A System for Developing Tablet PC Applications for Education

Sam Kamin Michael Hines Chad Peiper Boris Capitanu
Computer Science Dept.
University of Illinois
Urbana, IL
{kamin,mhines3,peiper}@uiuc.edu, borice@gmail.com

ABSTRACT

We describe a new system for building Tablet PC-based classroom software. The system, called SLICE, is built for extensibility, using a unique "explicit state" model. Applications developed thus far include presentation, classroom interaction, shared code review, and exam grading. The paper presents an overview of the system and describes these four applications. It then explains the extensibility model, showing how users can add new features.

Categories and Subject Descriptors

K.3.1 [**Computer Uses in Education**]: Collaborative learning

General Terms

Human factors

Keywords

Tablet PC,educational technology,end-user programming

1. INTRODUCTION

Tablet PCs have great potential for classroom instruction. Their uses have been explored in numerous publications [1,2,8,11]. They have been used in elementary and secondary education and for teaching numerous academic subjects, but development of new systems has, not surprisingly, been done mainly in Computer Science departments.

This paper describes a system, called SLICE, which shares the goals of these other systems, as did our previous system, eFuzion [6,7,10]. The name stands for "Students Learn In Collaborative Environments," which succinctly states the philosophy of our group. SLICE, however, has a further goal, which is to ease the development of new educational applications. Because the needs of classrooms are so varied — based on subject, class organization, educational level, and teacher preference — a monolithic approach to system development would lead to an impossible profusion of systems. Furthermore, it is our belief that *topic-specifc* features will be increasingly important, militating for a highly agile and decentralized approach to feature development. The history of extensible systems such as emacs [9] and TeX [3] shows that motivated users, given the tools, can bring a wealth of creativity to the task of creating custom functionality.

To achieve this second goal, we have developed an extensible system. SLICE applications are written using Python scripts running on top of a simple core of functionality. We have thus far developed four applications: single-user presentation, classroom interaction, exam-grading, and code review. (The first two of these share much of their code, as their user interfaces are similar.) These four applications are described in section 2.

The core of the system consists of 10,000 lines of C# code. To give one example, the single-user presentation system comprises an additional 800 lines of Python. Note that the core system does not even understand the concept of a "page;" even that primitive notion is programmed using scripts. As we will see in section 3.1, the page feature is not particularly difficult to implement; the additional generality obtained by not building it into the system allows for the programming of non-page-based applications, such as the code review application.

This paper gives a description of the system and the four applications mentioned above. It provides only minimal descriptions of classroom experiences, and no formal evaluation of educational benefit, as we have not yet done the requisite studies. We are planning such studies in the current (Fall, 2007) semester.

The structure of the paper is as follows: In the next section, we discuss the four applications mentioned above. In section 3, we explain the extensibility model and give two small examples of scripts — enough, we hope, that the reader will gain a general idea of how SLICE extensions are written. We end with a discussion of our future plans.

2. SLICE APPLICATIONS

The software architecture of SLICE facilitates the development of Tablet PC applications for education, and has enabled us to write a variety of such applications. The reader is encouraged to view these as just examples of what might be done, and to give SLICE a try him- or herself. The programming of extensions is discussed in section 3, and in more detail on our website.

A note on terminology: Essentially all functionality is provided by scripts in this system, and is therefore, in some sense, an "extension." We use the term "application" to re-

