Implementing a Problem Based Learning Environment in Flash

Gordon Graber
School of Information Science and Learning Technology
University of Missouri, USA
glgqk3@mizzou.edu

Abstract: Problem Based Learning Environments (PBLEs) can provide an integrated series of authentic learning activities where students employ multiple cognitive strategies to master complex and ill-structured problems. Technology enabled PBLEs are potentially effective delivery mechanisms for these unique learning experiences on a large scale. This paper examines the design and implementation issues involved in creating a PBLE for Mathematics Education Department using Adobe Flash. Best practices concerning Flash authoring, scripting and design issues, tracking and saving student progress, sequencing and linking activities, and saving final results for assessment using external tools are presented. Underlying data structures supporting the PBLE activity, and local file and server storage access issues are explored. Recommendations are forwarded for the design and implementation of dynamically generated PBLE content using Flash.

This paper presents conceptual and technical issues encountered in the design and implementation of a Problem Base Learning Environment (PBLE) using Adobe Flash CS4 and its companion ActionScript 3.0 programming language for the Mathematics Education department at the University of Missouri, Columbia campus. The author worked as the Flash designer and ActionScript programmer in a team effort with two content developers and a university mathematics education instructor client. Together we created a prototype PBLE designed to guide pre-service mathematics teachers through issues relating to maintenance of cognitive demand in 9th – 12th grade mathematics students. At the time of this writing, the resulting PBLE (http://web.missouri.edu/~glgqk3/MathPBLE) is under evaluation.

Problem Based Learning

Problem Based Learning Environments are designed to provide learning experiences of complex problem solving activities in authentic contexts. In traditional instructional design, problem-solving activities are often presented narrowly, decontextualized from the situations where they are encountered in the real world. Such problem solving activities are designed to help students dispatch well-structured problems, where the problem type is reduced to its essentials, the problem domain has been segmented into the minimum taxonomies necessary, and the variables and constraints, methods and outcomes are defined for the learner before hand. In this type of instructional design, the student engages in analogical reasoning that is relatively superficial and limited, and may not transfer well to other contexts (Jonassen, 2006). The worked examples and ensuing end of chapter practice exercises common in course textbooks represent this type of learning activity. Students are walked through specific problem types and their respective solutions, and are then expected to identify and resolve similar problems on their own, after enough practice. The instructional strategy is that learners will gain problem solving experience in phases, starting with specific activities in constricted contexts and building up successive skills, to be able to tack more generalized, complex problems in the future (Sweller & Cooper, 1985). However, we rarely encounter problems packaged in this manner in our work and private lives, and instructional design that focuses on presenting well-structured problem solving activities so as to ease cognitive load may limit student learning (Jonassen, 2006).

Proponents of Problem Based Learning suggest that the problems we meet in our goings on outside of traditional academic pursuits, and how we learn to solve them, are often not so simple. Problems are sometimes nebulous, with dependant parameters and limits that are mediated over time under social and environmental pressures. Real world problems may have outcomes that have multiple positive and negative impacts to the individuals, organisms, organizations and environment from which the problems spring (Jonassen, 2000). The development of the PBLE prototype used as the basis of this study is an example of this type of complex and ill-structured problem. As the software designer and developer of the PBLE development team, the author had to

negotiate with a client and two content developers to create a usable and useful instructional design that would help pre-service teachers of mathematics build the skills necessary to maintain their prospective students levels of cognitive demand. Among multiple implementation technologies, difficult choices of combinations of programming environments, scripting and server based systems had to be made, none of which had clear advantages over the other. At the same time, the project requirements evolved and where shaped by the influences of the people involved in the process. The form of final outcome - a model of a PBLE for pre-service mathematics teachers - was itself an unknown given that the requirements that it be usable and useful. Neither concept had an enumerable set of properties that could be known beforehand. Rather than learning how to deal with each of the components involved in the PBLE design processes in piecemeal fashion, as a student in traditional instruction, the author had to cope with the complexities of the design process all at once. Despite the difficulties, and perhaps because of them, in the experience gained in the process will likely develop into meaningful and memorable skills in computer programming, systems integration, working with people, and the related concepts necessary to design and produce the PBLE prototype.

