A FRAMEWORK DEVELOPING CROSS-PLATFORM PEN-BASED CLASSROOM APPLICATIONS WITH AN AUTOMATED MEASURE OF VOCAL PARTICIPATION

BY

WADE A. FAGEN

DISSENNATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Associate Professor Sam Kamin, Chair
Professor Klara Nahrstedt
Associate Professor Karrie Karahalios
Professor Mark Guzdial, Georgia Institute of Technology
Abstract

This thesis presents the development and deployment of educational software applications on a heterogeneous set of consumer-level devices. To do this, the SLICE framework was created to allow for application developers to quickly build software to target pen- and touch-based devices. This thesis discusses both the internals of the framework as well as how a developer may develop an application on the framework. Additionally, two flagship applications were built using the SLICE framework: an application to facilitate code review and an application to replace PowerPoint in large lectures. As part of the deployment of both applications, semester-long experiments were done to measure the impact of the deployment. In one, student audio was used as a metric to measure “vocal participation” – a metric defined in this thesis. In the other, students motivated the use of the technology in lecture through surveys given throughout the semester. One key result presented in this thesis is student vocal participation increased by nearly 20% when Tablet PCs were introduced (p < 0.001).
Acknowledgments

Many thanks to all those who helped me, personally and professionally, throughout the entire PhD and dissertation process! Specifically, I want to thank my thesis adviser (Sam Kamin), each of my committee members (Klara Nahrstedt, Karrie Karahalios, and Mark Guzdial), my parents (Ray Fagen and LaRue Fagen), my brother (Paul Fagen), and my sister (Sara Fagen). Additionally, I could not have done it without the support of the amazing friends I have met here at Illinois and the support of the Computer Science department. Yayy!
4.4. Results ........................................................................................................................................... 60
4.5. Conclusion ..................................................................................................................................... 64

Chapter 5. SLICE in Classroom Lectures .......................................................................................... 66
  5.1. Introduction .................................................................................................................................... 66
  5.2. A Large Lecture Experience with the Lecture App ................................................................. 68
  5.3. Summer 2012 Experiment: What is an optimal lecture format? ........................................... 72
  5.4. Limitations ..................................................................................................................................... 78
  5.5. Conclusion ..................................................................................................................................... 78

Chapter 6. Conclusion ......................................................................................................................... 80
  6.1. Overview of Thesis ......................................................................................................................... 80
  6.2. Future Work ................................................................................................................................. 81
  6.3. Final Remarks ............................................................................................................................... 82

References ............................................................................................................................................ 83

Appendix A. Complete Survey Forms .................................................................................................. 87
List of Tables

Table 1: Partial list of attributes used in a SLICE application’s XML to describe and interact with the user interface .................................................................................................................................................. 12
Table 2: Overview of user interface components available in the SLICE framework ........................................ 14
Table 3: Full set of XML attributes in the SLICE framework ................................................................................. 15
Table 4: Non-static methods of the TreeNode class in the SLICE API ................................................................. 19
Table 5: Static functions of the TreeNode class in the SLICE API ........................................................................... 20
Table 6: Functions in the Slice class of the SLICE API ......................................................................................... 20
Table 7: Functions in the SliceCloud class of the SLICE API .................................................................................. 21
Table 8: The suite of methods in the Component class that are used by the ComponentTreeNode class to interact with platform-specific SLICE frontend components ................................................................................................................. 40
Table 9: Full list of events handled by the SLICE core handlers .......................................................................... 43
Table 10: Categories of messages logged within the SLICE framework. ............................................................. 46
Table 11: Native platform-specific components used in each SLICE frontend .................................................... 47
Table 12: Basic information on audio recordings .................................................................................................. 56
Table 13: Statistical information on $\epsilon$ across different windows with and without Tablet PCs... 62
Table 14: Survey results from the first CS 241 Summer 2012 survey (Week #2) ........................................... 74
Table 15: Survey results from the first CS 241 Summer 2012 survey (Week #2) ........................................... 74
Table 16: Survey results from the second CS 241 Summer 2012 survey (Week #3) ................................. 76
Table 17: Survey results from the third CS 241 Summer 2012 survey (Week #5) ......................................... 77
List of Figures

Figure 1: Classroom Presenter from University of Washington .................................................. 7
Figure 2: The complete source code to a SLICE application, alongside the application running on Windows 7 with several strokes drawn on the InkPanel ......................................................... 13
Figure 3: Boilerplate XML code for a SLICE application .......................................................... 23
Figure 4: Continuation of Figure 3; XML attributes were added and highlighted in red ............ 23
Figure 5: Continuation of Figure 4; XML nodes for GUI components highlighted in red ......... 24
Figure 6: Continuation of Figure 5; XML attributes linking a Frame to a Panel highlighted in red ............................................................................................................................................. 24
Figure 7: Continuation of Figure 6; XML attributes for scriptable actions highlighted in red ... 25
Figure 8: Continuation of Figure 7; XML attributes for referencing a JavaScript file highlight in red ............................................................................................................................................... 25
Figure 9: Complete JavaScript for the student version of a sample “clicker” application ......... 26
Figure 10: Complete XML for the lecturer version a sample “clicker” application ................. 26
Figure 11: JavaScript method for the lecturer “clicker” application ........................................ 27
Figure 12: JavaScript method for the lecturer “clicker” application ....................................... 27
Figure 13: A simple, complete clicker application with a student clicker application and a lecturer answer viewer developed with less than 80 total combined lines of code. ....................... 28
Figure 14: The Model-View Controller design pattern of SLICE framework ......................... 30
Figure 15: The SLICE framework with the SLICE core split into subcomponents and interactions between subcomponents labeled. A brief description of each interaction is detailed below ...... 35
Figure 16: Two separate representation of nodes, both having the type of “Button” .......... 36
Figure 17: The class hierarchy of classes representing nodes in the SLICE state tree .......... 37
Figure 18: Two student instances of the SLICE Lecturer Application on heterogeneous platforms. Left: Android tablet; Right: Windows 7 Tablet PC .................................................. 47
Figure 19: slice.app file for the SLICE lecturer app ............................................................... 49
Figure 20: The SLICE Launcher’s app selection screen running on Windows 8 ...................... 50
Figure 21: Typical setup of a CS 242: Systems Programming section ..................................... 51
Figure 22: The Code Review Application built using SLICE for CS 242 ............................... 53
Figure 23: Two sections of the CS 242 Software Studio course (a): No Tablet PCs used (b): Tablet PCs used.................................................................................................................................................................................. 54
Figure 24: RCA VR5220 Digital Voice Recorder, used for recording audio during CS 242 sections. (The lapel microphone that was used is not shown.) .......................................................... 55
Figure 25: Experiment schedule for recording and using Tablet PCs in CS 242 during Spring 2011.................................................................................................................................................................................. 55
Figure 26: Five synchronized waveforms from a single section ........................................... 58
Figure 27: A visual overview of the “vocal participation” metric, $\varepsilon$ ........................................ 59
Figure 28: Average “vocal participation” ($\varepsilon$) per meeting. (Tablet / Non-Tablet: $p < 0.001$) .... 62
Figure 29: Average $\varepsilon$ per meeting, grouped by the six sections that were recorded as part of this study.................................................................................................................................................................................. 63
Figure 30: Average $\gamma$ per student, averaged across all meetings ......................................... 63
Figure 31: Student D's turns in the discussion from two different meetings, each approximately 75 minutes in length.................................................................................................................................................................................. 64
Figure 32: An iteration of the Lecturer Application in use in a classroom using a separate dashboard display.................................................................................................................................................................................. 66
Figure 33: A student view of a Lecture Application where a student’s work has been completed by the instructor. .................................................................................................................................................................................. 67
Figure 34: Lecturer’s view of the SLICE “Lecturer Application” .......................................... 70
Figure 35: Four students doing the same exercise, viewed by the professor......................... 71
Chapter 1

Introduction

1.1. Introduction

The focus of this thesis is the development and deployment of educational software applications on a heterogeneous set of consumer-level devices. The specific devices this thesis will focus on are those that generally have the ability to accept input through either a stylus pen or a finger – so called “tablets”. Due to this unique input feature, these devices have great potential to play an impactful role in the future of education.

A major component of this thesis is the development of a robust software framework that allows for a single application to be run natively on both Windows and Android and run within the Java environment for Java-supported devices. This software, SLICE (an acronym for Students Learn In Collaborative Environments), was used as the developmental framework for all of the applications used throughout this thesis. The SLICE website (http://slice.cs.illinois.edu) contains downloads for the SLICE framework and a gallery of applications running on top of SLICE.

This thesis details two of the major applications deployed using SLICE: an application that was used to increase the vocal participation of students in a code review course (Chapter 5) and the flagship “Lecturer Application” that is used to replace PowerPoint in large lectures (Chapter 6). The Lecturer Application specifically has gained significant adoption, having been used as the primarily lecture tool in CS 105, CS 225, CS 241, and CS 421 in the Spring 2013 semester alone.

Along with the development and deployment of the applications, research studies were done to measure the impact of the introduction of Tablet PCs using SLICE into the various classrooms. In relation to the code review course, this thesis defines a new metric for using audio as a proxy to measure a student’s “vocal participation” in a discussion. Through the analysis of the recordings of six discussion sections each week for ten weeks, the results
presented in Chapter 5 shows that the introduction of Tablet PCs and SLICE resulted in students being nearly 20% more involved or “engaged” (p < 0.001).

The rest of this chapter will detail the motivations for this research (Section 1.2), the contributions that this thesis provides to the general field of educational research (Section 1.3), the background of the SLICE technologies (Section 1.4), an overview of related work (Section 1.5) and a brief overview of each chapter (Section 1.6).

1.2. Research Motivations

In recent years, a number of researchers have studied the effectiveness of active and collaborative learning technology in classrooms [1, 2, 3, 4, 5, 6]. These technologies often make use of laptop computers, Tablet PCs, iOS or Android-based cell phones and tablets, and electronic voting systems (EVS) [7]. Based on the success in the classroom with these technologies, many higher-education institutions have adopted the cheapest form of this technology: EVSs, including i>Clickers, eInstruction clickers, and others [2]. Case studies and research alike have shown EVSs to be effective, though often note limited functionality as a drawback to complex active learning exercises [7].

As a contrast to EVSs, researchers have developed and studied active learning applications on pen-enabled (Tablet PCs) and touch-enabled (cell phones and tablets) devices [7]. Nearly all published research uses custom-developed applications running on a homogeneous platform. Due to the use of a single platform running a purpose-built application, the work is difficult or impossible to deploy in your own classroom. Instead, only a very small number of tools have been widely adopted at more than a single university.

One suite of technologies that has seen wide adoption is those that replace PowerPoint with the ability to annotate slides while lecturing. Tools such as Classroom Presenter [8], Ubiquitous Presenter [9], or the commercial DyKnow product [10] each provide lecturers with the ability to present on their own Tablet PC and have varying degrees of student interactions. However, each of these tools runs only on Windows.

The motivation of this research is to produce a framework and applications that can be used on both existing platforms as well as platforms not yet developed. It is important that this
framework provides applications with as much functionality as existing, platform-specific applications. Along with being able to build full-featured applications, the ability to change the applications for purpose-specific tasks is also critical. The framework presented in this thesis attempts to do both on cheap, consumer-level devices such as a modern cell phone or tablet.

1.3. Contributions

The framework that is the focus of this thesis, SLICE or Students Learn In Collaborative Environments, has been in use each semester at the University of Illinois since 2006 [11]. However, SLICE has only used internally and has worked only on Windows-based Tablet PCs. Realizing that the modern classroom is filled with cell phones, tablets, laptop computers, and other intelligent devices, SLICE has been extended to be a cross-platform, cross-device framework for developing applications primarily focused on educational use.

The thesis describes a programming environment where developers can develop an application once on the SLICE framework and deploy it seamlessly across a heterogeneous set of consumer devices. In the current scope, this includes the use of all desktops, laptops, and tablet PCs with a modern operating system in addition to Android-based phones and tablets. The research questions that this thesis answers are:

- Can pen- and touch-enabled software running on a heterogeneous set of devices from Tablet PCs to cell phones be used as a tool in active learning activities during an undergraduate college class?
- Can pen- and touch-enabled software running on a heterogeneous set of devices from Tablet PCs to cell phones be used as a tool to achieve real-time collaborative learning?
- Part of the research is developing a software framework. How can this help others study active and collaborative learning with technology in classrooms?
- What applications can be built using a software framework designed for active and collaborative learning that runs on a heterogeneous set of devices?

To answer these questions, this thesis presents several different contributions:
• Development of the SLICE framework, deployable natively on Windows and Android, as well as any operating system supporting Java.

• Development and deployment of a “Code Review Application” in the Software Studio (CS 242) course at The University of Illinois

• Analysis of over 200 hours of audio recorded in the Software Studio course to measure the impact of the introduction of Tablet PCs into a code review discussion.

• Providing support to users that use the SLICE framework, such as the use of the “Lecturer Application” in CS 105, 225, 241, and 421.

• Promoting SLICE and providing support for external application developers.

1.4. Background on the SLICE Framework

The SLICE research group, an acronym for Students Learn In Collaborative Environments, had its beginnings in a project that was starting in Fall 2001. At that time, work was being done on cutting edge pervasive computing research that focused on “applying active space principles to active classrooms” [12]. This project, referred to as eFuzion (also stylized as e-Fuzion or EFuzion), was a classroom presentation tool that allowed instructors to load lecture slides and annotate those slides with simple shapes and ink strokes; students could receive the slides and annotations on their machines [13]. The first documented deployment of eFuzion was in a Spring 2002 pilot course of CS 300. After a successful initial usage, eFuzion was then used in Summer 2002 to show a half letter grade improvement when students used eFuzion compared to a similar class without eFuzion [14].

After the success of the eFuzion application in 2002, the research group received grants for tablet PC technologies from Microsoft Corporation and Hewlett-Packard. With the grant, the eFuzion application was updated to incorporate active digitizer technologies using the .NET framework and the Microsoft.Ink API. In the years following, eFuzion was modularized and, in 2005, eFuzion v3 was released as a “system for developing Tablet PC applications for education” [15]. The release of eFuzion v3 didn’t incorporate any new functional changes to the classroom presenter, but laid out a software framework to build applications that ran on .NET-enabled Tablet PCs. The core structure of this framework has been modified slightly since it was introduced in 2005, but the separation of the layout of the user interface, the application-specific
logic in user scripts, and the core functionality into logically separate pieces remain the same. Each of these components will be discussed in detail later in this document.

Having a software framework to develop applications for education, eFuzion no longer specifically referred to the classroom presenter tool that was developed initially. Instead, eFuzion could support a variety of applications or “apps”. The first of these applications simply mimicked the functionality of the original eFuzion, a PowerPoint-like classroom presenter that allowed annotation of lecture slides by the lecturer. However, with the separation of code between the application functionality and the core eFuzion framework, the authors were able to develop customized lecturer applications that met the specific educational objectives of individual professors [13]. After 2005, publications began referring to the eFuzion system or SLICE, the name it is known by today.

In 2007, an application was developed for SLICE to study an experimental section of an introductory programming course at The University of Illinois [16]. The application was motivated by the observation that students coming into introductory programming courses with differing levels of experience and skill-level; some students needed almost no guidance while others feel that “they, in effect, have no teacher, in that the things the teacher is saying and doing are far removed from the student’s needs.” Instead of having a traditional lecture, the section consisted of a series of objectives that each followed a four-stage design: a pre-flight (introduction to the objective), content presentation, self-assessment, and a post-flight (recap). Each student had their own Tablet PC to complete these objectives and the lecturer had a separate display that monitored the progress of the class. The lecturer was able to design “synchronization points” into the class, where students could not continue until some (or all) of the students reached that point. When students reached these points before others, they were often encouraged to help others understand the objective until the entire class reached the “synchronization point”.

During the 2007-2008 academic year, a second experimental classroom also used a SLICE application [17]. In this work, a high-school algebra course used the following classroom setup: each student had a tablet PC and the lecturer had a private display (“dashboard”) to monitor the progress. Instead of self-paced objectives, students in the high-school algebra classroom heard a small (approximately twelve minute) lecture on the subject material and then
spent the remainder of the class working on worksheets of practice problems. In writing about this use of Tablet PC technology in the classroom, the authors noted that the high-school teacher didn’t change his style of teaching much at all in the experimental classroom. Instead, SLICE provided a way to passively monitor the student’s work from a single central location instead of walking around and looking at the work of his students at their desks. An unexpected benefit of using tablets, reported by the high school math teacher, was that some students who had felt shy about showing their work in person had no embarrassment in showing their work through the tablets. The work discussed in this thesis picks up from here, with maintenance of the original version of SLICE followed by the development of a new, platform-independent version of the framework.

1.5. Related Work

Researchers have looked at techniques involving active and collaborative learning in large classrooms for several decades and surveys of the work show that this poses special challenges [18]. With these special challenges, researchers have also examined the unique opportunities that large classes present [19]. With the introduction of laptop devices, researchers began looking into how technology could enhance the individual educational experience of students in a classroom [6]. Often, these technologies are purpose-built applications focused on accomplishing one specific goal [1, 2, 3, 4, 5].

