Vivio on Android

Robert Eastman
B.A.(Mod.) Computer Science
Final Year Project  April 2012
Supervisor: Dr. Jeremy Jones

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
Table of Contents

Chapter 1
Introduction
  Overview of the Project
  About Chapter 2
  About Chapter 3
  About Chapter 4
  About Chapter 5

Chapter 2
Background
  About Vivio
  About Qt
  About Android
  About Necessitas and Ministro
  Previous Work on Vivio

Chapter 3
The Project
  Setting up Environment
  Vivio – How it Works
  Qt Necessitas
  Designing the User Interface
    Menu Screens
    Storing Animation Directories
    Creating the Buttons
  Programming the User Interaction
  Testing and Evaluation

Chapter 4
Problems Encountered
  1 Fatal Error with Ministro 2
  2 Touch Events and Mouse Events
  3 Signed / Unsigned
  4 Signal Error
  5 Random Crashes

Chapter 5
Conclusion
  Critique
  Future Work

Appendix A
  Menu Screen Images

Bibliography
Electronic Resources
Chapter 1: Overview

Introduction

The aim of this project was to get the Vivio system working on the Android mobile platform. Vivio is primarily an e-Learning tool and by porting it to Android, it opens Vivio up to a much wider audience than it previously had. With Vivio on Android it would be possible to use Vivio while commuting to and from College, or work, on the bus, or on the train.

This report will begin with a brief background to Qt, Android, and Vivio. It will then give a description of what was done to get Vivio running on an Android device, describe how the user interface was designed, and how the controls were implemented using a touch screen, and will list the major problems that were encountered, how they were diagnosed, and how they were solved.

At the end of this report, critiques will be given based on what was done, and details of what work could be done on the application in the future.

About Chapter 2

This chapter deals with the background to the various elements involved in the project. It gives a brief overview and history of Vivio, Qt, Qt Necessitas, and Android. It also gives a description of what Vivio was like before the project was started.

About Chapter 3

This chapter contains details of what was done in the project. It describes what was done, how the environment was set up, the design of the GUI and why it was developed that way, as well as the touch screen interface and how the Vivio controls were implemented without a keyboard and mouse, and how the application was designed, developed, and tested, and what it was tested on.

About Chapter 4

Each of the major problems encountered in the course of the project are documented in this chapter. These problems range from Fatal Errors, to random crashes, to Segmentation Faults. It deals with how the problems were encountered, how the fault was found, and ultimately how the problem was solved.

About Chapter 5

This chapter is the conclusion, it is essentially a summary of this report. It details the status of the project in its current state, as well as critiques on what could be done differently and what could be improved upon, and the different design choices which could have been used for the Graphical User Interface. It also deals with the potential for future work that can be done to improve the application.
Chapter 2:
Background

About Vivio

Vivio is a system that makes it easy to produce interactive reversible vector based e-learning animations for the world wide web. It was initially developed to illustrate the operation of cache coherency protocols, but has since had many more animations written for it. It allows for users to view animations for anything from quick sorts and insertion sorts to B-trees and cache coherency protocols. [1]

Vivio animations can be developed using the simple integrated development environment.

This IDE allows the creation, compiling and running of Vivio animations. It can also be used to show the decoded vcode as it will be seen by the interpreter.

Animations created using Vivio have the ability to single step forwards and backwards through frames, which makes it easy to follow complex animations where many things occur at the same time. Animations can also implement checkpoints which can be snapped to to allow the user to skip to a certain part of an animation or back to a previous part in order to see it again without resetting the entire animation.
Animations created using Vivio are stored as “.viv” files. These files are then compressed and usually uploaded online as “.vcode” files.

Vivio Hello World Code: [helloworld1.viv]

```cpp
setViewport(0, 0, 200, 150, 1);

setBorderPen(SolidPen(1, 0, rgba(0, 0, 0)));
setBgBrush(SolidBrush(rgba(1.0, 1.0, 0.75)));

r = Rectangle(0, 2, 0, 0, 100, 75, -100, -75, 200, 150,
              SolidPen(0, 0, rgba(1.0, 0, 0)), 0, "Hello World!");

r.rotate(0, 0, 360, 60, 1, 0);
```

The Vivio player works by first loading the compressed “.vcode” files, decoding them and interpreting the instructions. These instructions are then passed to a C++ switch which allows the Vivio instruction to be converted into the relevant Qt function. This allows the animation to be drawn.

