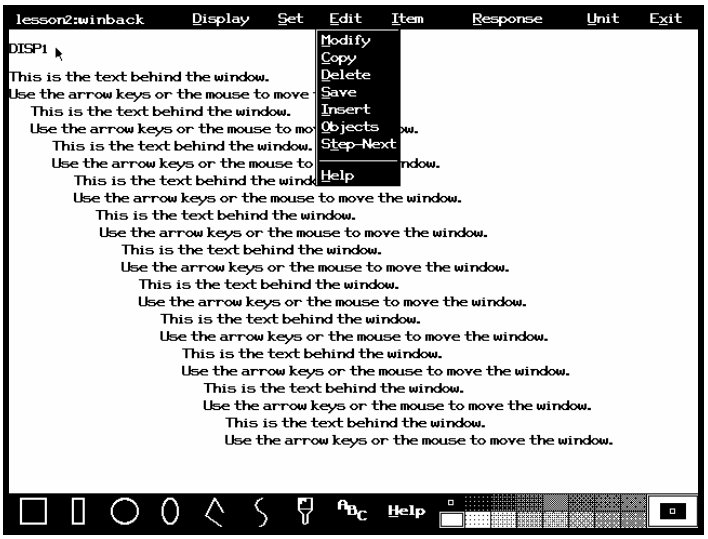

Figure 5. First part of a composite screen. Note that the template type specified in the upper left-hand corner of the screen is DISP1.

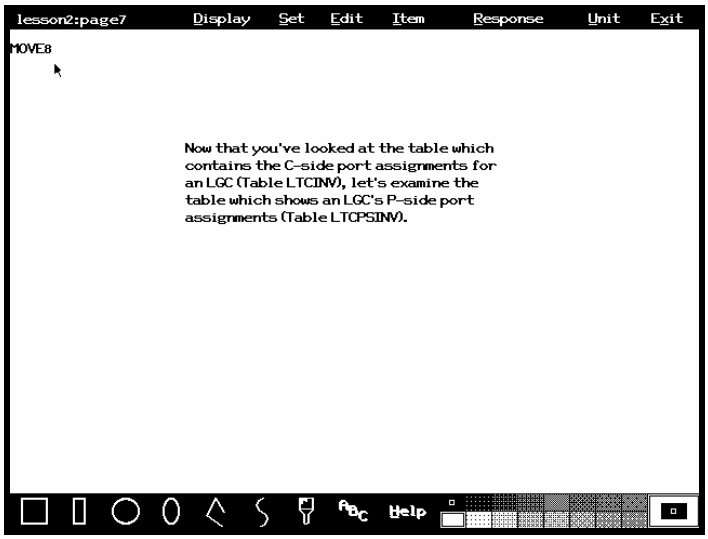

Figure 6. Second part of a composite screen. Note that the template type is MOVE8. (Northern Telecom, 1994).

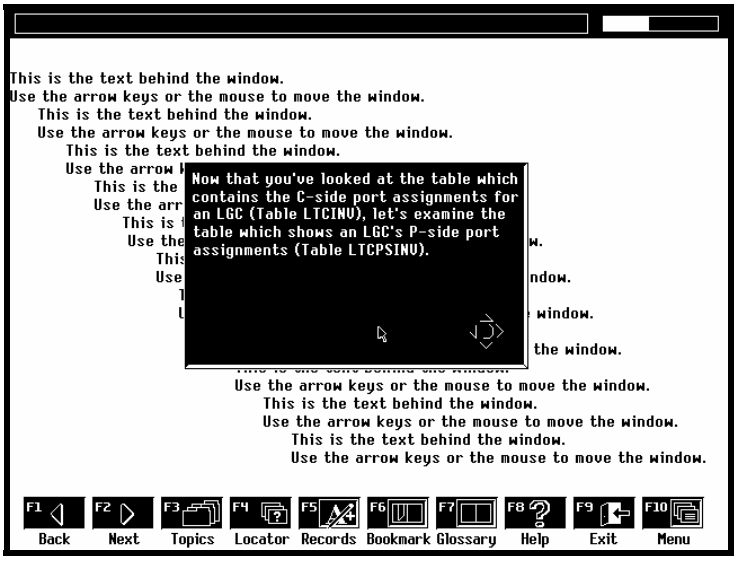

Figure 7. Composite screen with a movable text window as seen by students, built from the templates in Figures 5 and 6.

