University of Dublin

TRINITY COLLEGE

FYP - VivioApp

Cian Lafferty
B.A.I. Engineering
Final Year Project April 2012
Supervisor: Dr. Jeremy Jones

School of Computer Science and Statistics

O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

Signed: __________________________  Date: 3/4/2013

Cian Lafferty
Abstract

In this project I have developed VivioApp (a Vivio player), that runs on Android devices and is ready to be compiled for other mobile operation systems. VivioApp takes full advantage of the touch screen capabilities of modern mobile devices. Giving the user an easy to use app with access to the full set of Vivio animations. The app is designed so that further additions to the set of Vivio animations will not require a change to the app or the user to download any updates.
Acknowledgments

Sincere thanks to Dr. Jeremy Jones for his assistance, advice and encouragement throughout this project. Without his guidance this project would not have been a success. I would also like to thank my family and friends for the support I have received throughout my college life.
Table of contents

DECLARATION .................................................................................................................................................. 1
ABSTRACT ....................................................................................................................................................... 2
ACKNOWLEDGMENTS ........................................................................................................................................ 3
TABLE OF CONTENTS ...................................................................................................................................... 4

CHAPTER 1 – INTRODUCTION....................................................................................................................... 6
  1.1 MOTIVATION ........................................................................................................................................... 6
  1.2 OBJECTIVES ......................................................................................................................................... 7
  1.3 OVERVIEW .......................................................................................................................................... 8

CHAPTER 2 – BACKGROUND ......................................................................................................................... 9
  2.1 TABLETS AND SMARTPHONES (STATE OF THE ART) ...................................................................... 9
  2.2 VIVIO ANIMATIONS ....................................................................................................................... 10
  2.3 QT ...................................................................................................................................................... 13
  2.4 NECESSITAS ..................................................................................................................................... 14

CHAPTER 3 – ORIGINAL DESIGN SOLUTIONS .............................................................................................. 15
  3.1 ONSCREEN BUTTONS ...................................................................................................................... 15
  3.2 GESTURES ......................................................................................................................................... 16
  3.3 SELECTING THE ANIMATIONS ...................................................................................................... 17
  3.4 GESTURES AND MENU ............................................................................................................... 18
  3.5 THE DESIGN .................................................................................................................................... 18

CHAPTER 4 – GETTING STARTED ................................................................................................................. 19
  4.1 QT OR NOT QT .................................................................................................................................. 19
  4.2 NECESSITAS PROBLEMS ............................................................................................................. 19
  4.3 QT LEARNING CURVE .................................................................................................................. 20

CHAPTER 5 – IMPLEMENTATION .................................................................................................................. 23
  5.1 PLAYER CONTROL FROM MENU .................................................................................................. 23
  5.2 STATIC MENU ..................................................................................................................................... 25
  5.3 GESTURES ......................................................................................................................................... 26
  5.4 DYNAMIC MENU .......................................................................................................................... 30

CHAPTER 6 – VIVIOAPP THE FINAL RESULT ............................................................................................ 33

CHAPTER 7 – ANALYSIS AND CONCLUSIONS ............................................................................................ 36
  7.1 DOES VIVIOAPP MEET THE OBJECTIVES? ................................................................................ 36
  7.2 CONCLUSIONS ................................................................................................................................ 37
CHAPTER 8 – RECOMMENDATIONS FOR FUTURE WORK