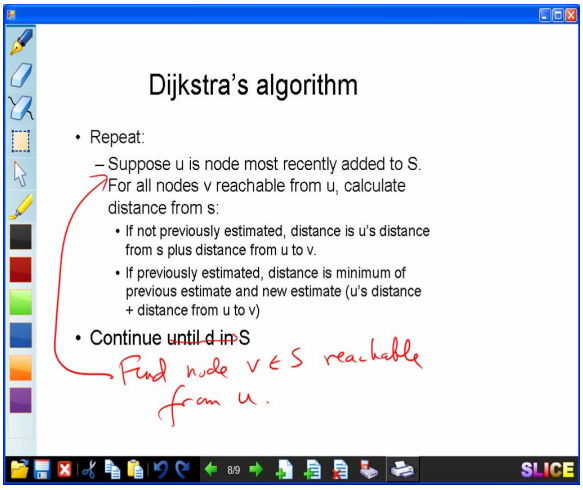

Figure 1: Screen shot of single-user presentation application

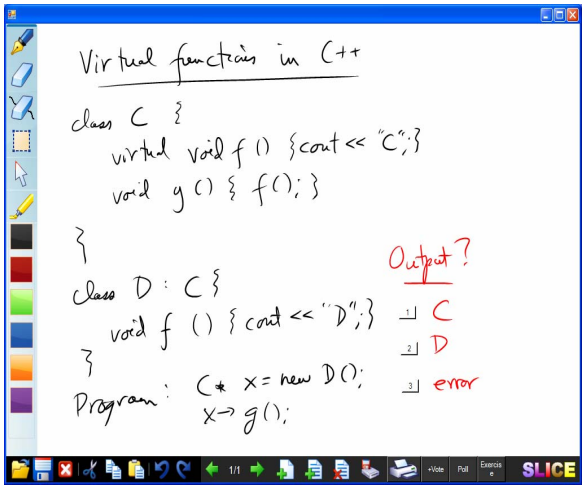

Figure 2: Screen shot of classroom interaction application

fer to a suite of scripts implementing a complete, stand-alone system, and the word "feature" to refer to a small enhancement of an existing system. We use the term "extension" when we do not care to make this distinction.

2.1 Single-user presentation

Instructors have begun to note the advantages of using Tablet PCs, with computer projectors, over whiteboards, overhead transparencies, or PowerPoint presentations. A projected image is usually easier to see than a whiteboard or blackboard, especially in a large classroom. Compared to overhead transparencies, presentations made using Tablet PCs are easier to correct as well as to save and post. Prepared presentations, such as PowerPoint, are too static for a class (even as they may be ideal for more formal presentations), but they do not exclude the use of Tablet PCs; many instructors find that the ideal solution is to use ink annotations over sparsely-populated PowerPoint presentations [8].

The basic features of a presentation system vary little among the systems cited earlier. It must provide for importing prepared presentations (such as PowerPoint or PDF), changing pen colors, erasing, adding new pages, printing, and so on. It is important that the user interface not be cluttered; giving a presentation is very different from preparing a document, and the typical layout of small buttons, multiple toolbars, and drop-down menus is too complex and cumbersome. Tablet PC-based presentation systems favor large buttons controlling a limited set of features.

A screenshot of the SLICE presentation application is shown in Figure 1. In addition to the standard features mentioned above, we have added a "laser pointer," which draws a bright red mark that disappears when the pen is lifted from the screen, and a "clone slide" feature, which makes an exact copy of the current slide (useful, for example, when hand-simulating an algorithm). The "clone slide" button has the additional feature that it copies to the new slide only the *selected* ink on the current slide, if the user has selected ink.

2.2 Networked presentation

Researchers in this field envision a future in which both teachers *and students* are equipped with networked, pen-enabled computers, and they are exploring the possibilities that such an environment presents. It is for this kind of exploration that we have created an extensible platform. Networked features, like those of the single-user application, are programmed using Python scripts. Thus, many types of interactions can be added with relative ease.[1] Here we discuss a particular, and fairly modest, set of features that we are currently using. (See section 4 for a discussion of our future plans.)

Figure 2 gives a screen shot of such an application. It is very similar to the single-user application, with the addition of a few buttons. In this application, the lecturer's initial set of prepared slides, if any, are transmitted to the students as they sign on. Specifically, once the lecturer has loaded any initial slides, she clicks the "start class" button; students who have signed in get the slides, and any students who arrive later get the same set of slides. However, the lecturer's notes during the lecture are not sent to the students. (This is not a technical limitation but rather a pedagogical decision, implemented in scripts.) The lecturer may, on the other hand, send entire pages to all the student machines by clicking one of two buttons: "Poll" and "Exercise."