Over the past decade, the most widely adopted and successful integration in using technology to promote active learning has been multiple choice clickers [20, 21, 22, 23, 24, 25, 10]. Specifically, [23] notes that lecturers “have used EVSs in their teaching without radically changing the traditional lecture format. With this method, standard lectures are supplemented with questions, and students’ responses provide feedback to both students and staff on the learning process.” However, other researchers have pushed the EVS interface to various extremes, including using the systems:

- "as a diagnostic tool in large group tutorials" and "[to ask] students questions about the material covered in earlier traditional teaching sessions" [24]
- as exam preparation [21, 25]
- as a method to test assigned readings [6]
- as part of group discussions [10]

Several commercial companies have been created to provide EVS technology to classrooms. Some of the most popular EVSs, usually known simply as “clickers”, are i>Clicker (used at The University of Illinois) and eInstruction clickers. However, researchers have found the expressiveness of multiple choice questions lacking [7].

With a laptop or tablet PC interface, the expressiveness of in-class interactions increases dramatically. Since the introduction of the Tablet PC in 2001, the most popular applications of the technology in a classroom have been interactive lecturer applications. These applications, similar to the original e-Fuzion system, allow for lecturers to load a presentation slide set and use a tablet computer to write and annotate on their slides [3]. Some of these applications include Ubiquitous Presenter from UCSD and the commercial product DyKnow [26]. Advanced versions of these systems, such as recent releases of Classroom Presenter by The University of Washington, allows for note taking by the student directly on the lecture slides, alongside the instructor [4].

Figure 1: Classroom Presenter from University of Washington
Additional work has been done towards a general, technology-enabled classroom environment with less emphasis on a specific product of application. One of the largest research projects in this space was known as Classroom 2000, recently renamed to eClass, out of Georgia Tech [27]. Other projects, such as the Gaia project out of The University of Illinois, look into the ever broader field of “active spaces” [28]. Many of these visionary research projects include prototype of several of the applications that have been developed with SLICE.

This thesis focuses on two separate bodies of significant research that have not traditionally had much overlap. The first area, technology tools for active and collaborative learning, has been an active area of research in both education departments and computer science departments for over ten years as discussed in the preceding section. However, the second area, the development and deployment of software frameworks, is heavily studied in software engineer research and related fields but has had very little study related to frameworks focused on active or collaborative learning.

Some of the best known frameworks include Microsoft .NET [29], Oracle’s Java [30], Adobe AIR [31], and Mozilla’s XULRunner [32]. In each of these frameworks, a core platform is developed to support developers developing applications on top of that base. In the case of AIR and XULRunner, these frameworks focus specifically on tasks related to rapid deployment and seamless deployment. Throughout the rest of this thesis, the thesis will examine how the SLICE framework incorporates some of the innovations contained in the most successful frameworks.

1.6. Overview of Thesis

The rest of this thesis is broken into five more chapters that explore the background work, the SLICE framework, and applications and studies done utilizing the SLICE framework. Specifically:

- **Chapter 2** provides a detailed description of the SLICE framework from an application developer’s point of view. Chapter 2 describes the XML document that represents the “state tree” internal to the SLICE framework, the JavaScript that a developer creates with
application-specific logic for their application. It builds an example application showcasing many of the features of the framework.

- **Chapter 3** provides a system-level description of the SLICE framework and how the internal components were developed to achieve a platform-independent “SLICE core” with light-weight “frontends” to allow for native execution of the SLICE framework on various platforms.

- **Chapter 4** details the “Code Review Application”, a SLICE application deployed in Software Studio course at The University of Illinois. As part of the application deployment, a large study was done on six sections of the course where audio was used as a proxy to measure vocal participation.

- **Chapter 5** details the “Lecturer Application” that is used in several CS course at The University of Illinois. As part of understanding the benefits of the application, a Summer 2012 session of CS 241 was designed to allow students to provide feedback to motivate what style of lecture was most helpful to them in learning the material. The surveys that were taken to motivate these results are also presented in this chapter.

- **Chapter 6** provides a brief discussion of the work presented in this thesis, a discussion of future work, and provides some concluding remarks on the work as a whole.
Chapter 2

Understanding and Building SLICE Applications

2.1. Introduction

SLICE, an acronym for Students Learn in Collaborative Environments, is a software framework that provides a suite of features and functionality that allows developers to rapidly develop cross-platform applications that involve classroom collaboration. The SLICE framework itself is not an application itself; you are unable to “run” the SLICE framework. Instead, it runs “SLICE applications”, often referred to as “SLICE apps” or simply “apps”, built on top of the features and functionality of the framework. This chapter will describe how to build a SLICE application. Details on how the SLICE framework is created and how the functionality described in this chapter is accomplished are provided in Chapter 4.

A SLICE application is made up of two main components: an XML document that describes the initial state of the application and JavaScript scripts that contain application-specific logic. In general, the XML document lays out all the components that may be displayed to the user. When special XML nodes have their visible attribute set to True (introduced in Section 2.2 and detailed in Section 3.3.1.1), the component that is described by the XML is displayed to the user. Section 2.2 details these special XML nodes and discusses the full details of this document.

As part of the XML document, some nodes contain special attributes that describe actions to take place when a user interacts with a visible component. For example, a Button node has a special OnClick attribute that, when the user clicks the button, invokes the script named by the OnClick attribute. These scripts, which can perform functions such as changing the XML that controls the user interface or sending a message over the network, are written in JavaScript and packaged with the XML to make a complete SLICE application. Section 2.3 details how an app developer would write these scripts and describes the “API” of the SLICE framework.

To illustrate, a sample “clicker” application is built in Section 2.4. This application uses an XML document to describe the layout, JavaScript to run application-specific logic, and the SliceCloud networking model to allow for real-time communication within a virtual classroom.
2.2. User Interface via XML

Every SLICE application includes an XML document. This XML document describes the initial state of the SLICE application. Internally, the XML document is loaded into a data structure referred to as the “state tree”. The state tree is described in detail in Section 3.3.1.1; its purpose is to store the state of the application; whatever is present in the state tree is always exactly what is displayed on the screen. For example, if a new button is added to the state tree (and is visible to the user based on the attributes of the new button), then the new button must be displayed to the user. The design choice of describing a user interface with XML and manipulating the XML via a “state tree” has also been chosen for other frameworks, including Microsoft .NET’s Windows Presentation Foundation (WPF) [33], Adobe AIR [31], and Mozilla’s XULRunner [32].

Every XML document is made up of a series of nested nodes starting from a root node. An example of a document with two nodes would be: `<Hello><World /></Hello>`. Additionally, XML nodes may contain attributes. An example node with two attributes is: `<Hello AttributeName="Value" SecondAttribute="Value" />`. The root node of every SLICE application’s XML file is a `<Slice>` node. The `<Slice>` node may contain a variety of attributes pertaining to the application as a whole. However, there are three special attributes that can only appear in the root node:

- **JSDefs**: A list of JavaScript files that contain the functions to be run when events occur.
- **Init**: A list of JavaScript functions to be executed immediately after the SLICE application has started and the initial XML has been completely read and displayed.
- **OnExit**: A list of JavaScript functions to be executed as the SLICE framework is exiting.

The `<Slice>` element’s children nodes will describe the application-specific user interface. The only requirement is that one child element must contain data about the application frames, the highest order user interface component. A frame is synonymous with an application window in Windows, Mac, or Linux, or a full-screen application on an Android-based device.

Information about all of the frames in a SLICE application is contained in a `<Frames>` element. Each individual frame is a `<Frame>` element as a child of `<Frames>`. Some of the
attributes that can be applied to the `<Frame>` and other user interface nodes are described in Table 1.

Since the frame is the highest order user interface control in SLICE, every other control is contained in a frame. Specifically, each visible `Frame` node must contain a `DisplayID` attribute that maps to the `Id` attribute of another node in the tree. For example, the for a `Frame` node linked to its content through an `Id` and `DisplayID` attribute pair:

```
<Slice>
  <Frames><Frame W="100" H="100" DisplayId="MainWindow" /></Frames>
  <Panel Id="MainWindow" />
</Slice>
```

The `DisplayID` and `Id` are only two of the attributes that have a special function with the SLICE framework. Table 1 shows a list of other attributes that have other special functions. Further, Figure 2 shows a simple SLICE application that contains an `InkPanel` inside of a single frame.
Beyond frames, the **InkPanel** (that was just referenced in the context of Figure 2) is a control that is central to many SLICE applications. The **InkPanel** allows for users to write on that user interface component with a tablet pen, a mouse, or their finger (on touch-based devices). In Chapter 4 and Chapter 5, the “Lecturer Application” and “Code Review Application” are described and both make extensive use of the **InkPanel** control. In the Lecturer Application, instructors of large classes use the app to annotate their slide set during lecture, to interactively work with students on solving in-class problems, and optionally to display student work publicly. In the Code Review Application, students use a SLICE app that displays a “digital whiteboard” of the student’s code.

Both the **Frame** and **InkPanel** nodes are considered a special type of node called a “container node”. Container nodes are like all other user interface nodes but have three special properties:

- Only container nodes may contain child nodes that correspond to user interface controls. A non-container node, such as a **Button**, cannot have another **Button** (or any other user interface element) as a child.
- All **X** and **Y** coordinates of child nodes of the container are relative to the top-left corner of the container node. If a **Panel** is located at \((X, Y) = (100, 100)\) and a **Button** inside the **Panel** is located at \((100, 100)\), its absolute location would be \((200, 200)\).
- If the container is invisible (**Visible** is set to **False**), all children are also invisible even if their visibility is **True**.
The third and final container node is the **Panel**. The Panel has no special properties besides being a container of other components, allowing for an application designer to easily group components together.

There are three non-container nodes. These nodes cannot contain other user-interface nodes within them and are sometimes referred to as “leaf nodes” as they must exist as leafs in the state tree. The three types of non-container nodes that have a user interface component are:

- **Button**, shows a button on the user interface allowing for a user to click on it for some action to be performed.
- **TextLabel**, shows a text label on the user interface allowing for text or an image to be displayed to the user.
- **Stroke**, shows a stroke on the user interface (often the result of a user drawing on an InkPanel)

All six of the components are detailed in Table 2. An example application using the components is designed later in this chapter in Section 2.4.

<table>
<thead>
<tr>
<th>Component</th>
<th>Container?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>Yes</td>
<td>Top-level user interface element. Must be a child of the <code>&lt;Frames&gt;</code> XML node.</td>
</tr>
<tr>
<td>Panel</td>
<td>Yes</td>
<td>Simple container element.</td>
</tr>
<tr>
<td>InkPanel</td>
<td>Yes</td>
<td>Ink-able container element. When a user draws or inks in this surface, Stroke nodes are generated as children nodes.</td>
</tr>
<tr>
<td>Button</td>
<td>No</td>
<td>Clickable button. The <code>onClick</code> attribute is used to invoke a JavaScript function when the button is clicked.</td>
</tr>
<tr>
<td>TextLabel</td>
<td>No</td>
<td>Non-clickable label containing text specified by the <code>text</code> attribute.</td>
</tr>
<tr>
<td>Stroke</td>
<td>No</td>
<td>Ink stroke, must be a descendent of <code>&lt;InkPanel&gt;</code>. Data about the contents of the Stroke is in child nodes a name <code>floats</code>, <code>pressure</code>, and <code>timerTicks</code>.</td>
</tr>
</tbody>
</table>

Table 2: Overview of user interface components available in the SLICE framework
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Inherited?</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>No</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>No</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>Yes</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Yes</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Id</td>
<td>No</td>
<td>---</td>
<td>Must be unique in the state tree.</td>
</tr>
<tr>
<td>DisplayId</td>
<td>No</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Enabled</td>
<td>Yes</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td>Special*</td>
<td>False</td>
<td>*: See Section 3.3.1.1.</td>
</tr>
<tr>
<td>BackColor</td>
<td>Yes</td>
<td>System default</td>
<td></td>
</tr>
<tr>
<td>ForeColor</td>
<td>Yes</td>
<td>System default</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>No</td>
<td>(Empty string)</td>
<td></td>
</tr>
<tr>
<td>Font</td>
<td>No</td>
<td>System default</td>
<td></td>
</tr>
<tr>
<td>FontSize</td>
<td>No</td>
<td>System default</td>
<td></td>
</tr>
<tr>
<td>TextAlign</td>
<td>No</td>
<td>LEFT</td>
<td>TOP</td>
</tr>
<tr>
<td>Center</td>
<td>No</td>
<td>False</td>
<td>Sets TextAlign to CENTER</td>
</tr>
<tr>
<td>Image</td>
<td>No</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>ImagePNG</td>
<td>No</td>
<td>---</td>
<td>Encoded in Base64.</td>
</tr>
<tr>
<td>PenColor</td>
<td>Yes</td>
<td>Black</td>
<td></td>
</tr>
<tr>
<td>PenWidth</td>
<td>Yes</td>
<td>5 pixels</td>
<td></td>
</tr>
<tr>
<td>Transparency</td>
<td>Yes</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>EditMode</td>
<td>Yes</td>
<td>StrokeAdd</td>
<td></td>
</tr>
<tr>
<td>OnClick</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>OnDoubleClick</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>OnRightClick</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>OnMiddleClick</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>OnMouseEnter</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>OnMouseExit</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>Init</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>OnExit</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>InkStrokeHandler</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
<tr>
<td>InkStrokeDeletingHandler</td>
<td>Yes</td>
<td>---</td>
<td>Scriptable attribute. See Section 2.3.</td>
</tr>
</tbody>
</table>

Table 3: Full set of XML attributes in the SLICE framework.
While some XML attributes have been described up to now as they relate to certain components, Table 3 presents a list of all of the XML attributes that have a special purpose within the SLICE framework. Most notable, some attributes have their values inherited from the parent onto the child. If an attribute is inherited, if the attribute is not locally defined at a specific node, the value of that attribute is determined by recursively look at the parent nodes.

2.3. Application Logic via JavaScript

Within the SLICE framework, all application logic must be defined by using JavaScript. With the exception of the initialization (Init) event that occurs when the SLICE application is loaded, the application’s JavaScript is called in response to events that occur within the SLICE framework. When developing application logic, you must inform SLICE which functions need to be invoked when a certain event takes place.

Table 3, introduced earlier, contains a section of XML attributes that corresponds to an invocation of an application’s script when an event takes place (eg: OnClick). These special XML attributes, known as “scriptable attributes”, simply provide a list of functions that should be invoked in JavaScript when the specific event occurs. This list may contain a single function – for example, OnClick=”ButtonClicked” runs the JavaScript function ButtonClicked() – or a list of functions separated by a pipe character. The XML attribute OnClick=”A | B” would run the JavaScript function A() followed immediately by the JavaScript function B().

A description of when the JavaScript functions are invoked for each scriptable attribute is detailed in the following list:

- **OnClick**, invoked when a user clicks the control.
- **OnDoubleClick**, invoked when a user double clicks the control. An OnClick will be invoked for the first click and then an OnDoubleClick will be invoked for the second click (and the OnClick will not be invoked).
- **OnRightClick**, invoked when a user uses the right mouse button to click a control. This is also invoked on stylus input where the user holds down the “alternate press” button.
- **OnMiddleClick**, invoked when a user uses the middle mouse button to click a control.
• **OnMouseEnter**, invoked when a user hovers over the control. This can occur with a finger or stylus when the hardware supports hover detection.

• **OnMouseExit**, invoked when a user stops hovering over the control. This can occur with a finger or stylus when the hardware supports hover detection.

• **Init**, invoked when the application is first initialized.

• **OnExit**, invoked when the application is exiting.

• **InkStrokeHandler**, invoked when an ink stroke is drawn.

• **InkStrokeDeletingHandler**, invoked when an ink stroke is deleted.

Additionally, the SliceCloud can be configured to have scripts called when certain network-related events occur. These callbacks are specified by calls to the SLICE API, discussed later in this section.

When any function is invoked (except **Init** and **OnExit**), a global variable in JavaScript named **Source** is set to the **TreeNode** of the component that was the source of the event. For example, the source of an **OnClick** event is the component that was clicked on by the user. In addition to the **Source** variable, before the script for the **InkStrokeHandler** function is invoked the global variable **StrokeNode** is set to the **TreeNode** of the stroke that was just added to the state tree as a result of the user drawing the stroke. Likewise, the **InkStrokeDeletingHandler** has the global variable **DeletedNodes** set to an array of **TreeNodes** where each element is a stroke that was just removed by the user deleting strokes.

The rest of this section will discuss writing the JavaScript that consists of the application logic in the SLICE app. This thesis will not discuss the JavaScript language specifics, as it is an internationally standardized programming language. Instead, this thesis will discuss the additional functions that the SLICE framework provides in JavaScript to interact with the SLICE framework. The functionality provided by the SLICE framework is divided up into separate classes; each subsection will discuss one of the classes that make up the SLICE API.