PBLEs attempt to address these ideas by presenting problem solving activities in instructional designs that mimic the complex difficulties we find in our everyday experiences, where initial problem parameters may be unknown; there may be no well-defined solution paths; and where multiple, equally valid outcomes are applicable and assessing the utility of one over another may be challenging (Jonassen, 2000). In designing the PBLE for this project, we attempted to represent these facets of complex problems through intertwined learning strategies, delineated by Wilson and Cole (1996): students develop an understanding of the problem space in activities that have them classify the problem; students draw inferences about the problem, methods and possible solutions from case studies which communicate disparate aspects of the problem incompletely and inconsistently; problem information is presented in multiple modalities – text, images, audio, and video; students engage in causal thinking, making connections between problem variables and outcomes in activities where they create causal maps; students engage in argumentation, both in analysis of case studies and in weighing the usefulness of outcomes they have constructed. One key learning strategy we sought to engender in the PBLE is that of Cognitive Flexibility (Spiro & Jehng, 1990), where students would need to access the teacher cases, depictions of classroom activity, and real teacher's audio stories in identifying themes that relate to the factors affecting the maintenance of cognitive demand, and produce arguments for where and how those themes may or may not be present in the different cases.

Choosing A Development Environment

One of the requirements for the PBLE was that it needed to be technology supported, and accessible to undergraduate students through the Web. Adobe Flash was selected as the development environment for the prototype PBLE for a number of reasons. Flash affords a level of interactivity that Web pages lack without resorting to other Web browser and server based scripting technologies. Interactivity in Flash can be created entirely independently, without the need to communicate with outside resources. Additionally, Flash can output stand-alone executable files that may be distributed on CD or over a network and run without a network and Flash Web browser plug-in. The latest version of the Flash programming language – ActionScript 3 - is a flexible, full-featured object oriented programming language that is well supported by Adobe and a host of enthusiasts across the World Wide Web. One concern our client expressed was that students would likely not be able to complete it in one sitting, and we would need some way of saving their progress. ActionScript provides API resources to save data locally and is trivial to implement. Another consideration was that the Flash development environment is better integrated than some other combinations of browser and server side technologies, allowing the designer to move between textual, graphic, user interface elements, and the ActionScript code needed to create the interactivity that ties the elements together with relative ease. Third party developers offer open source additions that extend the APIs, for example, AlivePDF (http://alivepdf.bytearray.org) that our PBLE uses to allow students to save their finished work as PDF files on their computer. Lastly, the author was familiar with Flash, having worked with it since 2001, and using it as a rapid prototyping system for developing a PBLE appeared to be feasible.

Flash Authoring and the PBLE Application Design

Working within the Flash development environment, the designer needs to determine the most efficient method of authoring content such that it can be interactively stitched together with ActionScript. Flash is foremost

an animation system with a timeline segmented into frames. On top of this fundamental construct, Flash provides the elements that make up a Flash product: ActionScript holders, text fields, graphics, interactive user interface (UI) elements, and Symbols. Symbols are ActionScript addressable groupings of script holders, text fields, graphics, interactive elements, and other symbols. Each symbol contains its own timeline is essentially a microcosm of the Flash itself. Symbols in Flash are a powerful method of encapsulating content and its scripted interactivity into autonomous objects that can be nested inside each other to create interactively complex, reusable, pluggable units of content. Content can be created in the Flash file graphically, by importing images and text from other systems or created directly on the Stage (the Flash metaphor for the working canvas area) at a particular frame in time, or programmatically using ActionScript. Content elements may be placed on a single frame or spread across multiple frames (Fig. 1 A). Flash allows the designer to mix any combination of these methods to construct the finished product. Additionally, layers are provided in the Flash editor as an authoring convenience, allowing designers to visually separate content to ease its manipulation.

Such flexibility in the design process presents content integration and management difficulties. In our PBLE, each activity we want the student to engage was envisioned as a Page of content that might include text, images, multi-media elements, and interactive buttons and fields. Conceptually, Pages are grouped into the Cases that our PBLE employs. Each of the pages could have been be constructed either on separate frames (Fig. 1 A), or encapsulated within symbols, all placed on the same frame, or a combination thereof. In our PBLE, for example, moving the student from one page to the next could be accomplished either by advancing the Flash timeline to the appropriate frame containing a specific page, or by making the grouped symbol that represents the page visible.

The mix of frames and symbols the designer chooses is constrained by the routes ActionScript provides to access the content elements. For ActionScript to be able to address Flash content elements, the elements must exist at the same point in time as the ActionScript holder that contains the script commands, which call them. Splitting elements across frames would provide convenient authoring and viewing of the pages of our PBLE (Fig. 1, A), but would make coding interactivity between pages more complex, because the ActionScript commands would need to be divided up and placed in the script holders on the frames containing the content to be referenced. We chose to create the cases as symbols and to encapsulate their respective pages as nested symbols, all placed on a single frame (Fig. 1, B & C). Doing so simplifies our use of ActionScript because we could create and edit all the functions and commands needed to manipulate our PBLE's content in one script editor window.