Polls work as follows: On any slide, the lecturer can add choice buttons by clicking the "Vote" button. (Figure 2 shows a page after the instructor has added several such buttons, in the southeast part of the screen.) She can then click the "Poll" button. The page will be sent to the student machines as is, with one difference: a "Send Vote" button will appear at the bottom of the page. At the same time, a frame displaying the results of the poll will appear on the instructor's machine; it will be updated as students click their "send vote" buttons.

The "start exercise" button is simpler. Exercises are written work that the instructor wishes the students to do, such as filling in the values in an array. The difference between an exercise and a poll is that an exercise is not graded in real time; that is, the instructor does not see the students' work (but see section 4 for more on this issue). Thus, giv-

[1]Relative, that is, to programming them directly in C# using the Ink API.

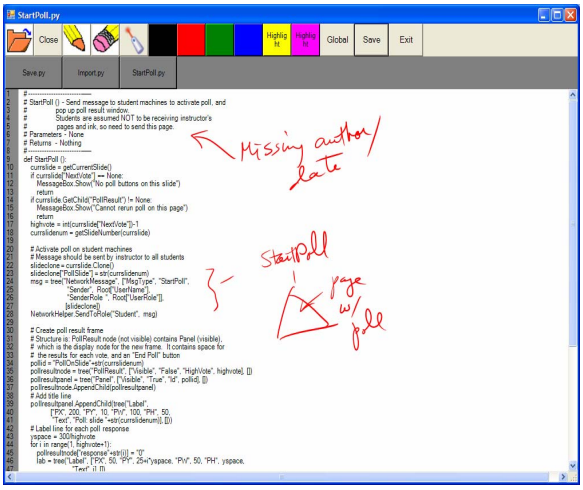

Figure 3: Screen shot of code review application

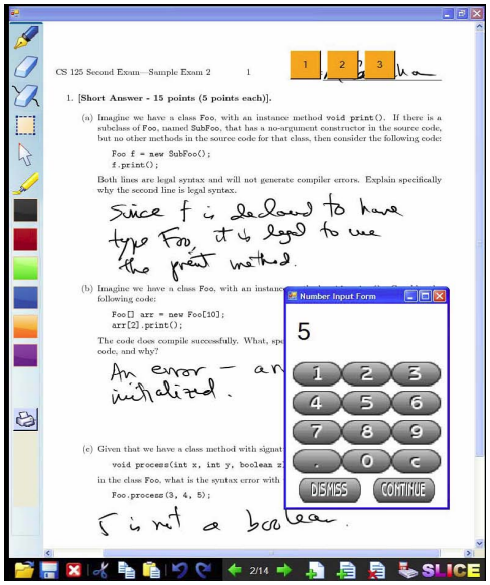

Figure 4: Exam-grading application

ing an exercise is simply a matter of sending a page to the students.

Another feature of this application is that there is actually a third party in the classroom: the display. In order that the instructor may look at private information, his tablet is not attached directly to the computer projector. Rather, the instructor's ink annotations are sent to another machine for display. (Another benefit of this arrangement is that the instructor's Tablet is not tethered to the cable for the display.) We note again that this arrangement is all under control of Python scripts, allowing it to be modified easily.

2.3 Code review

Our department has a required class for juniors, entitled "Programming Studio," in which students meet each week, in small groups, to review each other's code [13]. We have developed a SLICE application that allows the students to jointly view and annotate source files.

Figure 3 gives a screen shot of this application. Users open files using the "open folder" button. These are then transmitted to all other participants. The files are listed in tabs just below the top row of buttons. Each user's ink annotations are transmitted to all other users. The "Global comments" button pops up a separate window into which comments can be added that do not apply to specific parts of any of the files.

This application is under active development. We do not know what set of features will best facilitate discussion of each student's code. Some features that we intend to implement soon include ways of distinguishing among comments of different users (perhaps by ink color), and a facility to print only comments rather than entire files (since most parts of every file are unannotated). Future plans include the addition of roles, such as author, scribe, and reviewer, as described in McConnell [5], the course textbook; each role would have a distinct set of capabilities. Another enhancement is the ability to record "obligations," which are promises made by the author of the code; tracking such obligations is important in this class, as a large part of a student's grade is determined by his diligence in *rewriting* his code from one week to the next.