2.3.1. **SLICE API: TreeNode class**

The **TreeNode** class in the SLICE API makes up the majority of the SLICE API and provides functionality to manipulate the “state tree”, which was described earlier. An instance of
a **TreeNode** class represents a single XML node, such as the root `<Slice>` node, a user interface component such as a `<Button>` node, or a node with no special meaning in SLICE at all such as an `<Alice>` node. With an instance of a TreeNode class, the SLICE API provides programmers mechanisms to both read and modify all aspects of the node. This includes the node’s attributes, the node’s parent, and the node’s children. A list of all the non-static methods is listed in Table 4.

The TreeNode class also has several static methods. These primarily deal with search for nodes within the state tree and creating new TreeNodes that can be added to the state tree. Table 5 lists the static methods of the TreeNode class.

There are two constraints that the SLICE framework provides on the TreeNode class that an application developer needs to be aware of. First, a node is unable to have the XML name of its node changed after it has been created. Often, the name of the node is the type of component that it generates. However, the SLICE framework allows the XML to explicitly set its component type by setting the **Type** attribute. The details of the **Type** attribute is discussed in Section 3.3.1.1, but a node cannot change its **Type** attribute after it has been created. Therefore, it is impossible to change a node that is already a Button into an InkPanel; instead, you must create a new node of type InkPanel.
<table>
<thead>
<tr>
<th>Identity-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>String GetName(): Get the XML node’s name</td>
</tr>
<tr>
<td>integer GetNodeNum(): Gets a session-unique identifier for the node</td>
</tr>
<tr>
<td>TreeNode Clone(): Creates a deep clone of the node (no parent, no children)</td>
</tr>
<tr>
<td>String ToXml(): Serializes the node and all children to XML</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>String GetAttribute() or this[]: Gets the value of a specified attribute</td>
</tr>
<tr>
<td>void SetAttribute(): Tells the value of a specified attribute</td>
</tr>
<tr>
<td>boolean HasAttribute(): Returns if an attribute is present as a part of the node</td>
</tr>
<tr>
<td>void RemoveAttribute(): Removes an attribute</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parent/child-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeNode GetParent(): Returns the node’s parent node (or null)</td>
</tr>
<tr>
<td>void Remove(): Removes the node from its parent</td>
</tr>
<tr>
<td>TreeNode[] GetChildren(): Returns an array of children nodes</td>
</tr>
<tr>
<td>TreeNode GetChild(): Returns a specific child</td>
</tr>
<tr>
<td>boolean HasChild(): Returns if a specific child exists</td>
</tr>
<tr>
<td>integer Position(): Returns the node’s zero-based position relative to its siblings as a child</td>
</tr>
<tr>
<td>integer NumChildren(): Returns the number of children nodes</td>
</tr>
<tr>
<td>void AppendChild(): Adds a child node to the end of the list of children nodes</td>
</tr>
<tr>
<td>TreeNode RemoveChild(): Removes a specific child</td>
</tr>
<tr>
<td>TreeNode[] RemoveChildren(): Remove all children; returns children as an array</td>
</tr>
<tr>
<td>void InsertAfter(): Adds a child node after a specific child node</td>
</tr>
<tr>
<td>void InsertBefore(): Adds a child node before a specific child node</td>
</tr>
<tr>
<td>void InsertAt(): Adds a child node in a specific location among the children</td>
</tr>
<tr>
<td>TreeNode NextChild(): Returns the next sibling relative to the parent</td>
</tr>
<tr>
<td>TreeNode PreviousChild(): Returns the previous sibling relative to the parent</td>
</tr>
<tr>
<td>TreeNode GetFrame(): Returns the &lt;Frame&gt; node that the node is contained within</td>
</tr>
<tr>
<td>void AppendChildren(): Adds a list of children to the end of the list of children nodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeNode FindChild(): Searches for direct children nodes with specific features</td>
</tr>
<tr>
<td>TreeNode FindAll(): Recursively searches for any children nodes with specific features</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Print(): Sends an image of the current node (and its children) to the printer</td>
</tr>
<tr>
<td>void PrintChildren(): Sends images of the children nodes (and their children) to the printer</td>
</tr>
<tr>
<td>void Export(): Saves an image of the current node (and its children) to a PDF</td>
</tr>
<tr>
<td>void ExportChildren(): Saves images of the children nodes (and their children) to a PDF</td>
</tr>
<tr>
<td>void Maximize(): Requests for the frame node to be maximized, as if the user clicked the maximize window button. Only works on &lt;Frame&gt; nodes.</td>
</tr>
</tbody>
</table>

Table 4: Non-static methods of the TreeNode class in the SLICE API
2.3.2. SLICE API: Slice class

The Slice class is the second largest set of functionality in the SLICE API and is comprised of functionality that interacts directly with the SLICE framework (as opposed to with a single node, as in the case of the TreeNode class). The functions in this class often result in displaying a user dialog, such as saving the SLICE application to a file on the machine.
Table 6 shows a list of all the functions available to application developers in the Slice class. All the functions are static, as a script is unable to receive an instance of the Slice class.

2.3.3. SLICE API: SliceCloud class

One of the key features of the SLICE framework is the ability to build applications focused on active and collaborative learning. Both of these types of learning often involve real-time data. The SLICE API provides application developers a seamless way to communicate between the various devices that are running SLICE in a classroom. To do this, each instance of a SLICE application needs only to connect to the SliceCloud, a cloud-based service that enables real-time communication between different devices using SLICE. When connecting to the SliceCloud, an application must provide a classroom identifier that identifies which “virtual classroom” the SLICE application should join and a list of attributes. The classroom identifier needs to be the same for each device in the same classroom, but otherwise no format is enforced by the SLICE framework. The list of attributes will describe what types of messages should be received by that instance of the SLICE application.

As an example, suppose there are two devices running SLICE in a classroom. Both devices connect to the SLICE cloud with the same classroom identifier, so they are both in the same “virtual classroom” on the SliceCloud. The first device connects with two attributes: {“Observer” and “Lecturer”}; the second device also connects with two attributes, but one of them differs: {“Observer” and “Student”}. These identifiers may be any string, as determined by

<table>
<thead>
<tr>
<th>Connection-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect() : Connects to the SliceCloud. A classroom name and set of attributes is specified.</td>
</tr>
<tr>
<td>Disconnect() : Disconnects from the SliceCloud.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendTo() : Sends a message to a specific device.</td>
</tr>
<tr>
<td>SendByAttributes() : Sends a message to devices based on attributes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Callback-related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddMessageReceivedCallback() : Specifies a JavaScript function to be called when a message is received from the SliceCloud.</td>
</tr>
<tr>
<td>AddConnectionLostCallback() : Specifies a JavaScript function to be called when a connection is lost to the SliceCloud.</td>
</tr>
</tbody>
</table>

Table 7: Functions in the SliceCloud class of the SLICE API
an application developer. When a SLICE app in that virtual classroom sends a message, it specifies one or more attributes that denotes which devices should receive the message. The SliceCloud will simply broadcast the message to the devices with matching attributes. If a message is sent to devices with the attribute “Observer”, both of the devices will receive the message sent. If a message is sent to “Lecturer”, only the first device will receive the message. Finally, if a message is sent to “Bob”, the message would be delivered to no one. When a message is received, the message contains information about the sender of the message. The receiver may send a message directly back to the sender by using the sender’s unique address. Table 7 shows the set of SliceCloud functions in the SLICE API.

2.3.4. SLICE API: System class

For some applications, information about system-specific features may be necessary to ensure the application looks and feels appropriate for the device that is running the application. The System class in the SLICE API provides data on platform-specific properties. The only functions that this class provides are:

- `GetFrameBorder{Top, Left, Right, Bottom}()`, returns, in pixels, the width or height of the border that is included in the frame component. This represents space that is not usable by SLICE components within a frame.
- `GetScreenWidth()`, `GetScreenHeight()`, returns, in pixels, the screen's width and height. On some systems, the full screen width or height may be larger than the total number of pixels that can be displayed in a frame due to the border of a frame.

2.4. Sample Application

In Sections 2.2 and 2.3, the two main components of an app were outlined: the layout via XML and the application logic via JavaScript. A complete SLICE application will be made up of at least one XML file and at least one JavaScript file. Every app will also need a manifest file that is named `slice.app`. The `slice.app` file is a simple file that specifies the XML file that should be initially loaded when the app is started. Section 3.3.3 details the exact contents of this manifest.
In this section, the details of building a simple “clicker” application will be discussed. This app will have two different views: a view for the student to enter his or her answer and a view for the instructor to see the student responses. For the student, they should see the answer choices A, B, C, D, and E as buttons; pressing a button should send their response to the server. Therefore, we will need to build an XML file to describe the five buttons and ensure that an OnClick event will run the application’s JavaScript to relay the answer to the lecturer.

Working first on the XML, every SLICE app has to have the <Slice> root node, a <Frames> node to contain <Frame> children, and the <Frame> child that will be displayed. Building on this minimal boilerplate of code, our XML will be:

```
<Slice>
  <Frames>
    <Frame W="100" H="300" Visible="True"/>
  </Frames>
</Slice>
```

*Figure 3: Boilerplate XML code for a SLICE application*

If this app is run, the user would need nothing. To make the frame visible and of a reasonable size, a few attributes are added to the <Frame> node. The updated code for a clicker application might look similar to:

```
<Slice>
  <Frames>
    <Frame W="100" H="300" Visible="True"/>
  </Frames>
</Slice>
```

*Figure 4: Continuation of Figure 3; XML attributes were added and highlighted in red*

Running the application now would result in an empty form being shown of a specified size.

The next piece of the layout that is needed is the five buttons. To create the XML to show this, a Panel will be used to encapsulate the five buttons and then the five buttons will be added. After adding the Panel and five Buttons, the XML is:
Finally, the last layout-related change requires an app developer to connect the graphical elements that the user wants displayed in a certain frame to the frame itself. This is done through the \textit{Id} and \textit{DisplayId} fields discussed earlier in Section 2.2. The XML for the SLICE application would now be:

```
<Slice>
    <Frames>
        <Frame W="100" H="300" Visible="True" />
    </Frames>
    <Panel Id="Main">
        <Button X="20" Y="20" W="80" H="20" Text="A" />
        <Button X="20" Y="50" W="80" H="20" Text="B" />
        <Button X="20" Y="80" W="80" H="20" Text="C" />
        <Button X="20" Y="110" W="80" H="20" Text="D" />
        <Button X="20" Y="140" W="80" H="20" Text="E" />
    </Panel>
</Slice>
```

\textbf{Figure 5: Continuation of Figure 4; XML nodes for GUI components highlighted in red}

At this point, the last piece that needs to be added to the XML document is links to the application-specific JavaScript for when certain events occur. Specifically, there are two events that a clicker application is interested in: initialization (to connect to the SliceCloud) and a button press. As with everything else up to this point, these are also added as attributes to the XML. Updating the XML again:

```
<Slice>
    <Frames>
        <Frame W="100" H="300" Visible="True" DisplayId="Main" />
    </Frames>
    <Panel Id="Main">
        <Button X="20" Y="20" W="80" H="20" Text="A" />
        <Button X="20" Y="50" W="80" H="20" Text="B" />
        <Button X="20" Y="80" W="80" H="20" Text="C" />
        <Button X="20" Y="110" W="80" H="20" Text="D" />
        <Button X="20" Y="140" W="80" H="20" Text="E" />
    </Panel>
</Slice>
```

\textbf{Figure 6: Continuation of Figure 5; XML attributes linking a Frame to a Panel highlighted in red}
In this change, the XML makes use of the property that the `OnClick` attribute is an inherited attribute. When any of the children nodes of the `<Panel>` nodes are clicked, the `ButtonClick()` JavaScript function will be called. When the JavaScript is developed later, the `Source` global variable will be used to determine which button was clicked – it is unnecessary to have a series of functions such as “`Button1Clicked()`, `Button2Clicked()`, etc” in order to know which button was clicked.

To finish off the XML completely, the XML must also specify the filename of the JavaScript. This is done via the `JSDefs` attribute. Therefore, the complete XML file for the student side of the clicker is the following:

```xml
<Slice Init="OnInit"
   JSDefs="js.js">
   <Frames>
      <Frame W="100" H="200" Visible="True"
              DisplayId="Main" />
   </Frames>
   <Panel Id="Main" OnClick="ButtonClick">
      <Button X="20" Y="20" W="80" H="20" Text="A" />
      <Button X="20" Y="50" W="80" H="20" Text="B" />
      <Button X="20" Y="80" W="80" H="20" Text="C" />
      <Button X="20" Y="110" W="80" H="20" Text="D" />
      <Button X="20" Y="140" W="80" H="20" Text="E" />
   </Panel>
</Slice>
```

**Figure 7:** Continuation of Figure 6; XML attributes for scriptable actions highlighted in red

The application developer must now write the JavaScript logic for those two functions. In SLICE, nearly all callbacks from the SLICE framework are calls to parameterless functions. The JavaScript code for the student side of the clicker application would be the following:
When the application starts, the app connects to the SliceCloud via a call to `SliceCloud.Connect()`. This call connects the app to the virtual classroom named "Classroom" and identifies the instance by the attribute "Student". When a button is pressed, the app sends a message, with the answer the student selected, to all nodes that contain the attribute "Lecturer". To do this, the `OnClick()` function makes a `TreeNode` that contains the answer and sends that via a call to `SliceCloud.SendByAttributes()`. If the student selects the button A, the actual message sent will be the XML: `<Answer Answer="A" />`.

At this point, the student clicker application is done. To complete the full application, it is necessary to have an application that can view the answers the students selected. For this, a simple histogram will be designed that will add an ‘X’ each time an answer is chosen by a student. To complete this histogram, the following XML is used to place five labels on the user interface to be used as the histogram:

```
<Slice Init="OnInit" JSDefs="js.js">
  <Frames>
    <Frame W="500" H="200" Visible="True"
           DisplayId="Main" />
  </Frames>
  <Panel Id="Main">
    <Label X="20" Y="20" W="460" H="20" Text="A " Id="A" />
    <Label X="20" Y="50" W="460" H="20" Text="B " Id="B" />
    <Label X="20" Y="80" W="460" H="20" Text="C " Id="C" />
    <Label X="20" Y="110" W="460" H="20" Text="D " Id="D" />
    <Label X="20" Y="140" W="460" H="20" Text="E " Id="E" />
  </Panel>
</Slice>
```

One subtle design choice that was made in the XML was that each label is identified with an `Id` that is equal to the answer choice that was sent by the student. In the JavaScript, we will use this property to quickly get the label that corresponds to the student’s answer.
In the case of the lecturer/histogram version of this clicker application, there is only one JavaScript function that is defined in the XML: `OnInit()`. The JavaScript for that function will, just as in the student code, connect to the SliceCloud. However, the lecturer connects with the “Lecturer” attribute instead of the “Student” attribute. Instead of only calling `SliceCloud.Connect()`, the lecturer application is also interested in when a student sends it a message. To accomplish this, the lecturer application will also set up a handler to be called anytime a message is received. The JavaScript for the `OnInit()` function is shown here:

```javascript
function OnInit() {
    SliceCloud.Connect(classroom, "Lecturer");
    SliceCloud.AddMessageRecievedCallback("OnMessage");
}
```

*Figure 11: JavaScript method for the lecturer “clicker” application*

The JavaScript now must also contain a function called `OnMessage()` in order to process the message when it is delivered from the SliceCloud. This function will simply add an “X” to the label that corresponds to the answer chosen by the student.

```javascript
function OnMessage(message, senderId) {
    var answerBox = TreeNode.FindNodeById(message["Answer"]);
    answerBox["Text"] = answerBox["Text"] + "X";
}
```

*Figure 12: JavaScript method for the lecturer “clicker” application*

At this point, a complete sample application has been developed. To complete the application, a few convenience features were also added such as the ability for the student to see what answer choice he or she made and the ability for the lecturer to reset the histogram. The complete application, both student and lecturer, can be seen in Figure 13.

2.5. Conclusions

In this chapter, the SLICE framework was explained from the prospective of an application developer. As part of developing an application, the application developer provides an initial XML “state tree” that is loaded into the SLICE framework and then manipulates that tree via application-specific logic programmed in JavaScript. Both the XML document and the SLICE API that is used as part of the JavaScript scripts were detailed in this chapter. Additionally, an example clicker application was developed in this chapter.
Some work was done to explore the usability of the API. Much of the published work on APIs examines large APIs, such as the full Java API or other APIs that contain hundreds to thousands of classes [34]. By contrast, the SLICE API contains only five classes and less than 100 methods. With a relatively small API, a developer could easily read through the entire API and issues that are raised when dealing with large APIs, such as the ability to find specific functionality, is of significantly less concern.

Anecdotally, we have found the SLICE API has been easy to use by students. To date, an incoming freshman with only limited programming experience was able to successfully develop a clicker-like application as a summer project and a group of two undergraduate students were able to extend the Lecturer Application with a new interface to ask questions to the instructor of the course. Both of these projects were completed successfully by the students with relative ease.