Controlling the Vivio player involves using the mouse and keyboard. Each of the mouse buttons, left and right, are used as well as the wheel. On the keyboard, the shift and control, and arrow keys are used.

The controls are as follows:
- Left Click – Pause / Start
- Right Click – Context Menu
- Mouse Wheel -
  - Animation Running – Speed Up / Down
  - Animation Stopped – Back / Forward Tick
- Keyboard Arrow Keys -
  - Up – Reset Animation
  - Down – Similar to Left Mouse Click
  - Left / Right – Similar to mouse wheel
- Button / Mouse Combos
  - Control + Mouse Wheel -
    - Stopped – Speed Up/ Down
  - Shift + Mouse Wheel
    - Stopped – Snap toCheckpoint
  - Shift + Left Click -
    - Stopped – Play and pause at next checkpoint / Play to next checkpoint.
About Qt

Qt is an cross platform application framework originally developed by Haavard Nord and Eirik Chambe-Eng in 1991 before forming Trolltech[4]. It is widely used for developing applications with a graphical user interface. At first it was only compatible with Unix and Windows, but later support was added for Mac OS X. In 2008, Nokia acquired Trolltech. Nokia then used the Qt framework as the main development platform for its devices, as well as a port to the Symbian platform.

Sample Qt code Hello World:

```cpp
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel label("Hello world!");
    label.show();
    return app.exec();
}
```

This code sample will display a window with text reading "Hello world!"

On the 23rd of June 2010, Nokia released the Nokia Qt SDK[2]. This allowed for some unofficial external ports of Qt to appear, such as Necessitas for Android, and Qt-iPhone.

About Android

Android is a Linux-based operating system for mobile devices ranging from phones and tablets, to netbooks and laptops. Android was originally developed by Android Inc., prior to their acquisition by Google in 2005. Initially released in 2008, Android has a share of roughly half of the smart phone market.[3.1] Applications are run through the Dalvik virtual machine with just-in-time compilation to run Dalvik dex-code, which is usually translated from Java bytecode.[3.2]
About Necessitas and Ministro

Necessitas is the codename of the Android port of Qt and of the Qt Creator Integration with Android. It allows a user to create Qt applications for Android devices. Ministro is an Android application which will download the necessary Qt libraries for whichever Qt application you are trying to run. The Necessitas version of Qt Creator allows for the application to be configured properly by allowing you to select the relevant Qt version, Android SDK, and Android NDK. The version used in this application was Alpha 3 Update 4.

Project Before-hand

Before this project started, Vivio was written in Qt but had only been released on PCs running Windows with Internet Explorer, Firefox, Opera, Safari, and Google Chrome browsers. While it could be built on any platform that was compatible with Qt, i.e., Unix, Mac OSX, etc., it was not released. At the beginning of this project there was no existing port of Vivio on Android. The main controls for the Vivio player were the keyboard and mouse.

Below is an image of Vivio working through the Mozilla Firefox Browser.
Chapter 3:  
The Project

Setting up Environment

Setting up the environment for this project was both difficult and time consuming for many reasons. The main reason was getting the environment set up with the existing source code for Vivio. The existing source code had been programmed using Microsoft Visual Studio with a plug in for Qt. This code was compiled using the Visual C++ compiler, which was not available for the Necessitas Qt Creator. An ARM GCC compiler was used in its stead. Given the source code there were a few things not compatible with Necessitas such as spaces in path names and setting up the project space with Visual Studio project file. This meant that the environment would have to set up from scratch and the linkers, and all of the shared files and folders would have to be set up again.

Necessitas Qt Creator[2]

The Necessitas Qt Creator has the same functionality as the Official Qt Creator IDE, but is designed around use with Android. It has built in functionality for emulating Android devices, and for deploying Ministro along with any applications created.