8.1 DEPLOYMENT TESTS

8.2 USABILITY TESTS

8.3 LOOK OF VIVIOAPP

8.4 BUG FIXES AND ADDITIONAL FEATURES

REFERENCES

APPENDIX A – VIVIOIDE CODE

A.1 VIVIO SOURCE CODE

A.2 VCODE

APPENDIX B – DLX ANIMATION

APPENDIX C – QT PRACTICE CODE

APPENDIX D – HTML SOURCE CODE

APPENDIX E – MENU OPTION CODE

APPENDIX F – PROJECT CODE
Chapter 1 – Introduction

1.1 Motivation

Over the last 2-3 years students have progressed from having Laptops to having Smartphones and/or tablets. Many students use these highly portable devices to check emails, surf the web, check their timetables and download their course lecture notes and materials from Blackboard (TCD’s lecturer to student information system). A student with a smart phone can now access all the material needed to keep up with their courses. The student could, of course, access these resources, using a laptop but laptops are big and heavy compared with a smartphone or even a tablet. Unfortunately these devices have some limitations as their browsers don’t accommodate plugins such as Vivio.

Vivio is a player that plays Vivio animations a set of animations that have the ability to play backwards and forwards equally smoothly. These animations run programmatically and can have buttons that when pressed change the execution of the animation. Vivio gives the viewer a high level of control over the animation e.g. the animation can be single stepped through the animation letting the user absorb the full complexity of the problem they are viewing allowing a greater understanding of complex ideas.

The motivation for this project comes from the desire to be able to leave my laptop at home and still have the ability to access everything that I could if I had brought it.
1.2 Objectives

The objective of this project is to create an application that plays Vivio animations on smartphones and tablets. The app is primarily intended for use on Android devices but it should be developed in such a way that accommodates the possibility of being run on other touch screen operating systems e.g. IOS and windows phones. The app must be user-friendly and easy to use. Full control of the animations functions must be accessible to the user with similar ease as using the mouse with the desktop version of Vivio. A key aspect of this project is flexibility. The app needs to be designed in such a way that it can be modified so that it can access any animation/ group of animations.

VivioApp will be:

- Vivio on Android – the primary objective of this project is getting Vivio working on android. It needs to have all the functionality of any other player in the Vivio set. This means playing forwards/backwards, the ability to change the speed and to snap/play to a checkpoint. Another key function of the Vivio set is the ability to single step through the animations.
- Platform independent – although my project will focus on Android the VivioApp should be able to run on any touch screen device. This will require the app to be compiled for each platform, I will not do this but the source code should not need to be edited.
- User-friendly – Vivio can be easily controlled by a mouse. The final result should be just as easy to control from the touch screen. The app should not have a complex menu system. It should be easy to navigate for all users.
- Flexible – platform independence adds flexibility as it doesn’t limit the app to Android devices. The app should work independent of the screen size also. This will require the ability to zoom. The final app should be flexible so the addition of new animations does not require a change to the app. Another requirement is that the app needs to be flexible in that it shouldn’t require that all animations are in the same place. For example if another lecture decides that he/she wants to have a set of animations for his/her course and do not want to include them with all the other animations, he/she wants a new app for just his/her course, it should only take a change to one line of code to build the new app.
1.3 Overview

- Chapter 1 introduces the project to the reader, details the reasons and motivations for the project, goes through the goals of the project and gives a brief overview of each chapter.
- Chapter 2 provides a discussion of the current migration from laptops to tablets and phones. It also provides a brief history of Vivio animations, the Vivio set of programs, Qt and Necessitas.
- Chapter 3 explores the design of the project and the thought process behind the decisions that resulted in the final design.
- Chapter 4 discusses the reasons for selecting Qt and the initial stages of setting up Necessitas and learning Qt are discussed.
- Chapter 5 explores in detail the implementations of all the features of VivioApp.
- Chapter 6 looks at the final result of this project, VivioApp.
- Chapter 7 presents the final analysis and conclusions.
- Chapter 8 provides some suggestions for future work on VivioApp, such as independent user reviews and tests.
Chapter 2 – Background

2.1 Tablets and smartphones (state of the art)

An ever increasing number of Irish people are buying and using smart devices (smartphones and tables). All my friends and people, with smartphones, who I’ve asked say that they cannot understand how they lived without them. The purchase of a smart device is therefore, a trap door action. This is the major cause of continued expansion of the smart device market.