The next chapter details the internals of the SLICE framework. It will examine how much of the functionality explained in this chapter is achieved and will show how the same application is capable of running natively on both Android and Windows, as well as on Mac and Linux via Java.
3.1. Introduction

In the previous chapter, the SLICE framework was described from the point of view of the application developer. This chapter will detail the SLICE framework internals, exploring how all the functionality provided in the previous chapter is possible. There are two orthogonal ways to describe the SLICE framework. The first is to view the framework from a very high level that follows a well-known software pattern known as the Model-View-Controller (MVC) pattern [35]. A description of the full framework in terms of the MVC pattern is presented in Section 3.2. The second is examine it at a functional level Section 3.3 provides this in-depth, source-code based description of the SLICE framework by breaking it up into three components: core (Section 3.3.1), frontends (Section 0), and apps (Section 3.3.3). This chapter will first begin with the high-level view using the MVC design pattern.

3.2. High Level Overview of the SLICE Framework

To give a high-level overview of the SLICE framework, we will begin by looking at the entire framework as three distinct, logical pieces. The framework was designed using the Model-View-Controller (MVC) software design pattern, a common software pattern that is used in various existing software frameworks. The three logical components of the MVC design pattern are the “model”, “view”, and “controller”. Between each of these components, there are cleanly defined interactions. The three components and their interactions are illustrated in Figure 14 and explained in detail in this rest of this section.

3.2.1. The MVC “model” component

The first of the three logical components is the “model”. The model’s function is composed to two tasks: the model maintains a structured representation of the entire state of the SLICE application (consisting of the XML “state tree” described in Section 2.2) and the model
provides a suite of functions for SLICE applications to interact or modify the state of the SLICE framework.

The first responsibility of the model, the representation of the state of the SLICE application, is structured as a tree. This tree structure, referred to as the “state tree”, has a single root node with the name Slice. As a tree structure, this data can be serialized to XML. Throughout this document, we will refer to the state tree in terms of an XML document even though the internal representation in SLICE is not XML.

As stated earlier, the majority of the “state tree” is made up of nodes that describe the graphical user interface (GUI). These nodes are common GUI components that are used in programming, including <Button>, <Label>, and <Panel>. For example, the <Button> node will display a button on the screen. Beyond just displaying the button, the <Button> element may contain information saying what to do if the button is pressed by the user. The specifics of these GUI components, and full details on what goes into a state tree, can be found in Chapter 2. However, at a high level, what is important is that the state tree described exactly what could be displayed on the screen (based on the visibility property of the node) and what to do when the user interacts with a component.
The second subcomponent that makes up the model component of the SLICE framework is the suite of functions that the model provides to an application developer. In total, the full suite of functions is referred to as the “SLICE API”, or Application Programming Interface. The majority of these functions deal with modifying the state tree – operations such as “add a child node”, “remove a child node”, or “make this node (and its descendants) invisible”. However, the SLICE API also provides the ability to interact with several services the SLICE framework provides. These services include the SliceCloud, the networking model for communicating with other instances of SLICE, event handlers for when certain system-related events take place, and other functionality. In fact, all interactions with the SLICE framework by an application take place through the SLICE API. At a high level, it is important to understand that the SLICE API provides all the interactions between a SLICE application and the SLICE framework. This chapter will not go into the details of each of the functions that are provided, but the full suite of functions that make up the SLICE API is discussed in Chapter 2.

3.2.2. The MVC “view” component

The second logical component in the SLICE framework is the “view”. At a high level, this is the simplest of the three components. The view is responsible for displaying the content of the “state tree” to the user of the application and forwarding any interactions that the user makes with the view back to the SLICE framework. Since SLICE runs on a variety of platforms including Windows, Mac, Linux, and Android, there are three separate implementations of the “view” component of the SLICE framework. These different implementations are discussed in Section 0.

3.2.3. The MVC “controller” component

The final logical component of the SLICE framework is the “controller”, which connects the “model” and “view” together into the complete SLICE framework. The controller is responsible for running one or more script files (written in JavaScript) that contain the application logic and manipulates the SLICE “state tree” via calls to the SLICE API. Unlike the previous two components, these scripts are entirely written by an application developer and not part of the “SLICE core” (discussed in Section 3.3.1). The SLICE core only facilitates the
running of these scripts. Specifically, script functions are activated either by user events (such as the click of a button) or by state changes the SLICE framework (such as initialization or a network message). The very first state change, the startup of the SLICE application, will result in the SLICE framework calling the function specified for the Initialize event. A detailed list of all the events can be found in Section 2.2.

3.2.4. Interactions between the MVC components

With all three high-level components of the SLICE framework discussed, the interactions between the layers should be clear.

- The model provides a structured “state tree” (XML) that represents exactly what should be displayed to the user.
- The view renders and displays the state tree to the user and captures user events. The state tree describes any applications logic that should run when an event, such as a button click, occurs.
- The controller runs the application logic of a SLICE application. This application logic, written in JavaScript, makes calls to the SLICE API. The SLICE API will change the state tree, which will update the view, which will allow the user to take another action, which will call more application logic.

Figure 14 summarizes these interactions in a visual form, showing the cycle of a user interaction resulting in a call to the SLICE API, which changes the state tree, which updates the view.

This section provided a high-level view of the three major logical components in the SLICE framework. Now we will look in-depth at how these three high-level components interact at a technical level, including examining the modularization of the SLICE source code.

3.3. Technical Overview of the SLICE Framework

The SLICE framework is made up of three main technical components:

- The SLICE core, the core technologies that make up all non-platform specific features of the SLICE framework.
• SLICE frontends, the light-weight platform-specific implementations of the platform-specific features of the SLICE framework.
• SLICE applications, the files (scripts, state trees, and related files) that are designed by an application developer when writing for the SLICE framework.

These components do not strictly align with the MVC representation of the SLICE framework that was discussed in the previous sections. However, each technical component largely makes up one MVC component but each component often has interactions that span between two or all three MVC components.

We dig in with the largest component, the SLICE core, by dividing it into several subcomponents. Following that, a description of the frontend system will be provided and information on the three implemented frontends will be discussed (.NET for Windows, Java/Swing for Mac/Linux and Android for Android-based devices). Finally, a brief description of how the SLICE applications fit into the picture will be provided.

3.3.1. The SLICE Core

The largest and most complex single piece of the SLICE framework is known as the “SLICE core”. The core is responsible for operations that are not application-specific and not platform-specific. The core also provides interfaces for platform-specific and application-specific features to use the SLICE framework. In that way, the core is both the glue that holds the SLICE framework together and the center of the SLICE framework.

The whole of the core is programmed in Java using only platform-independent libraries. In the source code that makes up the SLICE core, the core is divided up into several subcomponents. The subcomponents are:

• “tree” (edu.illinois.slice.tree), responsible for maintaining the structured representation of the state of a SLICE application. Further, the tree subcomponent is also responsible for communicating changes to the state tree to the platform-specific frontend to ensure the view is always exactly what is described by the “state tree”.

33
• “scripting” (edu.illinois.slice.scripting), responsible for running SLICE application scripts that contain application-specific logic.

• “handlers” (edu.illinois.slice.handlers), responsible for handling events that occur within the SLICE framework. These events include user interaction events (eg: the user clicks a button), networking event (eg: a message is received from another device running your application), or system events (eg: the SLICE framework has finished being initialized). The results of these events are often a call to a SLICE application’s scripts.

• “networking” (edu.illinois.slice.networking), responsible for sending and receiving messages from the SliceCloud. The SliceCloud is the cloud-based networking model that allows for instances of a SLICE app on different devices to communicate with one-another seamlessly.

• “api” (edu.illinois.slice.api), responsible for providing all of the interfaces to platform-specific features that are used by the SLICE framework. These platform-specific features must be implemented in a SLICE frontend.

• “logging” (edu.illinois.slicelogging), responsible for logging events, primarily used for debugging purposes.

Additionally, there are two utility subcomponents that provide commonly used features to all of the other components: “assets” and “util”. Figure 15 shows all of the subcomponents within the SLICE core and their relations with one another. The remaining subsections in this section will discuss each of the subcomponents.

3.3.1.1. “Tree” subcomponent

The first subcomponent of the SLICE core that will be discussed is the “tree” subcomponent. Identified in the source code as the package edu.illinois.slice.tree, the “tree” subcomponent maintains the structured representation of the state of a SLICE application, known as the “state tree”, and provides an interface to update the display as changes are made to the “state tree”. As introduced earlier in Section 3.2.1, the state tree is a data structure in the form of a tree that can be serialized to XML at any time. Due to the wide-spread use of XML, the state tree is often viewed as a “living” XML document that is able to be manipulated by SLICE applications.
Every tree (and every XML document) is made up of nodes. The only general requirement for a tree is that there is exactly one root node. In SLICE, this root node is the `<Slice>` node. Beyond that, every node may contain any number (including zero) of sub nodes or child nodes and may contain any number (including zero) of attributes. The attributes of each node are key-value pairs where both the key and the value are strings. To enable serialization to XML, the name of the attribute key must not contain any spaces.

Figure 15: The SLICE framework with the SLICE core split into subcomponents and interactions between subcomponents labeled. A brief description of each interaction is detailed below.

(a): The initial state of the “state tree” is provided by the SLICE application to the `tree` subcomponent
(b): The script files of a SLICE application interact with the core via the `scripting` subcomponent
(c): Scripts may call for platform-specific functionality, requiring use of the `api` subcomponent
(d): Scripts may modify the “state tree”, requiring modification of the state tree in the `tree` subcomponent
(e): When a handler is triggered by an event, handling the event result in scripts being run
(f): Network messages create an event in the SLICE core that is handled by the `handlers` subcomponent
(g): The `networking` subcomponent manages all aspect of the connection to the SliceCloud
(h): The `api` subcomponent manages any necessary platform-specific resources
(i): Changes to the state tree that modify the user is relayed by the `tree` subcomponent to the `frontend`
(j): Changes to the state tree may result in event handlers being created in the `handlers` subcomponent
(k): User interactions, such as button clicks, trigger events that are handled by the `handlers` subcomponent
As in XML, every node has a node name. Often, this node name represents the function of the node in the state tree. For example, a node with the name Button will result in a button being displayed on the screen. However, this is not a hard requirement. Instead, the actual type of the node follows a simple set of rules:

- If the node contains an attribute with the key Type, the value of that attribute is used as the node’s type.
- If the node does not contain an attribute with the key Type, the name of the node is used as the node’s type.

Figure 16 shows XML representations of two nodes, one using the node’s name as its type and one using the Type attribute as the node’s type. If a node is used for only organization purposes, it can have a type that has no meaning to the SLICE framework. In Section 2.2, the node types available in the SLICE framework are explained in detail.

Different subclasses are used internally to represent the different node types. The class TreeNode is used to represent all nodes that do not have a specific pre-defined type (for example, a node with the type AliceInWonderland does not have any SLICE-specific functionality) and is also used as the base class for all node types that do have some specific function within the SLICE framework. Figure 17 shows the class hierarchy for all of the classes used in the state tree.

Out of all the classes in Figure 17, the ComponentTreeNode class is of most significance. This class provides a wealth of functionality to bridge the SLICE core, which manages a data representation of the state of the SLICE application, with the graphical user interface that is displayed to the user of an app. This includes three key concepts: activation, attribute actions, and the component interface.

![Figure 16: Two separate representation of nodes, both having the type of “Button”](image-url)
Activation exists entirely for optimization reasons to ensure a fast and robust user experience. A node is defined as activated if its graphical component is currently visible on screen. Visibility is a somewhat complicated attribute that follows these rules:

- If the node is contained within a “container node” (a <Frame> or <Panel>, the concept of a container node is discussed in detail in 2.2) that has a visible attribute of False, the node is not visible because it is contained in an invisible container.
- If the node is not contained within any invisible containers, the visible state of its graphical component depends on the visible attribute of the node itself. If the node itself does not have a visible attribute, then the visibility of the graphical component depends on the parent’s visible attribute. If the parent also does not contain a visible attribute, the node would attempt to find a visible attribute by recursively looking at parent nodes (eg: moving up in the state tree).

As the process of finding the visibility state of a node’s graphical component is somewhat complicated, each ComponentTreeNode stores the activated state of its node. If a node’s graphical component is not visible, and therefore the node is not activated, then it is not important for the node to provide updates to the view about changes to the node.

When a node is activated, any change to the state tree has to be reflected in the graphical user interface. To do this in a general way, the ComponentTreeNode provides the concept of an attribute action. An attribute action is simply an action that needs to take place when an attribute is changed. Each subclass of ComponentTreeNode has its own set of attribute actions and the

![Figure 17: The class hierarchy of classes representing nodes in the SLICE state tree.](image)
ComponentTreeNode class itself provides a base set of attribute actions that are applicable to all nodes that have a graphical component. As an example, one attribute that has meaning to graphical components is the attribute $x$, an attribute that describes the horizontal position in pixels within the node’s container. The attribute action associated with $x$ is called every time the value of $x$ changes. The attribute action of $x$ would notify the graphical component to change its position on the screen. It is worth noting that not all attribute actions relate to changing something on the screen: the OnClick attribute action sets up a “handler” to handle the action of a button being clicked by the user. The full set of attributes and a description of what each attribute does can be found in Section 2.2.

If a node is not activated, no graphical component corresponds to the node and no attribute actions are necessary. As part of activating the node, the entire list of attributes applied to the node is examined and all of the attribute actions associated with the attributes that are present are executed. Since the Visible attribute controls the visibility of the node, the Visible attribute action will always be the last attribute action to run in order to ensure that the component has received all the state information before being shown to the user.

The third concept provided by the ComponentTreeNode class is the interface with the actual graphical component. The graphical components are platform-specific features and not part of the SLICE core, but the interaction with them starts within the ComponentTreeNode. These functions often closely relate to the attributes that have special meaning within SLICE. For example, the $x$ attribute action calls the method setSizeAndLocation() in order to update the location of the graphical component. Table 8 shows the set of methods that are provided in the Component class. These methods are called by attribute actions in the ComponentTreeNode class and these methods must be implemented by any component that can be used within SLICE framework. The implementation of these methods in platform-specific code is considered part of the SLICE frontend, discussed later in this chapter in Section 0.

The five remaining subclasses of TreeNode each provide specific functionality:

- **RootTreeNode**, provides attribute actions for the JSDefs and Init attributes that can only be applied to the root node. Since the root tree node is not a graphical component, this class extends only the TreeNode class and not the ComponentTreeNode class.
FrameTreeNode, provides attribute actions for the DisplayId and FrameResizeHandler attributes that can only be applied to <Frame> nodes.

InkComponentTreeNode, provides attribute actions for the PenColor, PenWidth, and Transparency attributes that can only be applied to ink-able components. Additionally, it is the parent class of the InkPanelTreeNode and StrokeTreeNodeClass.

InkPanelTreeNode, provides the attribute action for the EditMode attribute that can only be applied to ink panel components.

StrokeTreeNode, provides specialized serialization and deserialization of strokes. Unlike all other nodes, a <Stroke> node must be a leaf in the state tree as the data that makes up the stroke is not stored as an attribute but is stored as children nodes within the <Stroke> node. The details of the <Stroke> node is of greater concern to application developers and is discussed in the context of building a SLICE application in Chapter 2.

Based on the minimal added functionality by each of these classes, it can be said that the bulk of the “tree” subcomponent of the SLICE core is comprised of the TreeNode and ComponentTreeNode classes.

3.3.1.2. “Scripting” subcomponent

Identified in the source code as the package edu.illinois.slice.scripting, the “scripting” subcomponent is responsible for executing the scripts that make up a SLICE application and for providing an interface for the scripts to interact with the SLICE framework.