The environment was configured for use with the tool chain ARM GCC 4.4.3 and uses the gdb included with the Android NDK. The build settings were set up for use with the Samsung Galaxy Tab 10.1 that was used for testing the application. As such, the default Android version was Android 3.1, API 12, using ARMv7.
**Vivio – How it works**

The player class, player.h and player.cpp and the code for the vcode interpreter, interpreter.cpp, were the most important code to understand when learning what the code for Vivio actually does. The player.cpp file is where all of the events involved in the user interface are captured and executed. So this is where the various different controls are implemented.

The interpreter.cpp file contains the functions necessary for the Player class to interpret the vcode files which are downloaded and converted just-in-time into Qt code for execution. The first step to learn how the Vivio player works is to look at how it decodes and interprets the vcode files that are selected to play. Each animation is first written in vcode, and then compressed.

As shown in the image below, each line of the vcode is turned into a number of assembly-like code which will be used to execute functions in the interpreter.cpp file. These functions can store variables, load variables, draw to the screen, etc.

![Vivio - How it works](image)

**Qt Necessitas**

Qt Necessitas is the Android port of the Qt application framework. It is still in alpha stage, alpha 3 update 4 at time of writing, and isn’t quite stable and doesn’t have all of the features of Qt implemented, such as predetermined touch gestures, and doesn’t have all of the features of Android available, such as the built-in context menus.

In order to ensure that the environment was set up correctly and that Necessitas actually worked, a simple application was developed. This application was a simple dial which could be rotated to change the number at the top of the screen.
This application was created, not only to see if Necessitas worked, but also to get a feel of how Qt GUI programming works. The above image shows a label containing a number and a dial which can be rotated. As the dial is rotated the number displayed changes accordingly. The code which connects the two is very simple:

```
connect( ui->SpeedDial, SIGNAL(valueChanged(int)),
         ui->SpeedLabel, SLOT(setNum(int)) );
```

Each of the components can be connected by SIGNALs and SLOTs. So here, the Dial, `ui->SpeedDial`, has a function called `valueChanged(int)`, which sends a signal to the Label, `ui->SpeedLabel`, which then updates the number it is displaying using the function `setNum(int)` using the integer value from the `valueChanged(int)` function.

The development of this application showed that Necessitas worked with Android, and also gave insight into how the various GUI components in Qt interact with each other.
**User Interface Design**

When designing the User Interface, there were a couple of design choices which were taken into consideration:

1) An application for each different animation.
   This is obviously the simplest to implement, because it is basically changing the URL variable of each application to allow for a different animation to be played. In this design there is no need for a selection menu or a back button. Basically, once you load the application, the animation runs, and then once it finishes, you quit out of the application. It has the downside of requiring multiple installations of the same application with only one line of code changed.

2) An application for each different type of animation, i.e., one application for sorting algorithms, and another application for cache coherency protocols, etc.. This design is more complex, with the application requiring another menu screen for the user to select which animation that they want to view, as well requiring a back button, to select a different animation.
   This approach could be useful for animations which may be improved by use of different touch controls. For example, the Cache Coherency animations require user input, in the form of button presses to select Read A0 or Write A0, in order to begin, while the Sorting animations just require the user to begin the animation, and watch it through to the end.

3) An application with all of the animations.
   This approach is the similar to the above but with an added menu screen for selecting the type of animation you wish to view, e.g., caches, trees, etc..

The ideal user interface is the one with access to all of the available animations, but the more animations that are available, the more complex the user interface becomes.
**Menu Screens**

There are four different screens in this application. The main screen, which acts a kind of X screen, a section screen, which has buttons for each type of animation; Cache, Cache Coherency, Trees, Sorts, and Other, a animation select screen, which has buttons for each animation of the specified type, and the player screen, which has the Vivio player embedded in to it.

Each screen is a separate C++ class, main_menu, section_screen, choice_screen, and play_screen. Each screen shares the same style; the background is the same colour, hex value #ffffc0, as the Vivio Web player web page, and the buttons are the same colour as the blue used for the banner at the top of the same web pages and many of the animations, hex value #000080.