2.4 Exam grading

In many, if not most, CS departments, exams are graded as follows: Each person involved in grading is assigned a group of questions, which may together occupy several pages. The grader takes a pile of of exams, scores those questions, records the scores on the front of the exam booklet, and then returns the pile and picks up another. When all questions in a group of exams have been graded, the grades are summed and the score entered on a computer. Normally, grades for individual questions are not entered, because of the extra work involved.

Transferring this procedure to a Tablet PC, the separate data entry process is avoided. This has the immediate effect that the instructor can see averages *for each question*, as well as the total, providing a better picture of where the students had difficulties than when only total scores are recorded. It can also ease the graders' work, in that there is no flipping to the front of the exam and no final summing up of scores. (An exam-*taking* application could further facilitate this task, as multiple-choice questions could be answered using a dialog box, and graded automatically; we have not implemented this application, mainly because multiple-choice questions are rarely used in our classes.)

A screenshot from this applications is shown in Figure 4. The exam author has provided some meta-data about the exam, giving, for each question, its page number and total points. When the grader switches to a page, buttons for the questions on that page appear in the upper-right of the screen. When the grader has determined the score for a question, he clicks the button and a numeric keypad pops up, with which the grade is entered. When all questions have been scored, the system fills in the total grade on the front page of the exam, and writes it to a file. Clicking the "next" button fetches the next exam.

3. EXTENSIBILITY IN SLICE

The SLICE architecture may be considered as an extreme implementation of the Model-View-Controller paradigm, with

this additional feature: the model can be manipulated by scripts programmed by the user.

To be more specific, SLICE is an example of what we call an "explicit state" system. The entire state of the system is stored in a single data structure, whose access operations are few and simple. The state includes all slides, ink strokes, and intermediate results — not just what is showing on the display. (The state data structure is certainly liable to be large, but there is no reason why it need be complicated.) Scripts use the state operations to change the state, potentially causing the display to change. Scripts are invoked when a button is clicked, or on other user events such as an ink stroke being drawn.

The distinguishing feature of this architecture is that the state is represented by a simple data structure with a few operations. (Specifically, it is represented as a tree, as we describe shortly.) This has several advantages for user-level extensibility. To extend an existing application, the user needs to know the structure of the state for that application (as determined by the original author of the application), and a small number of state operations. By contrast, with a conventional architecture, the user needs to understand many classes and their interfaces, both those written specifically to represent the state of this application and those provided by the underlying system (.NET, in our case).

Further, by eliminating "hidden state," some operations that might cause problems for extension-writers work automatically. One example is saving: the document consists of exactly what is in the system state; no matter what data the extension-writer adds, it will be saved when the system state is saved. Another example is undo/redo; since all state changes of interest are stored in one data structure, the system need only record each operation on this data structure. Similarly, replaying a session, such as a lecture, is a straightforward matter of repeating the sequence of operations on this state.

We will not attempt to supply full details here, but we can give enough details to show a simple example and, hopefully, have it be understandable.

The data structure we use for the state is a tree whose nodes are labelled with a name and a set of attribute/value pairs. The reader will note that this data structure corresponds precisely to XML trees. Indeed, we use XML as our external representation, and the internal representation is nearly isomorphic to it. In the following discussion, we will use XML format to show the internal state.

Consider the presentation application. The root of the tree has a child named `Lecture`, which has a child named `Slides`, which in turn has a list of children each labelled `InkPanel`:

```
<Lecture>
   <Slides>
      <InkPanel> ... </InkPanel>
      <InkPanel> ... </InkPanel>
        ...
   </Slides>
</Lecture>
```

The `InkPanel` nodes are the individual slides in the lecture; their children will consist mainly of `Stroke` nodes representing the ink strokes drawn by the instructor. `InkPanel` is a built-in node type representing a panel on which one can add ink annotations (as well as placing buttons, text boxes, etc.). `Lecture` and `Slides` are not built in; their significance lies only in how scripts use them to navigate through the tree. The order of slides in this application is given by their order within the list of children of the `Slide` node (that is, there is no additional `SlideNumber` attribute in these nodes). Exactly one `InkPanel` node has attribute `Visible` equal to `True`, while the others must have `Visible` = `False`. Thus, if we are on page one, the above tree would actually be:

```
<Lecture>
   <Slides>
      <InkPanel Visible=True> ... </InkPanel>
      <InkPanel Visible=False> ... </InkPanel>
        ...
   </Slides>
</Lecture>
```