The first responsibility of the “scripting” subcomponent is the execution of scripts. As mentioned earlier, all scripts in SLICE apps are written in JavaScript. Given the widespread use of JavaScript, this language is known to a large number of developers and several engines exist to run JavaScript code. In the SLICE framework, JavaScript code is executed using the Rhino JavaScript engine developed by the Mozilla Foundation [36]. The “scripting” subcomponent provides a ScriptingEngine class file that bridges the interactions between the Rhino engine and the rest of the SLICE core. The ScriptingEngine class provides only a few functions to the rest of the SLICE core:
<table>
<thead>
<tr>
<th>General Component Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setSizeAndLocation(float x, float y, float width, float height)</code></td>
</tr>
<tr>
<td>Sets the horizontal and vertical location of a component with respect to the top-left corner of its container and sets the width and height of the component. All units are pixels.</td>
</tr>
<tr>
<td><code>setEnabled(boolean enabled)</code></td>
</tr>
<tr>
<td>Sets the enabled state of the component. For example, a disabled button is not clickable.</td>
</tr>
<tr>
<td><code>setVisibility(boolean visible)</code></td>
</tr>
<tr>
<td>Sets the visible state of the component.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Textual Component Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setText(String text)</code></td>
</tr>
<tr>
<td>Sets the text of a component.</td>
</tr>
<tr>
<td><code>setTextAlign(TextAlign textAlign)</code></td>
</tr>
<tr>
<td>Sets the alignment of the text. The <code>TextAlign</code> class provides both vertical (<code>TOP</code>, <code>MIDDLE</code>, or <code>BOTTOM</code>) and horizontal (<code>LEFT</code>, <code>CENTER</code>, <code>RIGHT</code>, or <code>JUSTIFIED</code>) alignment.</td>
</tr>
<tr>
<td><code>setTextSize(float pixels)</code></td>
</tr>
<tr>
<td>Sets the size of the text in pixels.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Color and Background Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setForegroundColor(Color color)</code></td>
</tr>
<tr>
<td>Sets the foreground (text) color.</td>
</tr>
<tr>
<td><code>setBackgroundColor(Color color)</code></td>
</tr>
<tr>
<td>Sets the background color.</td>
</tr>
<tr>
<td><code>setBackgroundImageFromPNGImage(byte[] pngImage)</code></td>
</tr>
<tr>
<td>Sets the background to an image, specified as a PNG image file.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scale Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setScaleFactorX(int factorX)</code></td>
</tr>
<tr>
<td>Sets the factor in which to scale the component in the X direction. By default, every component is not scaled and would have a scale factor of 1. A scale factor of 2 would result in the component being displayed as twice as large as the width specified.</td>
</tr>
<tr>
<td><code>setScaleFactorY(int factorY)</code></td>
</tr>
<tr>
<td>Sets the factor in which to scale the component in the Y direction. By default, every component is not scaled and would have a scale factor of 1. A scale factor of 2 would result in the component being displayed as twice as large as the height specified.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lifecycle Control</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addToContainer(int index)</code></td>
</tr>
<tr>
<td>Notifies the frontend that the component should be added to its container. This is usually the last call made to a component when its node has become activated in the state tree. The index specifies the z-index that it should be added at to ensure that components are displayed in the correct z-order.</td>
</tr>
<tr>
<td><code>removeFromContainer()</code></td>
</tr>
<tr>
<td>Notifies the frontend that the component should be removed from its container and all memory associated with the component should be freed. After this call, the SLICE core no longer maintains a reference to this component.</td>
</tr>
</tbody>
</table>

| **Table 8**: The suite of methods in the Component class that are used by the ComponentTreeNode class to interact with platform-specific SLICE frontend components |  |
• `activate()`, used to initialize the JavaScript engine.
• `call()`, used to call a script function. This is primarily used by a handlers to call an a script based on a user event (such as a user clicking a button).
• `addScriptBySource()`, used to add JavaScript source code to the JavaScript engine.
• `setGlobal()`, used to set a global variable within the JavaScript context.

The second responsibility of the “scripting” subcomponent is to allow for the scripts to interact with the SLICE framework. To accomplish this, the SLICE framework provides a series of “bridge” functions that can be called from JavaScript but run in the SLICE core written in Java. The full set of functions available to be called from JavaScript make up the Application Programming Interface (API) for application developers. The full JavaScript API is found in Section 2.3.

To allow for the SLICE core to contain extremely optimized code, the SLICE API does not call the methods of the `TreeNode` class inside the core. Within the `TreeNode` class, it is assumed that the parameters being passed into each method are valid. However, code written by SLICE application developers may not always provide correct parameters to their calls. Instead, the functions that make up the SLICE API are wrappers around the SLICE core’s structures. For example, the SLICE API includes a `TreeNode.InsertAt(TreeNode treeNode, int index)` function that allows for a script writer to insert a subtree into another subtree at a specific index. The SLICE API will validate that the index is valid before calling the `TreeNode.addChildAtIndex()` call that is part of the SLICE core.

This section focused on the responsibility of the scripting subcomponent of the SLICE core. While the responsibility of this subcomponent can be easily described, there are nearly 100 different functions that make up the SLICE API. The bridged classes that sit between the JavaScript and the rest of the SLICE core make up a non-trivial amount of the SLICE core and are covered later as part of the SLICE API in Chapter 2.
3.3.1.3. “Handlers” subcomponent

Identified in the source code as the package `edu.illinois.slice.handlers`, the “handlers” subcomponent is responsible for executing scripts based on events that take place within the framework. In the SLICE framework, events come from three sources:

- **User Events:** events that take place because the user interacted with the SLICE framework’s graphical interface, such as a user clicking on a button.
- **Network Events:** events that take place due to network message. These events include receiving a message from another device using the SLICE framework or receiving a notification that a new device has joined in as part of your “virtual classroom”.
- **System Events:** event that take place due to internal SLICE framework activity. There are exactly two of these events: on start-up and on exit.

While the source of the event is interesting and logged for debugging purposes, the “handlers” source code does not make a distinction based on the type of the event. Instead, events are categorized based on the information that is processed as part of the event. Inside the source code, all events are processed by one of the following three handler classes:

- **TreeNodeHandler**, handles all events that relate to a `TreeNode`. For example, a button being clicked has a `Source` property associated with the button click; the `Source` is a `TreeNode` and a `TreeNodeHandler` is used to process this event.
- **MouseEvent**, handles all events related to a mouse or hover event. The `MouseEnter` and `MouseExit` events, used primarily in the context of changing the look and feel of the user interface when a user hovers over a spot on the interface, trigger this event class.
- **MessageEvent**, handles only the event of a network message being received from the SliceCloud.

Table 9 shows the full list of all events handled by the classes described in the list above.

3.3.1.4. “Networking” subcomponent

Identified in the source code as the package `edu.illinois.slice.networking`, the networking subcomponent maintains communications with the SliceCloud while the SLICE
The SliceCloud is the cloud-based networking model that is used for SLICE apps to communicate with other SLICE apps within a “virtual classroom”. The functionality provided by the SliceCloud consists of only a very primitive Publish-Subscribe message pattern where SLICE applications join a virtual classroom and subscribe to one or more channels in that classroom. Both the virtual classroom and the channels are identified by strings; these can be arbitrary, although we follow a convention such that a typical virtual classroom name is “1404sc:cs241:wade”.

For example, in the flagship SLICE application known as the “Lecturer App” (discussed in detail in Chapter 5) an instance of the Lecturer App is either acting as a student, a lecturer, or a display. If these instances were currently running in a CS 241 lecture, a student may connect to the SliceCloud and join the virtual classroom “cs241” and subscribe to the “student” channel. Any message sent by another device running SLICE that was in the same classroom that directed the message to the “student” channel would then be received by this student device.

At a technical level in the SliceCore, the interactions with the SliceCloud are very simple. The use of the SliceCloud in developing a SLICE application is discussed in detail in Section 2.3.3.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>System</td>
<td>The SLICE app is first initialized.</td>
</tr>
<tr>
<td>Exiting</td>
<td>System</td>
<td>The SLICE app is exiting.</td>
</tr>
<tr>
<td>MouseClick</td>
<td>User</td>
<td>The user clicked or pressed a graphical component.</td>
</tr>
<tr>
<td>TextChanged</td>
<td>User</td>
<td>The user changed the text of a graphical component.</td>
</tr>
<tr>
<td>KeyDown</td>
<td>User</td>
<td>The user pressed down a key on their keyboard.</td>
</tr>
<tr>
<td>KeyUp</td>
<td>User</td>
<td>The user released the key on their keyboard.</td>
</tr>
<tr>
<td>MouseDoubleClick</td>
<td>User</td>
<td>The user double clicked or double tapped a graphical component.</td>
</tr>
<tr>
<td>MouseEnter</td>
<td>User</td>
<td>The user’s mouse or stylus entered into the space of a graphical component.</td>
</tr>
<tr>
<td>MouseLeave</td>
<td>User</td>
<td>The user’s mouse or stylus left the space of a graphical component.</td>
</tr>
<tr>
<td>InkStrokeAdded</td>
<td>User</td>
<td>The user drew an ink stroke on an InkPanel.</td>
</tr>
<tr>
<td>InkStrokeDeleted</td>
<td>User</td>
<td>The user erased an ink stroke from an InkPanel.</td>
</tr>
<tr>
<td>NetworkMessageReceived</td>
<td>Network</td>
<td>A message was received from the SliceCloud.</td>
</tr>
<tr>
<td>FrameResized</td>
<td>User</td>
<td>The user resized the SLICE app’s frame.</td>
</tr>
</tbody>
</table>

Table 9: Full list of events handled by the SLICE core handlers
3.3.1.5.“API” subcomponent

Identified in the source code as the package `edu.illinois.slice.api`, the “api” subcomponent manages the set of features provided by the SLICE framework that cannot be written in platform-independent code and are not graphical components. The exhaustive list of these features is:

- **Component Factory:** This API follows the Factory Method pattern of software engineering that provides a single interface to create components for all possible graphical components. The `ComponentFactory` interface contains a single method: `createComponent()`, which takes the component type and the component’s graphical “container” as arguments.

- **Resource Resolver:** This API provides the SLICE core with access to device-specific methods of loading and saving files locally. The Windows frontend would implement saving a file via a Windows-standard save file dialog box; on Android, no such dialog exists and the Android frontend provides users with various options to load or save a file.

- **System Information:** This API provides the SLICE core with information about the system that SLICE is currently running on, such as the screen size in pixels, the size of any device-specific borders (for example, the frame around every window in Windows), and if the device supports the ability to read or save PDF files.

- **System Resources:** This final API focuses on features that are not a specific SLICE component but usually provide the user with information. This includes showing a message box on top of the application to the user, showing a progress dialog to the user while a long action is taking place in the background, or generating a screenshot of the rendered SLICE state tree for the SLICE core to use.

Each SLICE frontend is required to implement all of these interfaces in order to work with the SLICE core. As such, there exists an implementation of all of these features in all three SLICE frontends built to date: .NET, Java/Swing, and Android.
3.3.1.6. Utility subcomponents

In Figure 15 (page 35), there are three subcomponents that are ubiquitous in use across the SLICE core but do not specifically perform a role within the SLICE core on their own. These are “logging”, “util”, and “assets”.

The “logging” subcomponent allows the SLICE core to log messages as events happen. Each logged message is tagged with a specific “level” based on the message being logged and each level has a numerical value to denote its importance. The tag with the greatest importance, “Severe”, is used in cases where there is an unrecoverable event in the SLICE framework resulting in the framework crashing. Table 10: Categories of messages logged within the SLICE framework gives the tag categories used within the SLICE framework.

The “util” subcomponent provides utility methods. It is made up of base classes:

- **Base64**, a library for encoding and decoding data to and from Base64. The Base64 encoding standard is used by the SLICE framework to store binary data such as PNG images in an XML-serializable form. The library included in the SLICE core uses the iHarder’s Base64 library [37], which is released under public domain.
- **StrokeHelper**, a simple utility class allowing for the generation of an ink stroke node for the state tree. A user of this class provides a series of (X, Y) points (and optionally time and pressure information) and the class provides metadata about the stroke such as the number of points, the bounding box that fits the stroke, and the TreeNode node for the stroke.
- **StrokePoint**, the base object that makes up all stroke points. This class has two subclasses, StrokePoint_PointOnly and StrokePoint_FullData. The two different subclasses allow for a Java object to be created based on the stroke information that was provided: if only (x, y) points are provided, then memory isn’t needed to store auxiliary data such a time and pressure.

The “assets” subcomponent provides classes similar to the “util”, but generally only store information and don’t perform any functionality. These classes include **Color**, for storing
3.3.2. SLICE Frontends

Throughout the previous section, several mentions have been made of “platform-specific” code that exists outside of the SLICE core. As it relates to the full SLICE framework, the core consists of all of the platform-independent code that is shared by all SLICE frontends and all SLICE apps. Each frontend consists of all the platform-dependent code that is used by apps when running on a specific platform. This modular nature allows for SLICE to be ported to different platforms with ease.

As outlined in the previous sections, a SLICE frontend must implement the complete set of interfaces to the SLICE core. This consists of two broad groups: graphical components (SLICE core’s “tree” subcomponent) and platform-specific features (SLICE core’s “api” subcomponent). The goal across both groups of functionality for each frontend is to provide a

<table>
<thead>
<tr>
<th>Message Category</th>
<th>Level</th>
<th>Description of the event generating the message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severe</td>
<td>900</td>
<td>An unrecoverable failure in the SLICE framework.</td>
</tr>
<tr>
<td>Warning</td>
<td>800</td>
<td>Unexpected behavior within the SLICE framework, but not resulting in an unrecoverable state.</td>
</tr>
<tr>
<td>Info</td>
<td>700</td>
<td>High-level SLICE framework state information, such as a SLICE application starting.</td>
</tr>
<tr>
<td>Config</td>
<td>600</td>
<td>Technical details about the configuration changes in the SLICE framework.</td>
</tr>
<tr>
<td>User Action</td>
<td>570</td>
<td>A frontend notifies the SLICE core of a user interaction.</td>
</tr>
<tr>
<td>Network Message</td>
<td>560</td>
<td>A network message is received from the SliceCloud.</td>
</tr>
<tr>
<td>Handler Invoked</td>
<td>550</td>
<td>A handler is invoked (see “handler” subcomponent).</td>
</tr>
<tr>
<td>Fine</td>
<td>500</td>
<td>Uncategorized. Used for moderate importance.</td>
</tr>
<tr>
<td>Tree Created</td>
<td>480</td>
<td>A new subtree is created (see “tree” subcomponent).</td>
</tr>
<tr>
<td>Tree Element Changed</td>
<td>470</td>
<td>A node has gained or lost a child node.</td>
</tr>
<tr>
<td>Tree Activation Changed</td>
<td>460</td>
<td>A node has changed its activation state.</td>
</tr>
<tr>
<td>Tree Attribute Action</td>
<td>455</td>
<td>An attribute action was executed (see Section 2.3).</td>
</tr>
<tr>
<td>Tree Attribute Changed</td>
<td>450</td>
<td>An attribute of a node was changed.</td>
</tr>
<tr>
<td>Finer</td>
<td>400</td>
<td>Uncategorized. Used for limited importance.</td>
</tr>
<tr>
<td>Finest</td>
<td>300</td>
<td>Uncategorized. Used for low importance.</td>
</tr>
<tr>
<td>Tree Attribute Read</td>
<td>250</td>
<td>An attribute of a node was read.</td>
</tr>
</tbody>
</table>

Table 10: Categories of messages logged within the SLICE framework.

information related to what color to display, and TextAlign, for the various options for aligning text in a component.
There are currently three SLICE frontends: a frontend using Microsoft’s .NET platform for Windows-based devices, frontend using Java’s Swing graphic library for Apple and Linux-based devices, and a SLICE frontend for Android devices using the Android SDK. Each version of the SLICE framework runs entirely natively on the architecture that it targets – that is, the version of SLICE for Windows runs entirely within the .NET framework; the version of SLICE for Android runs entirely native dalvik class files. Future work for the SLICE framework could be to provide a frontend for a greater number of devices, such as the Apple iPad or iPhone.

The most widely deployed and used frontend of the SLICE framework is the Windows frontend. Like all of the frontends, the Windows frontend uses the same SLICE core that was described in Section 3.3.1. However, unlike the other frontends that are natively Java, the Windows frontend uses a translated version of the SLICE core using the IKVM.NET “compiler” (ikvmc) [38]. Effectively, IKVM.NET translates a compiled Java package (a jar file, written in

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Windows (.NET)</th>
<th>Java (Swing)</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>Window and Canvas</td>
<td>JFrame</td>
<td>Activity</td>
</tr>
<tr>
<td>Panel</td>
<td>Canvas</td>
<td>JPanel</td>
<td>AbsoluteLayout</td>
</tr>
<tr>
<td>Button</td>
<td>Label</td>
<td>JLabel</td>
<td>Button</td>
</tr>
<tr>
<td>TextView</td>
<td>Label</td>
<td>JLabel</td>
<td>TextView</td>
</tr>
<tr>
<td>ImageView</td>
<td>Label</td>
<td>JLabel</td>
<td>TextView</td>
</tr>
<tr>
<td>InkPanel</td>
<td>InkCanvas</td>
<td>JPanel</td>
<td>AbsoluteLayout</td>
</tr>
</tbody>
</table>

Table 11: Native platform-specific components used in each SLICE frontend

![Figure 18: Two student instances of the SLICE Lecturer Application on heterogeneous platforms. Left: Android tablet; Right: Windows 7 Tablet PC](image)

standard interface that is functionally identical across the different frontends while providing a device-specific look-and-feel.

![image]
Java bytecode) into a native .NET binary (a dll file, written with Microsoft Common Intermediate Language or CIL). Since this is translated and not interpreted, the result of the translation from Java bytecode to CIL is that the SLICE core is running as native .NET CIL and is directly accessible from any other .NET application and there is no dependency on Java.

With the SLICE core running as native .NET code, the only piece required to finish a SLICE frontend is the implementation of the interfaces required by the SLICE core. The choice was made to write the .NET frontend using the latest Windows model provided by Microsoft, known as WPF or Windows Presentation Foundation. By using WPF (instead of the older Windows Forms model), the SLICE framework on Windows gains the advantages of that WPF provides but restricts the SLICE frontend to Windows machines running .NET 4.0 or higher. This means that the Windows frontend will run on any computer running Windows XP or a later version of Windows (including Windows 7 or Windows 8).

The differences between the three SLICE frontends are minimal. Each frontend provides a native way to load files, save files, native message boxes to give users information, and other related functionality. Table 11 shows the native, platform-specific graphical components used as the base class for each component type by each of the frontends. Figure 18 displays a picture of two devices running the same application with different frontends.