The main_menu is simply a screen with two buttons, one which quits and closes the application, labelled “Quit,” and another labelled “Let’s Get Started.” This button creates and shows the next screen, the section_screen. On this screen, there is a button labelled “Main Menu,” which reopens the previous screen, the main_menu, and a button for each category of animation, trees, sorts, etc., labelled accordingly. When one of these category buttons is pressed, the choice_screen is created and shown. On this screen, there is another “Main Menu” button which returns to the previous screen, the section_screen, and a button for each of the animations in that section. On pressing one of the animation buttons, the player_screen is created and appears. On this screen, there is a “Back” button at the top of the screen which opens the choice_screen again, and the Vivio player covering most of the screen.

See: Appendix A for Images of the menu screens

**Storing the Animation Directories**

In order to generate the buttons for each animation, there has to be a way to store the URL directory of each of them. There are a few ways to do this but the two which were tried for this application were:

**Hard coding the URLs into the application:**

In this method, each URL is individually hard coded into the application and assigned to a button. So, there would be a number of lines code dedicated to creating a button for each of the urls associated with the animations.

```cpp
choice_button *animation_button1 = new choice_button("Firefly",  

choice_button *animation_button2 = new choice_button("MESI",  
```

etc., for each of the available animations. Using this method allows for careful placing of each button but, it requires editing the source code whenever a new animation needs to be added. This results in more complex code, the more animations are added.
Using a text file to store the urls:

Using this method, a text file stores each URL on a separate line. This file is then opened as a QFile, and each line is parsed and turned into a button using the data. This adds complexity to the code by requiring a way to open the file and read each line, and then create the buttons based on the URL. However, it also makes it easier to add new animations as they are made, because it is just adding one line to a text file.

At first the each line of the text was stored as a QString, and the last two elements of the directory were used as the label for the button. This was done by using the QString function “section()” which can divide the QString up based on a delimiter. For example;

```
QString line = "http://www.scss.tcd.ie/Jeremy.Jones/vivio6.0/caches/MESI.vcode";
QString label = line.section("/", -1, -2);
```

The variable label would then have the value “caches/MESI”.

![caches/MESI](image1)

![miscellaneous/newtonrhapson](image2)

Obviously this label was untidy, and would become very long for other animations such as “miscellaneous/newtonrhapson.vcode.” Because of this a better way of assigning labels would have to be used.

As such, the text file format was rewritten to include both a label for the button, and the URL separated by a different delimiter. In this case, a hash symbol, '#'. The lines in the text file then looked like this:

```
MESI#http://www.scss.tcd.ie/Jeremy.Jones/vivio 6.0/caches/MESI.vcode
```

Then using the section() function, “MESI” was stored as the QString label variable and the URL was stored as the QString URL variable.

```
QString line = "MESI#http://www.scss.tcd.ie/Jeremy.Jones/vivio6.0/caches/MESI.vcode";
QString label = line.section("#", 0, 0);
QString URL = line.section("#", 1, 1);
```

This format allowed each button to have its own label, but the buttons couldn't easily be categorised. Simply adding another piece of text indicating the section it belonged to, e.g., sorting algorithm, trees, caches, etc., allows the program to easily parse and categorise them.

So, continuing the above example, each line would look like this:

```
```

From this line, the program can determine the category to which the animation belongs, the label of the button associated with the animation, and the URL of the animation.
For the purposes of allowing comments in the text file, a number was added to the end of each line. If the number was “1”, the line is identified as a animation URL and is processed into a button. If it is anything else it is ignored.

This is valid:
Cache Coherency#MESI#http://www.scss.tcd.ie/Jeremy.Jones/vivio 6.0/caches/MESI.vcode#1

This is ignored:
Cache Coherency#MESI#http://www.scss.tcd.ie/Jeremy.Jones/vivio 6.0/caches/MESI.vcode#0

This is useful for allowing the application to ignore any animations which currently do not work properly for memory reasons or similar. For example, the DLX / MIPS computer architecture animation will not load, but will crash the application. For this reason it's number value is set to “0”, so it will not be turned into a button, but is still in the text file in case a fix is found.

The method of storing the URLs in the text file was chosen because it makes it easier to divide the animations up into sections, as well as making it easy to add new animations as time goes on.