3.1 Script examples

Bear in mind that the underlying system knows nothing about presentations or sequencing of slides. The scripts determine the sequence implicitly by the actions of the Next and Previous buttons. Consider the Next button (which is represented by a `Button` node in a separate part of the tree, which we have not shown). The script associated with this button has a simple job: Get the `Slides` node; find its unique visible child; find that node's successor, if any; make the visible node invisible and make the successor visible. Thus, the script fo the Next button is:

```
slides = Root.GetChild("Lecture").GetChild("Slides")
currslide = slides.FindChildByAttribute("Visible", "True")
nextslide = currslide.GetRightSibling()
if nextslide == None: return
currslide["Visible"] = "False"
nextslide["Visible"] = "True"
```

This script uses several tree-manipulating operations — `GetChild`, `FindChildByAttribute`, and `GetRightSibling` — whose meanings are, we trust, self-explanatory. The assignments in the last two lines give values to the `Visible` attribute in each node. There are all together perhaps twenty such operations that comprise the bulk of all the scripts that implement the applications described in section 2.

To give one more example, consider the "new page" button, which adds a blank slide after the current slide. The script finds the current slide as above, then creates a new `InkPanel` node, and adds it to the `Slides` node:

```
slides = Root.GetChild("Lecture").GetChild("Slides")
currslide = slides.FindChildByAttribute("Visible", "True")
newslide = tree("InkPanel", ["Visible", "True"], [])
currslide["Visible"] = "False"
slides.InsertAfter(newslide, currslide)
```

The `tree` function used here creates a tree with a given name, a list of attibutes, and a list of subtrees. In addition, this script uses tree operation `InsertAfter`, which inserts a child node after another child.

In providing for extensibility, our goal is to encourage experimentation. Even the functionality of page changing, which we have just illustrated, admits many variations. The presentation application has only one kind of navigation: going forward or backward by one slide. Many instructors will recognize that this is not always sufficient; often one wants to go quickly to a slide at a greater distance. Simple buttons that move forward or backward by multiple slides might suffice. Bringing up a page of thumbnail views might be a

good solution. If there are only a small number of slides that might be the target of non-local navigation, as is often the case, a method of marking slides as "important" and putting those in a drop-down menu might be easier to use. These are the kinds of ideas that our model is intended to be helpful in implementing.

3.2 Networked applications

The pedagogical features that most interest us involve the scenario in which students are equipped with Tablet PCs, creating a fully interactive environment. Our architecture also facilitates the creation of such features. In our model, users communicate messages which are just trees with name "NetworkMessage" and a "MessageType" attribute. A handler, also written in Python, interprets messages as they arrive. In practice, messages often have the form "here is a piece of my state that you should copy into your state;" this occurs, for example, when one user sends an ink stroke to another. These are simple to implement: the sender copies the subtree into a message with message type "`AddTree`", and with another attribute describing the location (e.g. `SlideNumber=12`); the receiver finds the location and appends the tree.

3.3 Summing up

Obviously, claiming that all scripts are easy to write would be tantamount to claiming that all *programs* are easy to write, since there is no bound on the potential complexity of an extension. We earlier gave a line count providing a general idea of the amount of code needed to implement functionality. We believe that the *process* of adding extensions — in which scripts can be changed or added by users in the field — and the *learning curve* for doing so — in which a small set of tree operations is used in place of a large set of interfaces — will facilitate the creation of features by users who would blanch at the idea of modifying a 10,000-line C# program.

4. CONCLUSIONS AND FUTURE WORK

We have described a new system for developing Tablet PC applications. The system is designed to facilitate experimentation with pedagogical features. We hope this system will help researchers converge on principles of design of such features. Another area in which we hope our system can contribute is the development of topic-specific interactive Tablet PC applications.