3.3.3. Launching a SLICE application

In Sections 3.3.1 and 0, technical descriptions of the inner-workings of the SLICE framework were described by detailing the SLICE core and the SLICE frontends. The final piece required to utilize the SLICE framework is a SLICE app. Given the large number of interactions between an app and the SLICE core, the process of writing a SLICE app is covered in detail in its own chapter. In this section, the structure of an application will be discussed as it relates to the SLICE framework as a whole. The specifics of the how to write that app was discussed in Chapter 2.

A SLICE application is made up of user created files that make up the SLICE app. The names and content of those files are entirely set by the user. There is only one file that is required and expected by the SLICE framework: slice.app. The slice.app file must be at the
root level of the application’s directory structure and contains information about the current version that the app and the initial “state tree” of the SLICE framework when the app launches. The slice.app file used for the Lecturer App that is discussed in Chapter 4 is shown in Figure 19.

When the application is launched by using a SLICE frontend, the state tree is loaded from the XML file that was specified in the slice.app file. Once the state tree has been loaded into the SLICE core, SLICE raises the “initialized” event in order to invoke any user-defined scripts that need to run upon application startup. After the initialization has finished, the application is running and the mechanisms described in this chapter are all active and running.

On the Windows platform, a tool was developed to help the user launch SLICE apps. This tool, called the SLICE Launcher, allows the user to choose the SLICE application that he would like to launch. Additionally, before launching, the SLICE framework ensures that both the SLICE framework and the selected app are up to date via an auto-update feature. The auto-update feature simply checks the version number of the SLICE framework and the app and compares them to the versions available on http://slice.cs.illinois.edu. A screenshot of the SLICE Launcher is shown in Figure 20. For other platforms, the application that is used with the SLICE framework is bundled as part of a distribution.

3.4. Conclusion

In this chapter, a technical description of the inner-workings of the SLICE framework was provided. This description began with a logical, high-level view of the SLICE framework using a common software pattern known as the Model-View-Controller (MVC) pattern. Inside the subsections of Section 3.2, the SLICE framework is discussed in terms of the “model” components, the “view” components, and the “controller” component.
Following the high-level discussions, the majority of this chapter detailed the three technical components of the SLICE framework: the core, the frontends, and the apps. Section 3.3.1 divided the SLICE core into eight subcomponents that corresponds to the structure of the source code that runs the SLICE core. Section 0 examined the three SLICE frontends that are currently available and their platform-specific nature, allowing for each SLICE application to run natively on any supported platform. Finally, Section 3.3.3 discusses the process that the SLICE framework goes through in launching a SLICE application.

This chapter was limited to a discussion on the inner-working of the SLICE framework and did not discuss an application developer’s interaction with the SLICE framework or how one would go about building a SLICE application. A guide to building a SLICE app and understanding the interactions between an app and the SLICE framework was provided in the previous chapter.
Chapter 4

Using SLICE with Tablet PCs in a Code Review Class

4.1. Introduction

In 2007, University of Illinois instructor Michael Woodley and Professor Sam Kamin presented a SIGCSE talk on a new course offering at Illinois: CS 242, Programming Studio [39]. The goal of this course was to give undergraduate students an opportunity to learn how to program well, as opposed to simply completing the task at hand with code that often ends up sloppy and poorly designed. In the concluding remarks of this paper, the authors wrote about a “system for doing code reviews on networked Tablet PC’s, which supports [student, presenter, and moderator roles]” that was being developed. This system, which was written as a SLICE application and is known as the “Code Review Application”, was deployed during parts of several semesters following the 2007 publication.

The CS 241 course consists of weekly discussion sections where a moderator typically meets with 4-6 students who have each independently worked on their own version of the same assignment. The students and the moderator typically sit around a rectangular table. A large display with VGA inputs for laptops sits at the head of the table (diagrammed in Figure 21). The students take turns giving a 10-15 minute presentation on their code. The students who are not

![Figure 21: Typical setup of a CS 242: Systems Programming section](image)
presenting are expected to critique the code presented and offer ideas on how the presenter could improve his or her code and/or design.

During the Fall 2010 semester, the Code Review Application running on Tablet PCs was deployed in a select number of CS 242 sessions in preparation for a full deployment in Spring 2011. During this initial trial deployment, several observations were made about how students interacted with and without the Tablet PC. These observations, followed by our experimental design and hypothesis, are detailed in Section 4.2. In Spring 2013, the Tablet PCs running the Code Review App were deployed for part of a ten weeks experiment in six sections of CS 242. The data collection and analysis of the data is discussed in Section 4.3.

As part of the experiment, a metric was developed that uses the features of a conversation as a proxy to student vocal participation (a metric that may be very close to capturing if the student was engaged in the conversation). This metric, called “vocal participation”, drives the majority of the results. One of the most notable results is that the use of the Code Review Application increased the average vocal participation by over 16% (from 0.68 to 0.84). This result and all of the other results are discussed in Section 4.4.

4.2. Experiment Design and Hypothesis

The idea of adding Tablet PCs to facilitate code review had been considered since the course was created [39]. However, it was also important that evaluation was done to measure the success of adding this technology. To help understand how students might use the Tablet PCs, they were introduced in two sections of CS 242 for a total of four weeks in the Fall 2010 semester. During the time before, during, and after the use of the Tablet PCs, many of the sections were observed by a non-participant. This means that some observations included use of the Tablet PCs and some observations were done using the traditional setup of using the flat panel display.

When viewing these discussion sections, observers noted the difficulty of explaining code with a laptop computer connected to a large display. When a student wants to focus on only a few specific lines of code, pointing at the screen or highlighting the lines in a source code editor were often the only two options. Additionally, observes noted that discussions often devolved
into conversations between only two people. It is believed this may have been caused by the fact that the entire state of the conversation was verbal – no written notes, no diagrams, only a mental progression. As soon as anyone else in the discussion got lost, it was impossible for them to be brought up to speed in the discussion. The Code Review Application was designed to address these shortcomings. Instead of having a single display, each participant (student and moderator alike) would have a Tablet PC that served as a shared whiteboard.

The app allows the presenter to load their source code onto the shared whiteboard. Each separate file received a separate tab at the top of the screen. Every participant could use one of several ink tools, including pens and highlighters of various colors, to annotate or highlight the code. Each participant could browse the code independent of the presenter or “sync” with the presenter. When synchronized with the presenter, each participant’s Tablet PC would mimic what the presenter sees. The large display that is installed at the head of the tablet was used to always show a synchronized view. Figure 22 shows a recent version of the Code Review Application and Figure 23 shows pictures of CS 242 sections meeting with and without Tablet PCs.

The app also had some limitations on functionality. There can only be one presenter at any given time, therefore no student can “pre-load” his or her code into SLICE. While all
students could write on the shared whiteboard, there was no way to determine who wrote a specific stroke on the whiteboard. To solve this, it was observed that most students would self-select a color different from other students in the section. In that way, the different colors were not used for annotating a diagram but to provide information on who was the author of each stroke.

When the Tablet PCs were introduced, it was believed that digital communication channel provided by the tablets would displace the traditional oral channel. However, it was observed that the tablets might have the opposite effect, actually enhancing oral communication in the meetings. That observation gave rise to the current study. It was hypothesized that we could record the conversations in CS 242 discussion sections, analyze then, and quantify the changes. The next section in this chapter details the process of collecting the audio and the analysis done to quantify the changes in vocal participation.

4.3. Data Collection and Analysis

The Studio discussion can switch easily from the traditional structure to the tablet structure. The design of the experiment was to have the discussions operate first in the traditional setting, introduce tablets for several weeks in the middle, and then move back the traditional setting towards the end of the semester. During this entire time, audio recordings of each
meeting were made.

The experiment ran in the Spring 2011 semester for ten weeks, with six sections participating, with a total of 17 students and 4 moderators. Starting with the fifth week and continuing through the entire semester, the students and moderators for the sections were recorded. The recordings were done using small lapel microphones worn by each student, connected to personal voice recorders. Specifically, we used the RCA VR5220 Digital Voice Recorder for this experiment, shown in Figure 24. In weeks 1 – 6 and 11 – 14, the Studio ran in its traditional structure; in weeks 7 – 10, the tablets were used. Figure 25 shows the full schedule of recordings and use of Tablet PCs.

To have minimal impact on the meetings, Sam Kamin, Chris Cortez (an undergraduate working as part of the research group during the Spring 2011 semester), or I attended only the first few minutes of the first meeting where recorders were used, to show the participants how to use them. We instructed the students not to stop the class if the recorders malfunctioned. After
each week, we collected the audio from the devices for storage. The stored recordings were
tagged with the participant's name, and the time and date of the meeting.

The large majority of participants from each section had successful recordings. However,
for both mechanical reasons and human error, we did not always obtain a complete recording for
every participant in every section. The audio data that was collected from the personal audio
recordings came in as single-channel audio at a sample rate of 8000 Hz and totaled over eight
days of continuous audio. Table 12 presents basic statistics about the recordings.

<table>
<thead>
<tr>
<th>Week</th>
<th>All</th>
<th>5 – 6</th>
<th>7 – 10</th>
<th>11 – 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablets Used?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Recorded Meetings</td>
<td>49</td>
<td>11</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Total Recordings</td>
<td>222</td>
<td>47</td>
<td>97</td>
<td>78</td>
</tr>
<tr>
<td>Length (Hours)</td>
<td>195.1 h</td>
<td>53.81 h</td>
<td>72.3 h</td>
<td>69.0 h</td>
</tr>
<tr>
<td>Size (GB)</td>
<td>10.46 GB</td>
<td>2.89 GB</td>
<td>3.88 GB</td>
<td>3.70 GB</td>
</tr>
</tbody>
</table>

Table 12: Basic information on audio recordings

4.3.1. Identifying Turns

The setup for this experiment had participants in a section wear their own microphone
and recorder. When listening to an audio file, a listener could make out the entire audio of the
discussion section – all participants, not simply the participant wearing the specific lapel
microphone associated with the recording. However, the volume level of the participants who
weren’t speaking directly into the microphone was quiet. This quiet audio, the speech of other
participants in the discussion section, was labeled as “cross talk” and was present in every
recording in every section.

The presence of this “cross talk” turned out to be very useful. To begin more detailed
analysis of the audio, I used the “cross talk” to precisely synchronize the audio recordings in
each meeting. Since part of the tagging of the audio files included which section and day the file
was recorded, the audio files already were grouped with other files that were recorded at the
same specific section of CS 242.

The process to perform this synchronization was a simple three-stage process. This
process synchronized files in a pair-wise fashion, such that the end result effectively resulted in a
multi-channel audio file where each channel was the microphone of one of the participants. The two stages in the algorithm are:

- First, look for an interesting segment of audio (explained in the next paragraph) in one of the two audio files.
- Attempt to find the same audio in the second file using a volume-invariant algorithm.

The first stage, finding an interesting segment of audio in a file, was the most complex stage of this process. To do this, I made use of the fact that, while the “cross talk” resulted in audio showing up in every file at some audio level, periods of complete silence shows up as complete silence on every audio file. The algorithm found a region of the audio interesting if there was approximately 50% silence and 50% volume across a one-minute clip of audio.

The second stage, finding a match in a second file, was simply to look for the “fingerprint” of the silence-volume that was identified as interesting by the first stage of the algorithm. Since the source audio was 8000 Hz, a sixty second clip of audio resulted in a 480,000 point fingerprint. Each data point was labeled as either being silent or having volume. This fingerprint was then placed at every moment in the second audio file to find how well the fingerprint matched that moment in time. The best match, the result of one pass of this algorithm, would be the offset in time between the start of the first audio file and the start of the second audio file. For example, if the sample audio from the first file was taken at +00:03:13.400 and the match was found in the second file at +00:06:31.600, the offset in time between the first and second file is the difference of +00:03:18.200.

Even with this simplistic approach, the results were near perfect. It was very rare for the algorithm to ever, even once, pick up an offset that was wrong. To ensure that all of the synchronization was correct, the algorithm would not consider files synchronized until the same offset was found using five different “interesting” clips of audio.

The next phase of the analysis relied on the observation that the channel with the loudest volume (over a threshold of background static) was almost certainly the speaker at that time. These periods of volume, called “volume events”, were combined into “speech events” if multiple volume events occurred by the same speaker in a short period of time. Figure 26 shows a synchronized waveforms with “speech events” identified.
To validate this approach, three different humans tagged a recording with the marks for when a participant spoke. The small differences between the human interpretations of the audio were within the same margin as the differences between any human's interpretation and the algorithm’s result. Given that the three humans did not completely agree on what volume consists of a “volume event” and that the algorithm did just as well of a job, this provided confidence that our algorithm would do as accurate as a job a human would.

As a whole, the treatment of the raw data transformed groups of 8,000 Hz input waveforms into a series of “speech events” for each participant. In the context of a synchronized conversation, it was said that were a person had a “speech event” it was their turn in the conversation. Like each speech event, each turn had a speaker, a starting time, and a length. For all the analysis from this point forward, the results are based on the turns identified by the algorithm that was just described.

4.3.2. Vocal Participation

The hypothesis was that the tablets would have a positive effect on the oral channel. Specifically, the goal was to find how often an individual in a meeting of the studio section would contribute to the conversation. To measure this, a “vocal participation” metric, denoted by the Greek letter epsilon ($\varepsilon$), was developed.
This metric asks a simple question: “At a given moment in time, how many of a meeting's participants had a speaking turn within the last time window?” ε is the ratio between the number of participants who had a speaking turn within the time window and the number who were present at that time. If the window was one minute, and all the participants spoke within the past minute, ε would be 1.0. If only two of the six participants spoke, ε would be 0.33. The solution of calculating ε is shown graphically in Figure 27.

I analyzed the average ε across the full meeting, for several window sizes. For each hundredth of a second (0.01s) in each meeting, ε is calculated. For an hour long section, this would result in 360,000 individual values. Effectively, we have performed a detailed approximation of integration across the entire meeting.

Consider an average ε of 0.7 for a given meeting. This value would indicate that, across the entire meeting, an average of 70% of the participants contributed to the conversation within the last time window of time.

Up until now, no time window has been specifically defined. Using a small window, on the order of only a few seconds, would be uninteresting as one wouldn't consider someone “unengaged” or a “non-participant” in a conversation if they went just ten seconds without speaking. On the other hand, a large window would be equally uninteresting as everyone is
likely to get a turn in each window. Table 13 shows time windows ranging from 30 seconds to 300 seconds. What was most notable was that, for any reasonable time value, the same trends and statistical significance existed in the data. The focus of most of the analysis will be on a window of one minute (60 seconds), but, as Table 13 shows, the statistical significance remains throughout a full range of time windows.

4.4. Results

The key result of this work is presented in the graph in Figure 28. This graph shows the average $\varepsilon$ of all 49 meetings that were recorded, using a one-minute window. On average, the introduction of tablets significantly boosted the average $\varepsilon$ from less than 70% to nearly 85%.

Looking deeper into the data, I compared the meetings on a section by section basis. As the CS 242 studio course was a graded course and students signed up for a single, specific section, there was little variance week-to-week in the students who attended each section. Figure 29 shows the same data that is displayed in Figure 28, but displays the points grouped by section. There is very little difference between the sections. By in large, the sections are identical.

One of the first questions I had on the data was if the number of turns impacted $\varepsilon$? Since $\varepsilon$ is defined by the number of people who spoke within a time window, if there were more turns in the conversation then one would expect $\varepsilon$ to be higher. However, analysis on the turns showed that the Tablet PCs did not affect how much or how often participants took turns. On average across all the audio files:

- With Tablet PCs: 10.41 turns /minute
- Without Tablet PCs: 10.59 turns /minute

Furthermore, the number of participants in a given section is another factor that might have a strong impact on $\varepsilon$. However, the results presented range from 2–6 student sections (3–7 participants) and the section size proved not to be correlated with $\varepsilon$.

---

1 When students were unable to attend their assigned section due to a conflict, they were allowed to attend other sections. Our analysis used all students present at the section, even when a student only appeared once.
Up until now, the results have focused on the notion of the average overall speech in a section. Just as interesting as an average overall view of the various sections are the individual participants within each section. To analyze an individual’s contribution to the discussion, a metric similar to $\varepsilon$ was developed. This metric, denoted as $\gamma$, is the percentage of time in the meeting that the individual is “participating” in the sense of having had a turn within the previous time window. For example, a $\gamma$ of 0.8 indicates that, at any given time in that meeting, there was an 80% chance that this participant had spoken within the previous window.