Creating the Buttons

With Qt, buttons are created from the QPushButton class. These buttons are very basic and only store the text which is written on them. For example, if a QPushButton is created labelled “Back,” it is possible to use a function to get that QString value.

```cpp
QPushButton *back = new QPushButton(“Back”);
QString label = back->text();
```

The above code creates a button labelled “Back” and then stores the text value of the QPushButton in the QString variable label.

For the purposes of generating a button for each available Vivio animation, more data needed to be stored in each button. Because of this, a subclass of QPushButton was created. This class, choice_button, only has one additional value to store than the QPushButton class, the URL of the .vcode file.

For example, “http://www.scss.tcd.ie/Jeremy.Jones/vivio 6.0 sorts/BubbleSort.vcode”, is stored as well as text labelling the button as “Bubble Sort”. This allows the button to have a neat label but also allow easy access to the URL.

This button labelled “BubbleSort” also contains the URL for the animation.
Programming the User Touch Interface

Programming the application to recognize the touch inputs and gestures given to it by the user was one of the most important part of the project. Because of the nature of the hardware that Android is typically run on, phones and tablets without keyboards and mouses, it was necessary to implement the controls on the touch screen.

While most tablets and phones have the capability for multi-touch, there are still a few examples, e.g., the Samsung Galaxy Europa phone, which only have a single touch point. Accordingly, the best course of action to take when deciding how to implement the controls was to have the basic functionality of the Vivio player, i.e., starting and stopping, clicking on buttons, speeding up and slowing down, changing the direction of the animation, and ticking through the frames of the animation, mapped to just a single touch point input. Any of the advanced features such as snapping to checkpoints were mapped to a multi touch point input.

As such, each of these controls were implemented using this mind set.

The basic functionality of the Vivio player was mapped to the touch screen as follows:

1) Starting and stopping the animation is controlled by simply touching anywhere on the player widget for one fifth of a second. If the animation must be started by pressing a button, e.g., the cache coherency protocol animations which require one of Read or Write actions to start, the animation must be running in order for these to do anything. The one fifth second delay is needed in order to prevent the player from starting and stopping whenever a button is pressed. This delay is just long enough that a quick press of a button will not cause the animation to start or stop, but just short enough to not be too annoying.

2) Buttons can be clicked by simply touching them. If the button is held for any longer than one fifth of a second, as mentioned above, the animation will start or stop as applicable.

3) While an animation is running, its speed can be controlled by swiping horizontally across the touch screen with one finger. Swiping from left to right increases the speed, while right to left decreases the speed. While an animation is paused, the speed can be changed by swiping vertically instead. Top to bottom to slow down, bottom to top to speed up. The length of the swipe must be further than a given threshold, currently defined by MIN_MOVEMENT to be 15.0 pixels in length, to prevent a touch from being interpreted as a small swipe. This is due to the possibility of slight movement, of one or two pixels for example, between the first touch and the release point.

4) To tick through the frames of the animation, while the animation is paused, swipe horizontally across the touch screen. Left to right ticks forward through the frames, right to left ticks back through the frames. As above there is a threshold for minimum swipe distance to allow for small movement when touching the screen.

5) The direction of the animation can be changed by first pausing the animation, and then swiping in the direction you want the animation to run. Right to left for backwards, left to right for forwards. This action is the reason for the need for the
minimum swipe distance, because, when paused and started again, any difference between the point where the touch began to where it was released would cause an undesired change of direction.

The advanced functionality is mapped to multiple touch points as follows:

1) To snap to the previous checkpoint or to the next checkpoint two touch points must be used. To snap to the previous checkpoint use a pinching gesture, i.e., two touch points moving towards each other, and to snap to the next checkpoint an expanding gesture, i.e., two touch points moving away from each other, must be used. Because of the relatively small amount of checkpoints in an animation compared to frames, there is a certain length that the gesture must be before a checkpoint is snapped to. This is currently defined by INTV_PINCH_EXPAND to be 30 pixels, in the application.

2) The context menu can be brought up by tapping the screen with three fingers at the same time.
Testing / Evaluation Environment

Android Emulator

In order to use the Android emulator, an Android virtual device must be made first. For this reason, several devices were made based on different versions of Android. For example, one device was created which used Android 2.2, and ARMv5, while another used Android 3.1, and ARMv7. Using the Android emulator proved to be time consuming and generally a waste of time for this project because of the amount of time it took to boot up and start running. It took an average of 3 minutes to boot up and for debugging purposes this was far too long. It was quicker to build and compile the application into a .apk file, copy it to an actual device, install it, and run it.