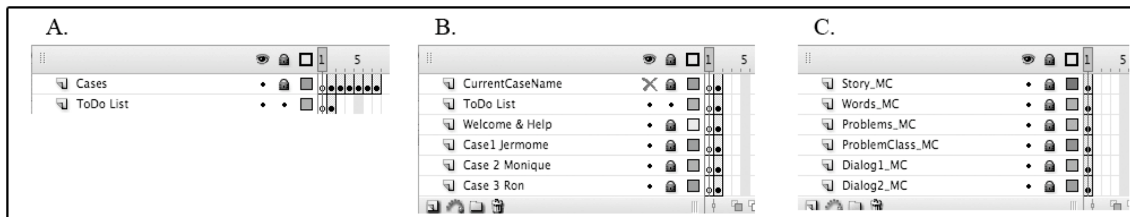

Figure 1: Three sample Flash timelines: A method we chose not to use (A.) - case content placed on separate frames. All PBLE case content grouped into symbols on frame 2 (B.). Note that each symbol has been placed in its own layer, visible as the labeled rows in the graphic. A sample symbol timeline (C.): the "Case 1 Jerome" symbol has been opened in the Flash editor, showing its content pages grouped into page symbols on frame 1. Note that each symbol has been placed on its own layer. Each page symbol contains all of the elements that make up one PBLE activity.

Generalizing Interactivity

ActionScript provides APIs to control nearly every Flash element and symbol, but to do so the elements must be referenced through either pointers or unique Instance Names. User interface access to our PBLE's cases and pages of activity needed to be provided somehow, as well as means for capturing the states of interactive elements, and saving and reporting those states that represent student input and progress. As the number of content elements in prototype PBLE grew, it became obvious that generalized data structures and algorithmic to access the elements were necessary to alleviate the burden of addressing every element uniquely in ActionScript code. To provide access to the cases and pages, we created a hierarchical menu mechanism we termed the To Do List. Without employing

some generalized programmatic method, the designer would have to hard-code links to each new page into the To Do List menu.

A.
```
public class aCase{ // represents an activity in the
    public var caseItems = new Array(); // the list o
    public var currentItemIndex = 0; // which page wi
    public var isOpen = true; // the case list item i
    public var caseMC = null; // the movie clip conta
    public var name = "no name"; // the name displaye
    public var firstView = false; // this case will b
    public var required = false; // must we complete
    public var isComplete = 0; // is this case comple
    public var prerequisite = null; // do we have pre
    public var imgBtnRef = null; //
    public var lineNum = 0; // store our position in

    public function aCase(){ // our constuctor
    }
}
```

B.
```
var nCase = new aCase();
    nCase.caseMC = Case1_MC; // reference to the actual symbol
    nCase.name = "Case 1: Jerome"; // name displayed by the me
casesArray.push( nCase ); // hold references to all case symbo
```

C.
```
public class CaseItem{ // represents an activity in the c
    public var listItem = ""; // the title of the case li
    public var tabMC = null; // this is the movieClip ins
    public var isUnvisited = true; // hasn't been visitec
    public var isComplete = false; // is complete - the u
    public var isRequired = false;
    public var giveFeedback = false; // does this case it
    public var prerequisite = null;
    public var caseIndex = 0; // a link back to the caseA
    public var interactiveFields = new Array();
    public var interactiveButtons = new Array();
    public var feedbackItems = new Array();
    public var feedbackFields = new Array();
    public var lineNum = 0; // store our position in the

    public function CaseItem(){ // our constuctor
    }
}
```

D.
```
// case 1 - Jerome
var mCaseItem = new CaseItem(); // item index 0
mCaseItem.listItem = "Case Background";
mCaseItem.tabMC = Case1_MC.Story_MC;
mCaseItem.caseIndex = 1;
casesArray[1].caseItems.push( mCaseItem );

var mCaseItem = new CaseItem(); // item index 1
mCaseItem.listItem =  "Teacher's View";
mCaseItem.tabMC = Case1_MC.Words_MC;
mCaseItem.caseIndex = 1;
casesArray[1].caseItems.push( mCaseItem );

var mCaseItem = new CaseItem(); // item index 2
mCaseItem.listItem = "Solve the problems";
mCaseItem.tabMC = Case1_MC.Problems_MC;
mCaseItem.isRequired = true;
mCaseItem.caseIndex = 1;
mCaseItem.interactiveFields[0] = Case1_MC.Problems_MC.prob1Explanation;
mCaseItem.interactiveFields[1] = Case1_MC.Problems_MC.prob2Explanation;
mCaseItem.interactiveFields[2] = Case1_MC.Problems_MC.prob3Explanation;
casesArray[1].caseItems.push( mCaseItem );
```