Figure 30 shows a graph of each participant’s individual $\gamma$ averaged over the sections with Tablet PCs and without Tablet PCs. The moderators are denoted with an asterisk. For the first 17 participants, the lighter colored region at the end represents the increase in $\gamma$ experienced when Tablet PCs were used. For the last 4 participants, the lighter colored region represents a decrease in $\gamma$ when Tablet PCs were used. From the graph, four observations were made:

- The vast majority of students (15 of 17) had more vocal participation when using tablets. The average student saw a boost to their $\gamma$ by 0.0834 (an increase of 11.64%) with tablets.
- Of the students who had the least vocal participation without the use of tablets (participants A, C, D, and E all had $\gamma < 0.50$), all saw a significant boost from the tablets (an average increase in $\gamma$ of 0.1726, or 38%).
- For two of the moderators, $\gamma$ increased when using tablets, but the increases were small. For the third moderator (U), $\gamma$ changed by less than 0.1%. For the fourth moderator (B), $\gamma$ decreased, and by a much greater amount (0.73 to 0.57) than either of the two students whose $\gamma$ decreased. It is not surprising that moderators should show up differently from students – after all, if the students are “unengaged”, we might expect the moderators to talk more – but there is not enough data to draw definite conclusions in this regard.
- Preforming pair-wise analysis between the student’s $\gamma$ with and without Tablet PCs, the results are statically significant (two-tailed paired t-test: $p = 9.14 \times 10^{-4}$ among all participants; $p = 4.16 \times 10^{-5}$ among only students).
<table>
<thead>
<tr>
<th>Window</th>
<th>Mean</th>
<th>St. Dev.</th>
<th>Mean</th>
<th>St. Dev.</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>30s</td>
<td>0.5682</td>
<td>0.0508</td>
<td>0.7401</td>
<td>0.0230</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>Δ = 0.1719 (+30.3%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60s</td>
<td>0.6765</td>
<td>0.0519</td>
<td>0.8446</td>
<td>0.0189</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>Δ = 0.1681 (+24.8%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120s</td>
<td>0.7902</td>
<td>0.0438</td>
<td>0.9292</td>
<td>0.0135</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>Δ = 0.139 (+17.6%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180s</td>
<td>0.8526</td>
<td>0.0343</td>
<td>0.9607</td>
<td>0.0095</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>Δ = 0.1081 (+12.7%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300s</td>
<td>0.9190</td>
<td>0.0209</td>
<td>0.9818</td>
<td>0.0056</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>Δ = 0.0628 (+6.8%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 13: Statistical information on ε across different windows with and without Tablet PCs

Figure 28: Average “vocal participation” (ε) per meeting. (Tablet / Non-Tablet: p < 0.001)
Figure 29: Average $\varepsilon$ per meeting, grouped by the six sections that were recorded as part of this study.

Figure 30: Average $\gamma$ per student, averaged across all meetings.
In order to make these measures more concrete, Figure 31 shows Student “D”’s turns in two meetings: one with Tablet PCs and one without. Observe that in both meetings, there is a period of high speech activity; this is to be expected: it is when student D is presenting. More interesting are the other parts of the graphs, when student D was not presenting but was one of six reviewers. The graphs show clearly that student D had more vocal participation when using the tablets.

4.5. Conclusion

This chapter described how, in a course in which Computer Science students gather for mutual code reviews, Tablet PCs were introduced to help facilitate and enliven the discussions. Through the recording of ten weeks of weekly meetings across six different sections, detailed analysis was done of the vocal participation of the various participants in each section.

To perform this analysis, a “vocal participation” metric was established that evaluated how lively or engaging the discussions were in each meeting. This metric is defined simply as how many participants had a “turn” in the discussion within a window of time before the current time. Using this metric, we found the use of tablets, for every reasonable window of time (from 30 seconds to 300 seconds), significantly increased the “vocal participation” in all of the discussion sections (p < 0.001).

Further, this chapter presented analysis on the percentage of time each individual student was participating in each of the meetings they attended. With the introduction of tablets, the large majority of students had more vocal participation (15 of 17), and, interestingly, the students who had the least vocal participation gained the most in our metric (an average of +17.26%).

Figure 31: Student D's turns in the discussion from two different meetings, each approximately 75 minutes in length. At full resolution, each horizontal pixel represents one second of the meeting. A black line at a specific pixel for a given meeting indicates that a speaker was speaking during that second of the meeting. Graph (A) shows a meeting where Tablet PCs were used while Graph (B) shows a meeting where Tablet PCs were not used. Each graph shows a large, dark area where Student D was the presenter. However, Graph (A) shows significantly more total turns by Student D throughout the meeting outside of his presentation.
found strong significance in the increase in “vocal participation” of the participants as a result of the use of tablets (p < 0.001).
Chapter 5
SLICE in Classroom Lectures

5.1. Introduction

One of the flagship applications of SLICE has been the “Lecturer Application”. The Lecturer Application is a PowerPoint-like presenter that allows a classroom lecturer to use a Tablet PC to present a set of slides and annotate them with ink strokes. In the initial versions of the Lecturer Application, this process was done by one machine connected to a projection system. In more recent versions, the Lecturer Application can be launched on separate devices with different roles, including: a “lecturer role” used by the lecturer to present his or her slides, a “display role” used on the house computer to display the current slide and annotations currently being presented by the lecturer, a “student role” used by students to follow along with the lecture on their own machine, and a “dashboard role” that allows for a lecturer or TA to view the students’ displays.

With the ease of customization of a SLICE application, the Lecturer Application has

![Figure 32: An iteration of the Lecturer Application in use in a classroom using a separate dashboard display.](image)
(a): An on-going lecture in a CS2-style course. (b): The dashboard display monitor. (c): The dashboard display visible to the lecturer.
undergone several revisions in the previous three years to meet differing classroom objectives. Two of these versions were written into the Lecturer Application by Professor Sam Kamin, a third version was written into the Lecturer Application by two undergraduate students working on an undergraduate research project, and I developed a fourth version of this same application based on bi-weekly survey feedback from students during the Summer 2012 semester. The use of SLICE made the development of the Lecturer Application much simpler than what would be required to develop the full application in Java or C#.

The first application allows the lecturer to *constantly* monitor a subset of students. It employs an additional display, placed in a location where the lecturer, but not the students, can see it. That display shows a dashboard of all of the current students’ notes. Figure 32 shows a dashboard with four students’ work displayed. The instructor can gauge student progress on a question, answer questions posed by the student through the tablet, or gauge understanding of a topic by looking at the students’ notes. This application can be helpful in an active learning environment (see the active learning app in Figure 33), but it is mainly intended for a more traditional classroom; the primary feedback to the lecturer is the notes that students are taking during the lecture.

The second application addresses classes that employ active learning, where a considerable amount of class time is devoted to student problem-solving. This is similar to the

![Abstract syntax of OCaml](image)

**Figure 33:** A student view of a Lecture Application where a student’s work has been completed by the instructor.
first application (the “dashboard”), but the dashboard resides on the lecturer’s own tablet, and student work can be seen only by changing modes. In “monitoring mode,” the lecturer can scroll through the students who are using tablets and see their progress on the current in-class exercise. The instructor may then write directly on a student’s slide privately (in effect, sending a message to that student), but the more important feature is that he can show a student’s slide on the classroom display. This use of a peer’s answer publicly allows for the entire class to collaborate on an answer with the instructor being able to fill in errors or complete the student’s answer. Figure 33 shows a student machine’s display where the instructor completed the answer the student originally wrote.

A third version of this application, built by two undergraduate students doing an undergraduate research project, focuses on providing the instructor feedback immediately after the class, to be used to guide subsequent classes. Instead of a student’s work being shown to the instructor, the student uses a special interface to leave questions on the slides. At the end of lecture, the instructor can review those questions.

Finally, as part of teaching CS 241 during Summer 2012, I ran a semester-long experiment where students’ feedback was used to motivate the development of the Lecturer Application. This experiment was possible due to the smaller class sizes of the Summer semester. After several iterations, the Lecturer Application that was developed based on student feedback was closer to a shared whiteboard allowing for collaboration between students and lecturer for working through in-class sample problems. The results to student’s surveys also showed some strengths and weaknesses of the app.

5.2. A Large Lecture Experience with the Lecture App

The first and second versions of the Lecture Application discussed in the introduction both used the idea of a “student dashboard” where a lecturer is able to actively monitor the student’s work. Professor Sam Kamin used the application extensively in CS 421, Programming Languages and Compilers. Programming Languages and Compilers, is a required course in our CS curriculum, normally taken by students in their senior year. It is taught in twice-weekly, 75-minute lectures, in a large lecture hall; enrollment is approximately 150 students. The course
material is generally considered quite challenging. Homework is given weekly and usually involves programming.

The goal of each lecture is mainly to teach the students the new skills needed to complete that week's homework. Accordingly, the in-class exercises are versions of the same kinds of problems as in the homework, often building up from much simpler versions of those problems, which can be solved completely in the class, to fuller versions that can only be solved partially. The goal is to leave students in a good position to complete the homework. For example, the second assignment of the semester is to write functions over linked lists using recursion, in the programming language OCaml [40]. The in-class exercises that week built up from simple non-recursive programs on lists, to complicated recursion. Later in the semester, “compilation schemes” – rules for generating machine language code for various high-level language constructs – are taught and the exercises involve writing these rules; the exercises start with simple examples that involve little more than copying examples given by the instructor, to creating rules from scratch.

The exercises are included within the class lecture slides. The slides containing exercises are printed before class and handed out to all students. (The lecture slides themselves are posted online before class, but very few students print those ahead of time.) In addition to the printed exercises that are given, at the start of each class a set of four computers were distributed to students randomly. All the instructor’s notes, including the exercise slides, are transmitted to those machines wirelessly. The students are asked to do one exercise on them and then pass them along to someone else. Classes usually have at least half a dozen exercises, so a significant portion of the class will use the tablets at some time during the class.

Figure 34 shows the instructor’s version of the Lecturer Application used in CS 421. It is an ordinary Tablet PC-based presentation app, with buttons to change pen colors, erase pen strokes, move to the next slide, and so on. Most interesting is the set of buttons near the bottom left, highlighted in the figure. The “heads” button switches the instructor's machine to monitoring mode, displaying the first student's tablet on the lecturer’s tablet (at the same page as

---

2 The choice to choose four tablets were for the most mundane of reasons: it is the number we can comfortably carry from the lab to the class. The app can handle any number of student machines, although there is a real question of how many response the professor can realistically monitor during the exercise periods.
the lecturer’s tablet, that is, the current exercise page). The arrow buttons next to it move from student to student. The display icon on the left displays, on the room display, the page of whatever student the professor is currently viewing; this allows the instructor to either show a good solution or point out an error that he suspects is common. The student’s version of the app is similar, but with fewer buttons. It lacks the monitoring buttons, of course, but also lacks the erase button; students can only erase by crossing out. This was intentional, so that the instructor could see the students’ false starts.

The application is effective in giving the lecturer a window on the students' thought processes. The students seem to enjoy using it, and will sometimes draw pictures, or write their answers in a rainbow of colors. (The machines are very simple and natural to use; we have had no need to give the students any training on how to use them.)

Sometimes, the best information comes from the lack of student responses to an exercise; if none of the four students is working on an exercise after, say, one minute of thinking, that strongly suggests that those students --- and most likely, almost all the students --- are confused by the problem. But one of the benefits of getting this feedback constantly is that the professor learns over time how to pace the class, so that he gives fewer and fewer exercises like that. Or, to put it differently, he learns how to be so clear and concrete that students always understand
what he asking of them. Nearly all the time, the students respond, although at very different speeds.

More typical are responses like those shown in Figure 35, where most of the students are completing the questions. One student will get the first part of the problem, and the professor will use that as the starting point for a discussion, and so on. (These screenshots show the students’ final responses, after they have benefited from the entire discussion.) In programming classes, it is worthwhile to point out even trivial mistakes, because these can cost a lot of time when they sit down to do their homework.

Another way the system supports active learning is that it helps the teacher calibrate the pace of the exercises and, by extension, of the course material itself. The best example came from the very beginning of the course. Lecture 2 covers basic material that the students have seen before: writing recursive functions. Either because they hadn’t done this in some time, or because they were using a new programming language, they struggled with these exercises. Because the material is so fundamental to the course, the professor changed the schedule of the
course, and continued working on these problems in class 3. Absent the monitoring system, it may have seemed that the students were confused or disengaged, but the system provided concrete, unignorable evidence that he needed to spend more time on this topic.

5.3. Summer 2012 Experiment: What is an optimal lecture format?

In Summer 2012, I had the opportunity to be the sole lecturer for CS 241: Systems Programming. As part of this course, in an effort to show the ease of development of the SLICE framework, an experiment was designed to use student feedback as the primary mechanism to motivate the format of the lecture. The students would see one week of a “classical” style of lecture using PowerPoint slides, one week using a basic version of the SLICE Lecturer Application, and then the students would be surveyed on which format the students felt was most helpful in learning the course material. The results of the survey would be used to motivate the format of the lecture for the next week of lectures.

The Summer offering of CS 241 was an 8-week course that met for six hours each week (three meetings each lasting two hours). Unlike the CS 421 course discussed in the previous section, or the offerings of CS 241 during the full semesters, the summer offering was a much smaller class of only 30 students. This smaller class size presented a manageable number of students to allow for near weekly changes in the format of lecture.

To begin the semester, the first week of lecture was given entirely using traditional PowerPoint (without the use of SLICE in any way). For the second week, a simple version of the Lecturer Application was used consisting of only of a “display” and a “lecturer” (that is, unlike CS 421, at this point no Tablet PCs were given to students). Even though there were only two roles, occasionally I gave my Tablet PC to students to complete the problem\(^3\). At the end of the second week, the first of four surveys were given to the students. The surveys were given anonymously and I left the room while the students filled out the survey, to remove as much bias as possible.

In this first survey, there were a series of questions that asked the student to rate statements on a scale from 1, very unhelpful or not useful at all, to 7, very helpful or very useful.

\(^3\) Cinda Heeren, who uses SLICE in CS 225, has been using SLICE in this way for some time.
There were three questions that specifically targeted the differences between the use of PowerPoint and the use of SLICE:

- Question #1.3: During the first week of class, lectures were given in a “classical” lecture style. Thinking back to those lectures, how was the “classical” lecture style?
- Question #1.4: During the second week of class, lectures were given in a “pen-based” lecture style. Thinking of the lectures this week, how was the “pen-based” lecture style?
- Question #1.5: During lecture, some in-class examples were done by fellow students using educational technology running on TabletPCs called SLICE. How helpful was the use of this SLICE technology to this course?

Additionally, there were three other questions regarding SLICE:

- Question #2.1: Thinking of the in-class SLICE technology, have you had a chance to use SLICE with a TabletPC?
  - Provided answer choices: “No, I have not”, “No, but I would like to in the future.”, and “Yes, I have.”
- Question #2.2: During the semester, we will be updating the in-class technology each week based on your feedback. Thinking about how the technology could be more useful for your learning, what would change about the SLICE application being used?
- Question #2.3: Is there anything you particularly like or dislike about the SLICE application being used this week?

In all, these questions were designed to gather an anecdotal, but detailed understanding of the student’s opinion on the use of SLICE in the classroom.

The results of the survey showed a significant preference for the style of lecture using the SLICE framework. Table 14 details the results of the first set of questions. In the results, it is particularly notable that the lecture style using SLICE scored over a full pointer higher (5.14 vs 6.43) and had a much smaller variance. That is, the average student thought lectures using SLICE was more helpful and many more individual students felt the lectures using SLICE was

---

4 The full survey is provided for the reader in Appendix A as documents A.1 and A.2.
“very helpful” (a rating of 7/7). Further, the large majority of students who hadn’t had an in-class interaction with SLICE also “would like to [use SLICE] in the future” (see Table 15).

Given the results, SLICE would be used in the following week since the survey supported SLICE as the style of lecture that the students felt was most helpful to their learning. Moreover, the responses to how the students felt SLICE could be improved would be used to motivate the development of the next iteration of the Lecturer Application used. On Questions 2.2 and 2.3, several students commented on two aspects of how the SLICE framework had been used so far:

- It would be nice if everyone could use SLICE
- When you didn’t have the tablet, it was hard to do the examples.

These responses were not unexpected. Both of these responses get at how SLICE was used in other classes at Illinois (such as the CS 421 class discussed earlier in this chapter). To respond to
these changes, two changes were made: handouts with examples were printed before class to distribute to the students to allow students not currently working on SLICE to do the in-class examples, and several additional Tablet PCs were added with a “student” role.

Since the changes were generally minor, the second survey was given only a week later at the end of the third week. One notable change to my lecture format was, now that students worked on their own worksheets and/or Tablet PC, the lectures paused for students to work through the example. Several of the questions in the Week #3 survey looked to distinguish if this was a helpful feature, independent of SLICE.

Table 16 shows several key questions and their results. The most overwhelmingly positive result was the introduction of handouts to the course (Question #1.5). Additionally, no students found the Tablet PCs that moved throughout the classroom were “unhelpful” (Question #1.6). Students also preferred the lecture to pause for them to work through problems themselves rather than working with the instructor on the problem in real time (Questions 2.1 and 2.2). As with the first survey, students were also asked about their overall thoughts on SLICE and the overwhelming majority of students found it helpful (Question 2.3).

Similar to the first survey, students were asked what changes that they would like to see made in the SLICE application to help their classroom experience. Unlike the first week, there were fewer students who responded to the question. However, many of the students who did respond expressed a desire to see SLICE have some form of source code integration for doing the C programming examples that are a major component to CS 241. This result posed some interesting challenges.