A further disadvantage was the lack of ability to use more than one touch point at a time, which was required during the project to implement some of the more advanced features of Vivio.

Android Tablet

In order to test and evaluate the working of Vivio on an actual Android device, a Samsung Galaxy Tab 10.1 was used. This device uses an nVidia Tegra 2 CPU which uses an ARM7 instruction set.[6] The instruction set is important to note because the code was originally designed for use on PCs, running Windows, through a browser, and these Windows systems typically used IA-32 or x86 instruction sets.

The Samsung Galaxy Tab had 3 touch points, and was running Android version 3.1. Testing on the Tab was a lot easier than using the emulator because of the tablet’s USB debugging mode.

To test the application using this mode was as simple as connecting the tablet to the computer running the Qt Creator, enabling debugging mode on the tablet and running the application. Breakpoints could be set in the IDE and would stop the application at the relevant points. Generally, the program would start running, or would crash immediately giving a pretty clear indication of whether the changes made had worked or not.

![Image of Samsung Galaxy Tab 10.1 using Vivio application]
Chapter 4: Problems Encountered

1. Fatal Error with Ministro 2

When trying to run the application on both the emulator and on the Tablet, there would be an error which said “Your application encountered a fatal error and cannot continue.” This error caused a lot of trouble because it occurred before the program started and as such could not be debugged properly. Upon reading the compiler output,

Ex:
```
> W/System.err( 359): Caused by: java.lang.UnsatisfiedLinkError: Cannot
> load library: link_image[1962]: 33 could not load needed library
> 'libQtDeclarative.so' for 'libt3.so' (load_library[1104]: Library
> 'libQtDeclarative.so' not found)
```

it showed that some of the Qt libraries were not loaded correctly by Ministro. There were various attempts at solving this problem by changing the deployment configuration of the application, i.e., changing how Ministro was loaded onto the device. None of these different configurations worked, but upon uninstalling Ministro from the device and reinstalling Qt Necessitas and reconfiguring the environment, it was discovered that there had just been a version mismatch between libraries used by Qt Creator, Necessitas alpha 3 update 3, and the libraries downloaded by Ministro, Necessitas alpha 3 update 4. Once this issue was solved, the application would start.

2. Touch Events and Mouse Events Overlap

When programming the touch events, the mouse events which were previously implemented in the Vivio player were conflicting with the touch events. For example, when you tapped the screen with your finger, a mouse click event was triggered, and when you removed your finger, a mouse release event was triggered. This caused problems for touch gestures, because the mouse release and mouse click events would trigger every time a gesture was started and finished, causing the Vivio player to start or stop.

One solution was to ignore the mouse click events completely by using:

```
#ifndef VIVIO_APP
#endif
```

around the mouse event functions in the player class. This was not an ideal solution, because the mouse click and release events are important in the functionality of the Vivio player. They are used to not only control the starting and pausing of the animation, but also for using the various buttons which can be included in animations.

Another solution was to ignore the first touch point of the gesture completely and allow the mouse event to take over. To determine if a gesture had been input, the first point of a touch gesture was recorded, and compared to the last point of the gesture. If the two were the same, the animation would start or pause as relevant. If the two points weren't the same, then the release would be ignored.

The final solution, which was ultimately implemented, was to time how long a touch point
was pressed and measure how far it had moved. When released, if the touch point had moved a
certain distance, i.e., it was a gesture, or the touch was shorter than 200ms, then the touch would be
ignored. Otherwise the animation would start or stop as applicable.

3. ) Instruction set / Signed / Unsigned Character

When trying to run a Vivio animation, there would always come a point where a segmentation fault
would occur. The cause of this error was discovered, through use of breakpoints throughout the
functions in the player class, to be occurring in the interpret function of the class. This class was
difficult to debug using breakpoints because of the sheer number of different cases which could be
countered throughout the program. The method to find a solution was to use Qt’s built-in debug
method, “qtDebug()”, to output the current command being interpreted. This gave a very clear
indication of which command in the Vivio program was causing the segmentation fault. The fault
was caused by the different instruction sets used depending on whether it was the PC version or the
Android version of Vivio that was being used. In the case of the PC version, x86, while in the case
of the Android Tablet, ARMv7a. The segmentation fault was caused by the fact that in x86, a
character value is by default signed, while in ARM, its default is unsigned. After locating where the
error took place, all that was required was specifying the signed nature of the character variable.