Figure 2: Sample ActionScript snippets delineating our PBLE case class definition, we named aCase (A.) and the instancing of an aCase object (B.) and Pushing a reference to the newly created instance into the array variable: casesArray. Similarly, (C.) denotes the class definition of a page of a case's content, we named CaseItem, and (D.) the instancing of several CaseItem objects. Note that the references to each instance of a CaseItem are "pushed" into the array, caseItems - a property of the aCase class. We only need to explicitly specify the casesArray index of the case we were linking the page content to (D.) for each page. Note in the 3rd page instanced in the sample (D.) that we store references to UI elements, contained within the page symbol, in an array.

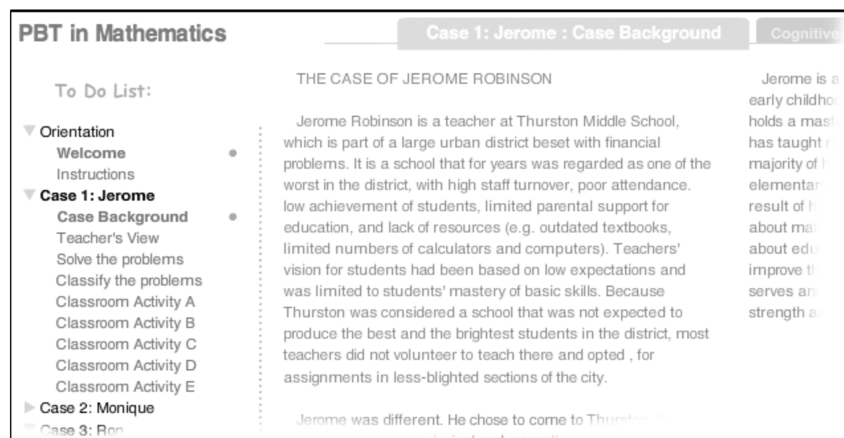

Figure 3: A partial view of the PBLE. The To Do List is a collapsible, hierarchical list construct programmatically by ActionScript commands. It's content and linkages are created by iterating through the casesArray and caseItems arrays visible in the sample code in Fig. 2. The tab heading text, in gray at the top of the graphic, is also constructed from the data stored in the arrays.

ActionScript supplies constructs for organizing access to large numbers of data elements. To avoid having to code individual references to the content items, two of these constructs provided pivotal in the design of our PBLE: Classes and Arrays. Classes are programmer defined objects that group properties - variables that reference other data, elements and objects - into complex data structures which may then be Instanced and populated with unique data and references. Classes can be filled with default data in their definition, reducing the programming effort necessary when instancing them (Fig. 2, A & C). Only the distinctive attribute values need to be assigned to the specific instance's properties after it has been instanced (Fig. 2, B & D).

In our prototype design, arrays afford storage of references to instances of classes and UI elements through array indices, rather than assigning each a unique identifier. ActionScript arrays can be dimensioned dynamically – for example, ActionScript provides arrays with a Push method for growing arrays without needing to calculate the number of cases, pages, and elements we wanted to create before hand (Fig. 2, B & D). By storing all the reference to the cases in our PBLE in one array, and the pages of each case in arrays within the case class object, we can iterate through the content of our PBLE using generic loops. Adding new cases and their constituent pages and interactivity could be accomplished with little effort, by copying existing code and altering the pasted copy with the unique references and array indices pointing to newly authored content.

The To Do List menu (Fig. 3), which provides students with the UI point of access to the PBLE cases, pages and activities, could then be created completely programmatically, without hard-coding the links between the menu items and the pages they referenced. The menu was implemented using a TextField object populated with HTML link tags, containing array indices to each case and page as attributes, by iterating through the casesArray, and the nested pages pointed to by each caseItems index. Adopting this approach meant we could author new content onto the Flash stage, code the corresponding class instances referencing the content, and the content's name would be written into the To Do List, embedded in a clickable link. When the student clicks on a To Do List item link, an ActionScript event is generated and a listening function triggered. The listening function was designed to dissect the event object passed it by ActionScript's event feature, and retrieve selected case and page indices. The function then obtains a reference to the corresponding symbol using the casesArray and caseItems array described previously. The symbol containing the selected case page content was then made it visible after hiding the previous page.