Integrating the ability to compile and show running C code within the SLICE framework would be a challenging task that would be near impossible to accomplish in the five days between the survey and the next lecture. However, while discussing these results, the question arose if students preferred SLICE only because of the handwriting capabilities? Are the interactions that the lecturer can have with the “dashboard”, such as responded to student’s work, worthwhile? Is seeing answers that other students wrote for a question helpful? These questions had not been tested by the experiment yet.
For the fourth and fifth week of lectures, the format of the course was changed dramatically. SLICE and Tablet PCs were removed. PowerPoint was also removed. The only aspect of the lecture that carried over from the previous week was that handouts were still provided with the problems that were going to be worked out in lecture. However, all examples would either be done inside a source code editor or on the whiteboard. That is: if the problem was coding, it would be coded live; if the problem was not coding, it would be worked and explained on the board.
At the end of the fifth week of lecture, a third survey was given. In this survey, students were asked to think about all three styles of lecture they had seen and make pairwise comparisons of each lecture style. Table 17 shows an overview of these results.

Two of the results were very clear: students preferred lectures using SLICE than lectures without SLICE (Question #2.3) and students preferred the whiteboard/live-coding over the traditional style of lecture (Question #2.2). However, the transitive conclusion from those results, that students would then prefer SLICE over the whiteboard/live-coding was unclear. In the survey, Question #2.1 had a slight bias towards the whiteboard/live-coding over how SLICE was used during the earlier weeks in the semester.

Examining the written responses, 75% of students that responded on how SLICE could be improved noted that IDE integration would improve SLICE. Based on these results, a hybrid result was used to finish off the semester: live coding was still done in an IDE, but SLICE was re-introduced and used as a sort of a “shared whiteboard” when in-class practice problems were done. That is, lecture was paused for students to work on the problems and then one student

<table>
<thead>
<tr>
<th>Week #5 Survey Results</th>
<th>Avg.</th>
<th>Var.</th>
<th>n</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2.1: During the second and third week of lecture, SLICE was used to allow for real-time student interaction with lecture with the use of Tablet PCs. During the past week, the SLICE technology was not used and lecture focused on solving problems by editing C code in an editor and doing examples on the whiteboard. Thinking of these two styles of lecture, which one would you prefer? 1: SLICE ... 4: Neutral ... 7: Whiteboard/IDE</td>
<td>5.06</td>
<td>2.60</td>
<td>16</td>
<td>![Bar Chart]</td>
</tr>
<tr>
<td>#2.2: Thinking of the two lecture styles without SLICE (Week #1: Traditional Lecture and Week #5: Coding/Whiteboard), which of these two methods do you prefer? 1: Traditional ... 4: Neutral ... 7: Whiteboard/IDE</td>
<td>6.06</td>
<td>1.26</td>
<td>16</td>
<td>![Bar Chart]</td>
</tr>
<tr>
<td>#2.3: Thinking, in general, of the course with or without SLICE technology, which one would you prefer? 1: No SLICE ... 4: Neutral ... 7: With SLICE</td>
<td>5.44</td>
<td>2.66</td>
<td>16</td>
<td>![Bar Chart]</td>
</tr>
</tbody>
</table>

Table 17: Survey results from the third CS 241 Summer 2012 survey (Week #5)
would use a “student” Tablet PC to present his or her answer. When lecture material was given, SLICE was used as a replacement for PowerPoint, as it had been since the second week of class.

At the end of the semester, an attempt was made to deploy one last survey. Unlike the first three surveys, at this time the lectures had ended and a survey could not be given at the end of a lecture. Instead, an online survey was developed and sent to students. However, only three people responded to that survey, making the results not representative of the class as a whole and, therefore, not reported.

5.4. Limitations

When the Lecturer Application was used, Tablet PCs were distributed to three to four students. The students who initially had the tablet would do one example and then were asked to pass the tablet on to another student. In doing this, two things were achieved: several dozen students would use SLICE during the course of a class and it prevented one student’s work to always be used since a single student would rarely have a tablet more than once.

An experiment was never done using the Lecturer Application in a “1-to-1” environment (one tablet per student). Likewise, limited SLICE usage data was collected as part of this experiment. The primary goal of the experiment was to evaluate the use of different styles of lecture and detailed analysis of what was written might be interesting future work.

As noted earlier, the surveys given as part of the Summer 2012 experiment in CS 241 were administered to the class by me (and I also served as the lecturer of the course). While I left the room while the students took the survey and assured them that they would be anonymous and would not affect their course grade, there was likely some bias in the survey results.

5.5. Conclusion

In this chapter, one of the flagship SLICE applications, the Lecturer Application, was discussed. At a minimum, this application allows for a lecturer to remotely use a Tablet PC to annotate lecture slides in real time in a classroom. Specific versions of this application also allow lecturers to monitor the students via a “dashboard” display. One version of the Lecturer
Application, discussed in 5.2 allows for the lecturer to show the student’s work to the full class for an element of active learning or privately communicate with the student without displaying the work to the class to correct an error as it’s being made.

In Summer 2012, a student-driven approach to the lecture format was done as an experiment in CS 241. Through surveying the students when different lecture styles were used, several key observations were made about the SLICE Lecturer Application:

- The SLICE Lecturer Application is a highly effective replacement to traditional PowerPoint applications
- For very task-specific problems, such as coding in the C programming language and showing the output of the program, using a more native interface just an IDE proves also to me much more effective than a “traditional” lecture and only slightly more effective as the SLICE Lecturer Application.

The SLICE application described in this chapter was used in CS 225, CS 241, CS 421, and CS 105 for the majority of lectures in the Spring 2013 semester. It is anticipated the SLICE will continue to be used next semester, even after this thesis work has been completed.
Chapter 6

Conclusion

6.1. Overview of Thesis

This thesis presented work on the development and deployment of educational software applications on a heterogeneous set of consumer-level devices. The specific devices this thesis focused on were tablet devices, those that generally have the ability to accept input through either a stylus pen or a finger. This unique input feature allows them to have several unique uses within Computer Science education.

The most significant contribution of this thesis was the development of a robust software framework that allows for a single application to be written once and run on many different devices (Windows, Linux, and Apple computers; Android-based mobile and tablet devices). This framework, called SLICE, is available for download on the SLICE website at http://slice.cs.illinois.edu.

This thesis detailed two specific applications that were built using SLICE: the “Code Review Application” for use in the CS 242: Software Studio course at The University of Illinois, and the “Lecturer Application” that has been used as a replacement to PowerPoint in CS 105, CS 225, CS 241, and CS 421. As part of the deployment of each of the apps, an experiment was designed to gather data on the impact of the application in the classroom.

For the “Code Review Application”, this thesis presents the results of a semester-long experiment that measures a student’s “vocal participation” in a code review discussion. Through the analysis of the recordings of six discussion sections each week for ten weeks, the results presented in Chapter 5 showed that the introduction of Tablet PCs and SLICE resulted in students’ vocal participation increasing by nearly 20% (p < 0.001).

For the “Lecturer Application”, another semester-long experiment was presented in this thesis. In this experiment, the format of the lecture and the use of the SLICE technology was motivated by the students’ feedback through anonymous surveys given throughout the semester.
This experiment resulted in a greater understanding of the impact of the application in classrooms, and identified some of the strengths and weaknesses of the Lecturer Application.

Through the presentation of this work, this major contributions of this thesis were:

- Development of the SLICE framework, deployable natively on Windows and Android, as well as any operating system supporting Java.
- Development and deployment of a “Code Review Application” in the Software Studio (CS 242) course at The University of Illinois
- Analysis of over 200 hours of audio recorded in the Software Studio course to measure the impact of the introduction of Tablet PCs into a code review discussion.
- Providing support to users that use the SLICE framework, such as the use of the “Lecturer Application” in CS 105, 225, 241, and 421.
- Promoting SLICE and providing support for external application developers.

6.2. Future Work

The work presented here shows the development of a strong framework for multi-platform application development. During the Spring 2013 semester, there were some minor deployments of the heterogeneous platforms in classrooms. However, no large scale study was done using a mixture of several different devices with different capability levels (eg: a Tablet PC, a simple “smart phone”, and a tablet all working together). Performing such a study would be interesting to continue the work on the cross-platform nature of the SLICE framework.

Additionally, the SLICE framework provides for the ability to quick develop applications that utilize pen- and touch-based input. To date, several flagship apps have been developed and deployed, but a larger collection or “suite” of applications would be interesting to show the full benefit of developing with the SLICE framework.

As the advancement of cloud-based services continue, an expansion of the SLICE framework’s cloud presence would be a significant next step in the developing the SLICE framework. At the current state, the SliceCloud provides a simple to use networking model to communicate between different devices running SLICE in the same classroom. An area of
future work could explore the use of the SliceCloud to store more data “in the cloud” to increase the student’s (or lecturer’s) experience with the application.

Finally, this thesis presents a new metric for using audio as a proxy for vocal participation. In the study presented in this thesis, very impressive results were obtained when Tablet PCs were introduced into a code review course. Using the metric developed in this thesis, interesting work could be done in exploring how Tablet PCs impact other forms of discussion. Further, applying the metric to other existing data sets could give researchers a greater understanding of the strengths and weaknesses of the metric.

6.3. Final Remarks

It is my hope that this work will provide insight to others in developing software frameworks for education and allow for others to develop applications to improve the state of education in large classes. Through doing this work, I strongly believe that lectures and educational experiences will look a lot closer to the experiments that were performed with SLICE than today’s lectures with PowerPoint.
References


[7] K. Nahrstedt, L. Angrave, M. Caccamo and R. Campbell, "Mobile Learning Communities – Are We There Yet?," Information Trust Institute, University of Illinois at Urbana-Champaign, 2010.


Appendix A

Complete Survey Forms

(Appendix document A.1 begins on the next page.)
### Part I:
For the first several questions, please rate each question on a scale from 1, very unhelpful or not useful at all, to 7, very helpful or very useful. Feel free to add any written feedback.

1. **Lectures (Monday, Tuesday, Wednesday from 11:00am-12:50pm)**
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Helpful

2. **Discussion Section / MiniMP (Tuesday Afternoons)**
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Helpful

3. During the first week of class, lectures were given in a “classical” lecture style. Thinking back to those lectures, how was the “classical” lecture style?
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Helpful

4. During the second week of class, lectures were given in a “pen-based” lecture style. Thinking of the lectures this week, how was the “pen-based” lecture style?
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Helpful

5. During lecture, some in-class examples were done by fellow students using educational technology running on TabletPCs called SLICE. How helpful was the use of this SLICE technology to this course?
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Helpful

6. **The Piazza online discussion board**
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Haven’t Used It

7. **Office hours (Wade on Tuesday, Brian/Yang on Wednesday/Friday/Saturday)**
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Haven’t Attended

6. **MP1 (First Steps / Basic Dictionary)**
   - 1 Unhelpful
   - 2 Neutral
   - 3 Helpful
   - 4
   - 5
   - 6
   - 7 Helpful

(This survey continues on the back of this page...)
Part II:
1. Thinking of the in-class SLICE technology, have you had a chance to use SLICE with a TabletPC?
   
   ____: No, I have not.
   ____: No, but I would like to in the future.
   ____: Yes, I have.

2. During the semester, we will be updating the in-class technology each week based on your feedback. Thinking about how the technology could be more useful for your learning, what would change about the SLICE application being used?

3. Is there anything you particularly like or dislike about the SLICE application being used this week?

4. Thinking of the class in general, do you have any feedback for lectures, the discussions, MPs, or any other anonymous feedback?

Thanks!
(You can leave this survey on the back table as you’re leaving the classroom.)
Part I: For the first several questions, please rate each question on a scale from 1, very unhelpful or not useful at all, to 7, very helpful or very useful. Feel free to add any written feedback.

1. Lectures (Monday, Tuesday, Wednesday from 11:00am-12:50pm)
   - Unhelpful
   - Neutral
   - Helpful

2. Discussion Section / MiniMP (Tuesday Afternoons)
   - Unhelpful
   - Neutral
   - Helpful

3. MP2 (malloc)
   - Unhelpful
   - Neutral
   - Helpful

4. Office hours (Wade on Monday, Brian/Yang on Wednesday/Friday/Saturday)
   - Unhelpful
   - Neutral
   - Helpful

5. Based on your feedback from the last survey, handouts were given that contained example problems to each lecture. How helpful was the use of the handouts to the course?
   - Unhelpful
   - Neutral
   - Helpful

6. During lecture, some in-class examples were done by fellow students using educational technology running on TabletPCs called SLICE. How helpful was the use of this SLICE technology to this course?
   - Unhelpful
   - Neutral
   - Helpful

Part II: For the next questions, please compare two different styles used in CS 241 on a scale of 1 to 7. Choosing 1 or 7 represents a very strong preference towards one style over the other, scaling to the mid-point of 4 representing a neutral preference of the two styles. Feel free to add any written feedback.

1. During the second week of lectures, SLICE was used to allow students to work out problems live, in real time, on the lecture notes for the full class. During the third week, handouts were given and lecture was paused to give time for each problem to be worked out by hand with select students using SLICE to work the problem. Thinking of the two uses, which one do you prefer?
   - Week #2 Real-Time
   - Week #3 Paused Lecture

2. Thinking of the two uses of SLICE from the previous question, if handouts were given alongside both lecture formats, which one would you prefer?
   - Week #2 Real-Time
   - Week #3 Paused Lecture

3. Thinking of the course with or without SLICE technology, which one would you prefer?
   - Without SLICE
   - Week #2/3 With SLICE
### Part III:

1. Thinking of the in-class SLICE technology, have you had a chance to use SLICE with a TabletPC?

   - ____: No, I have not.
   - ____: No, but I would like to in the future.
   - ____: Yes, I have.

2. In the near future, SLICE technology may be available for use on Android-based phones and tablets. Would you be interested in using SLICE on your own device?

   - ____: I don't have an Android device.
   - ____: I wouldn't want to use it on my own device.
   - ____: I would consider using it on my own device.
   - ____: I would prefer using it on my own device.

3. During the semester, we will be updating the in-class technology each week based on your feedback. Thinking about how the technology could be more useful for your learning, what would change about the SLICE application being used?

4. Is there anything you particularly like or dislike about the SLICE application being used this week?

5. Thinking of the class in general, do you have any feedback for lectures, the discussions, MPs, or any other anonymous feedback?

---

Thanks!

*(You can leave this survey on the back table as you’re leaving the classroom.)*
### Part I: For the first several questions, please rate each question on a scale from 1, very unhelpful or not useful at all, to 7, very helpful or very useful. Feel free to add any written feedback.

1. Lectures *(Monday, Tuesday, Wednesday from 11:00am-12:50pm)*
   - 1 2 3 4 5 6 7
   - Unhelpful Neutral Helpful

2. Discussion Section / MiniMP *(Tuesday Afternoons)*
   - 1 2 3 4 5 6 7
   - Unhelpful Neutral Helpful

3. Machine Problems
   - 1 2 3 4 5 6 7
   - Unhelpful Neutral Helpful

4. Based on your feedback from the last survey, we have focused on a code-based approach to teaching systems programming concepts. Thinking of lecture during the past week, how helpful has this been?
   - 1 2 3 4 5 6 7
   - Unhelpful Neutral Helpful

4. Thinking of this course in general, how helpful has the course been to learning the concepts associated with Systems Programming?
   - 1 2 3 4 5 6 7
   - Unhelpful Neutral Helpful

### Part II: For the next questions, please compare two different styles used in CS 241 on a scale of 1 to 7. Choosing 1 or 7 represents a very strong preference towards one style over the other, scaling to the midpoint of 4 representing a neutral preference of the two styles. Feel free to add any written feedback.

1. During the second and third week of lecture, SLICE was used to allow for real-time student interaction with lecture with the use of Tablet PCs. During the past week, the SLICE technology was not used and lecture focused on solving problems by editing C code in an editor and doing examples on the whiteboard. Thinking of these two styles of lecture, which one would you prefer?
   - 1 2 3 4 5 6 7
   - Week #2/3 SLICE Based Neutral Week #5 In-Class Coding and Whiteboard

2. Thinking of the two lecture styles without SLICE (Week #1: Traditional Lecture and Week #5: Coding/Whiteboard), which of these two methods do you prefer?
   - 1 2 3 4 5 6 7
   - Week #1 Traditional Neutral Week #5 In-Class Coding and Whiteboard

3. Thinking, in general, of the course with or without SLICE technology, which one would you prefer?
   - 1 2 3 4 5 6 7
   - Week #1/5 Without SLICE Neutral Week #2/3 With SLICE
Part III:
1. Is there anything you particularly like or dislike about the style of lecture used during the past week?

2. During the semester, we have been updating the in-class lecture format each week based on your feedback. Thinking about how the lecture format could be more useful for your learning, what would change about the lecture? Should SLICE integrate a coding editor?

3. Anonymous midterm exam feedback? Too hard? Too easy?

4. Thinking of the class in general, do you have any feedback for lectures, the discussions, MPs, or any other anonymous feedback?

Thanks!
(You can leave this survey on the back table as you’re leaving the classroom.)