4. ) Signal Errors

Occasionally, when in the process of debugging using the tablet, there would be Signal errors for
input and output. The problem occurred when the debugging cycle was aborted during an
animation. Basically, whenever the debugging stopped when an animation was playing, and was
restarted again later, signal errors would be thrown and stop the application from running. The
debugging could resume when the error boxes were closed, and would not return until the next lot
of debugging began.

5. ) Random Crashes

After running for a while, the Necessitas version of Qt Creator would stop working, and the only
solution would be to force quit, effectively losing all unsaved progress. This was usually caused by
program running for too long and taking up too much RAM. The solution was to restart the program
periodically, usually after a few rounds of debugging.
Chapter 5: Conclusion

Critiques

At the end of the project, the main goal was reached, i.e., getting Vivio to work on Android. However, there is still a lot of room for improvement to make it into a proper application.

First of all, the user interface is currently very bland and very blue. It is basically just buttons which link to the vcode file which is to be downloaded and run.

Secondly, the touch screen interface, while it does a serviceable job, is very basic. Most of the touches and gestures just behave the same way as if a mouse had been used. Gestures also aren’t featured beyond the basic pinch and expand. Each of the gestures, such as the swipe gesture, would behave a lot better if the speed increase was logarithmic as opposed to linear.

Finally, the animations seem slower and a bit rougher running on Android than they did when running on a PC. This may be due to the reduced power of the Android device, or due to the implementation of Qt on Android using Necessitas.

Future Work

Future work with Vivio could involve a port to the iPhone and iPad. There is currently work under way to port Qt to iOS [5], so this should be possible soon.

Work could be done on improving the speed and smoothness of the animations, because they seem slower and rougher, and some artefacts of previous frames appear and stay on screen. These issues could be dealt with.

Having a way of updating the text document containing the URLs with new animations via RSS is something that would make it very easy for new animations to be added to the application.

Releasing the application in the Android App Store is also something that should be done in the future, in order to get Vivio to as wide an audience as possible.

Conclusion

This report documented the work done on the project “Vivio on Android.” It began with a brief overview of the background of the project, Qt, Android, and Vivio.

It detailed the work that was done to get Vivio working on Android; the design process for the user interface, how the main_menu, section_screen, choice_screen, and player_screen classes work and interact with each other, how the animation URLs are stored and accessed, and how the controls for a Vivio player, without a keyboard and mouse, were designed and implemented through the touch screen. It documented the various problems that were encountered with the environment, with Necessitas, and with Android’s instruction set, what caused the problems, and how these problems were solved.

This project had the requirement of getting Vivio to work on Android and enabling the functionality of the Vivio player through the touch, and in this it succeeded.
Appendix A: Menu Screen Images

Menu Screens

This appendix is for images showing the various menu screen:

The Application starts with the first screen below, the main menu, then moves to the second screen, the section screen where the user chooses the type of animation, which moves onto the choice screen where the user chooses and animation, which moves to the player screen, where the animation is played.

In the example below, the user selects “Let’s Get Started,” then chooses the “Sorts” category, followed by the BubbleSort animation, and then finally watches the animation.
Main Menu

- BubbleSort
- BubbleBackSort
- ExchangeSort
- Insertion Sort
- Quick Sort B
- Quick Sort W
- Shaker Sort
- Shell Sort

Bubble Sort

Comparisons: 0
Exchanges: 0

Click "start" to sort list into ascending order.


      http://www.tbray.org/ongoing/When/201x/2010/11/14/What-Android-Is


[5] Qt iPhone Webpage:

[6] Samsung Galaxy Tab 10.1 Specification:
      http://www.gsmarena.com/samsung_galaxy_tab_10_1_3g-3892.php

[7] Qt Doc References