The combination of the previous three elements goals has helped us achieve our fourth goal: to create a system that allows instructional designers to update their lessons easily. While we do indeed have a team of programmers building the content-independent lesson engine,

instructional designers are able to update their own lessons without needing to convey the changes to a programmer. What's more, the separation of logic and content has made the entire course much easier to update than our client's previous courseware. When the client changed management and lead designers, the programming team was able to accommodate design requests by changing the lesson engine and have those changes reflected throughout the entire course. Such flexibility would have been impossible had we not taken the instructional database approach.

Application

The approach described here is being used to produce a course with over ten thousand screens, thousands of interactions, and hundreds of graphics. When the course is analyzed, however, there are less than 200 varieties of screen layouts and 20 types of interactions, each of which can be specified in a template that developers can copy into a lesson sequence and then edit to add course content. The presentation engine takes advantage of the similarities between screen layouts in much the same way that word processing style sheets obviate the need to store complete formatting information with each instance of a paragraph of a given type. That is, the engine knows how to present and have the user interact with each template type, and thus requires that the database to contain only the information unique to each instance of template.

While developers certainly need to be trained in the use of a tool such as that we are developing, that training is minimal (on the order of a few days), and the resultant course is guaranteed to have a consistent user interface throughout its entire domain. We expect our approach to reduce programming time and code size, thus reducing the inevitable bugs. It should also allow developers to be more attentive to function and content by automating routine tasks and choices having to do with form, i.e., screen layouts and interactions. Our experience is that freedom from such details allows developers the flexibility to create more meaningful interaction and instructional strategies.

These characteristics should have a direct effect on productivity, allowing the large course to be developed with minimum cost and maximum efficiency. Empirical data on such gains was not available at the time of this paper's writing (February 1994), but should be available by the time the paper is presented (June 1994). Readers are invited to contact the major author (heines@cs.uml.edu) to request an addendum on such data when it is available.

Acknowledgments

This work is funded in part by Northern Telecom of Raleigh, North Carolina, and TW Design of Atlanta, Georgia. The current presentation engine is being coded by Heather Ehrlich, Ngo Phan Greg Flynn, Graham Gerade, and Mike Chartier working with Dr. Heines at UMass Lowell.

References

Alessi, Stephen M., and Stanley R. Trollip (1991). Computer-Based Instruction: Methods and Development. Prentice-Hall, Englewood Cliffs, New Jersey, pp. 332-334.

Brooks, Frederick P., Jr. (1975). The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Publishing Company, Reading, Massachusetts.

CBT Directions (1992). *Comparing Authoring Systems: Where Do You Start?* CBT Directions, 5(3):15-29, May/June 1992.

Fairweather, P.G., and A.F. O'Neal (1984). *The Impact of Advanced Authoring Systems on CAI Productivity*. Journal of Computer-Based Instruction 11(3):90-94, Summer 1984.

Gery, Gloria (1987). Making CBT Happen. Weingarten Publications, Boston, Massachusetts, pp. 95-96.

Heines, Jesse M., and Larry Israelite (1986). *Increasing CBT Increasing CBT Developer Productivity with an Instructional Database*. 28th International Conference of the Association for the Development of Computer-based Instructional Systems, Washington, D.C.

Northern Telecom, Inc. (1994). *DMS-100 Family/DMS SuperNode System Maintenance Curriculum Course Management Program* (CBT Course). Northern Telecom Technical Education Center, Raleigh, NC.

Sampath, Santha, and Andy Quaine (1990). *Effective Interface Tools for CAI Authors.* Journal of Computer-Based Instruction 17(1):31-34, Winter 1990.

Scientific Systems, Inc. (1986). *MAP/TOP Advanced Concepts* (CBT Course). Scientific Systems, Inc., Cambridge, Massachusetts.