Aside from general enhancements to the framework and development of new features, we are currently concentrating on a potential use of Tablet PCs that appears not to have received very much attention. In a networked classroom, the teacher may be able to learn much about the "state of mind" of the class by observing the students' note-taking behavior. We are exploring the notion of a "teacher's dashboard," a summary display of this information. The dashboard might show thumbnails of the screens of all or a selected set of students; however, this presents a difficult visualization problem. An alternative is to focus on physical aspects of the students' writing, such as speed and quantity of ink. Either way, the data can be gathered unobtrusively, and may give the teacher an idea of which students are out of step with the class, or whether the entire class has become disengaged.

The SLICE system, including the four applications described in this paper, and full documentation, can be downloaded from our website, `slice.cs.uiuc.edu`.

5. ACKNOWLEDGMENTS

Contributors to the current code base of SLICE include Mike Woodley, Matt Klupchak, Alex Kurilin, Victor Huang, and Abdullah Akce. Support for this work was provided by Microsoft under the 2005 Tablet PC Technology, Curriculum, and Higher Education program. The authors wish to thank the anonymous SIGCSE reviewers for their very helpful suggestions.

References

[1] Anderson, R. J., Anderson R., Simon, B., Wolfman, S. A., VanDeGrift T., Yashuhara, K. Experiences with a Tablet PC Based Lecture Presentation System in Computer Science Courses. Proc. 35th SIGCSE. Norfolk, Va. 2004. 56–60.

[2] Berque, D. Pushing Forty (courses per semester): Pencomputing and DyKnow tools at DePauw University. in **The Impact of Tablet PCs and Pen-based Compouter on Education: Vignettes, Evaluations, and Future Directions**. Purdue University Press. 2006.

[3] Knuth, D.E. **The TeXbook (Computers and Typesetting, Volume A)**. Addison-Wesley. 1984.

[4] Lord, S.M., Perry, L.A. Tablet PC — Is it worth IT? A preliminary comparison of several approaches to using Tablet PC in an engineering classroom. Computers in Education Journal, 42(3). July–Sept. 2007. 66–75.

[5] McConnell, S. **Code Complete, Second Edition**. Microsoft Press. 2004.

[6] Peiper, C. Warden, D., Chan, E., Campbell, R., Kamin, S., and Wentling, T.L. Applying Active Space Principles to Active Classrooms. Workshop on Mobile Peer-to- Peer Computing at Conf. on Pervasive Computing (PerCom 2005). Hawaii. March, 2005.

[7] Peiper, C., Warden, D., Chan, E., Capitanu, B., and Kamin, S. E-Fuzion: The Development of a Pervasive Educational System. Proc. 10th Intl. Conf. on Innovation and Technology in Computer Science Education (ITiCSE). Lisbon, Portugal. June 2005.

[8] Price, E. Simon, B. A survey to assess the impact of Tablet PC-based active learning: Preliminary report and lessons learned. In **The impact of Tablet PCs and Pen-based Technology on Education (WIPTE 2007)**. Purdue U. Press. 2007. 97-105.

[9] Stallman, R. M. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. MIT Artificial Intelligence Laboratory. AIM-519A. 1979 (updated 1981).

[10] Wentling, T.L. Parkz, J., C. Peiper, C. Learning gains associated with annotation and communication software designed for large undergraduate classes. J. of Computer Assisted Learning 23. 2007. 36–46.

[11] Wilkerson, M., Griswold, W.G., Simon, B. Ubiquitous Presenter: Increasing student access and control in a digital lecturing environment. Proc. 36th SIGCSE. St. Louis. 2005. 116–120.

[12] Willis, C. L., Miertschin, L. Tablet PC's as instructional tools or the pen is mightier than the 'board!. Proc. 5th Conf. on Information Technology Education (CITC5 '04). Salt Lake City, UT. 2004.

[13] Woodley, M., Kamin, S.N. Programming Studio: A Course for Improving Programming Skills in Undergraduates. Proc. 38th SIGCSE. Covington, Ky. 2007. 531–535.