Tracking and Saving Student Progress

Because we implemented a generalized set of data structures and reference retrieval mechanisms for all of the cases, pages, and interactive UI elements in the PBLE, recording and saving student progress is a simple matter of iterating through the sets of arrays to access the stored references to the UI elements, and using those to access the data the student had input. The collected data was passed to ActionScript's SharedObject.getLocal method, which creates a file on the student's hard drive. A login screen is provided to open previously stored data, keying on the input user name. When the student logs in, the PBLE is configured with all of the interactive elements in the same condition the student left them in.

Storing data locally allowed us to short-circuit the process of developing server side scripts and databases necessary to save student progress remotely, speeding up the prototype implementation. In the current prototype we do not track student interface performance data, such as student clicks or page views, though the stored references to the UI elements would easily allow us to do so. For future research purposes accessing this type of data may be desirable.

When the student determines they have completed the activities in the PBLE, they save a PDF file that they may email to the instructor for evaluation. A 3rd party, open source ActionScript class, AlivePDF, effects this procedure. AlivePDF offers a capable API that allows the programmer to copy text and graphics from the Flash application into the saved PDF. Providing this capability is possible on the server side, but nearly as convenient as implementing it in Flash's integrated development environment.

Conclusion: Lessons Learned

Using Flash as a rapid development environment allowed us to create a usable prototype PBLE that has a high degree of interactivity relatively quickly. ActionScript is complete programming language that contains APIs for all of the features we needed to implement, and we were able to write the code that drives the PBLE in less than

60 hours over the course of a semester. Because the PBLE design and implementation is a complex, and ill-structured design problem, much of the code evolved as the PBLE was created. There are features that may have been constructed differently, had we time to do proper analysis and top down program design,

One of the important difficulties creating applications in Flash is the timeline / frames / layers construct. While suited to animation, creating interactive applications is limited by how and where content can be placed, and referenced by ActionScript commands. Authoring content across multiple fames makes viewing the content straightforward to visualize, but complicates the programming structures that need to access it. Authoring on one frame makes the ActionScript code less convoluted, but makes viewing each unit of activity more difficult.

A solution to this conundrum might be to author the content externally, and dynamically add it to the PBLE at runtime by fetching it over the Web. The Flash API provides a URL Loader class for this purpose. On the surface it would seem the designer could author each page into a separate flash file, making viewing the page content less cumbersome, but doing so leaves the developer with the same problem, as spreading content across multiple frames: and programming commands that need to reference elements in the external file need to exist at the same time the elements do. Again, the developer would have to split code into the external flash files, making code integration problematic.

One possible, related, solution would be to devise a mark up language, possibly based on HTML, and code all elements as external Web pages in a Web page editor. The PBLE could then load and parse the external page markup, and dynamically generate the UI elements that need to be referenced. ActionScript references necessary to access, track, display, show and hide the respective pages, activities, and UI elements would be stored at the time the page is reconstituted. This approach would require designing a generalized a markup language extension to HTML, that would encapsulate all of the possible interaction types a PBLE would need to employ. Doing so would allow PBLE designers to create content using a Web page editor, without Flash. Flash would then become the vehicle for displaying and sequencing PBLE, and recording student activity to the content in a completely generalized fashion. Such a system could serve as a PBLE generator. This idea represents a possible future project.

References

Jonassen, D. (2006). Facilitating Case Reuse During Problem Solving. *Technology Instruction Cognition And Learning, 3*, 51-62.

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development, 48*, 63-85.

Spiro, R. J., & Jehng, J-C. (1990). Cognitive Flexibility and Hypertext: Theory and Technology for the Nonlinear and Multidimensional Traversal of Complex subject matter. In D. Nix & R. J. Spiro (Eds.), *Cognition, education, and multimedia: Exploring ideas in high technology,*163-205.

Sweller, J., & Cooper, G. A. (1985). The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction, 2*, 59 - 89.

Wilson, B., & Cole, P. (1991). A review of cognitive teaching models. *Educational Technology Research and Development*, *39*, 47-64

Acknowledgements

The author wishes to thank the PBLE content developers, Weichao Chen, Yayun Yang, in the School of Information Science and Learning Technology at the University of Missouri, who helped create and edit much of the textual content used in the PBLE, as well as guiding and negotiating the conceptual design with the client, Dr. Oscar Chavez in the Mathematics Education department at the University of Missouri, the client for whom the PBLE was created. We would also thank Dr. David Jonassen at the University of Missouri, who's contributions on PBLE design concepts were instrumental.