“Research conducted by RED C on the RED Express in early January 2013 outlined high incidences of smartphone ownership among the Irish population, with 46% claiming to own a smartphone while 23% claim to own a tablet.” [1]

This has increased from figures from 2011.

“In 2011, 80% of the Irish population owned a laptop, while 35% owned a smartphone and 10% owned a tablet device such as an iPad.” [2]

A majority of smart device users are students. Almost 80% of my college friends have a smartphone or tablet [*]. While other users mainly use their devices for browsing the web, looking at emails, using social networks, watching YouTube and reading articles, students use their devices to keep up with college workloads on the go, checking timetables, downloading lecture notes to be read in/out of class. This can be very useful for asking questions at the end of class, when referencing a slide you can now simple show the slide in question to the lecture.

For an increasing number of students leaving your bulky laptop at home and bringing your smartphone device to college is the norm. The portability of smart devices is the major advantage over laptops. Laptops also have, depending on the model, a rather long starting up and closing down time. It is rather irksome than having to wait for your laptop to shut down while trying to rush to your next class. Another advantage that smart devices have over laptops is battery life. But there is a catch, processing power. Figure 1 below shows two of the Vivio benchmark animations run on the tablet verses on my laptop. On the next page are the specifications of the two processors.
## 2.2 Vivio animations

“Vivio is a system that makes it easier to create interactive reversible 2D vector based e-learning animations for the WWW. Vivio animations describe how the properties of graphical objects change as function of time or as the result of external events. Since Vivio animations follow the execution of a program, they can respond to user input and are, consequently, not limited to preset animation sequence.” [3]

Above is a quote from Dr. Jeremy Jones and explains exactly what Vivio is. This can be summed up in even fewer words as “Interactive reversible 2D vector based e-learning animations”.

Interactive: Vivio animations can contain buttons that affect the flow of the animation.

Reversible: they can run backwards and forwards with the same performance.

2D: the animations are in 2D.

Vector based: this means that the animation can be scaled bigger/smaller without loss of detail or pixilation. Vivio animations can be resized to and size only limited by screen size and your ability to see very small objects.

E-learning: designed for educational purposes, to be viewed on electronic devices.

Soon it will be possible to view these animations on any electronic device with the appropriate software.
To understand Vivio I must first explain some terminology, Vivio animation an animation that runs on a Vivio player. The Vivio animation file type is “.vcode”.

There are three players in the Vivio set one of these must be downloaded from this link [4] in order to run .vcode files.

Vivio: the program that runs on desktops/laptops, it can run any .vcode files on your desktop/laptop.

VivioNP: the Vivio Netscape Plugin, is a plugin for browsers and can run any .vcode file on the web.

VivioApp: Vivio for Android, it is a Vivio player that has a menu for selecting an animation.

For developing Vivio animations there is an Integrated Development Environment, VivioIDE.

The VivioIDE allows a programmer to create smooth and complex animations with an ease that eludes other animation languages. Take for example translating an object across the screen from A to B over time let’s say 5 seconds and we want this to look like the object is moving across the screen smoothly. We would need the animation to refresh 60 times a second. In most programming languages this would be very complex, we would have to divide the distance from A to B by 300 and then create a loop that moves the object this distance and then make sure that the program executes this loop 60 times a second. You can do this in a few lines with the VivioIDE.

```
setTPS(60);
object.translate(Bx-Ax, By-Ay, 300, α, β);
```

Where α{l≥1} is the number of ticks between each step, and β{0,1} is if you want the program to wait for the function to finish. Set to 1 when you want the program/animation to move the object and then execute the next line of code. Vivio runs on ticks everything that happens takes one or more ticks in the example above the translation will take 300 ticks. setTPS(int ); sets the ticks per second(TPS) so in this case we want the object to move from A to B in 5 seconds so the TPS are set to 60. This means the animation will refresh 60 times per second.

As research for this project I developed a very simple animation that moved a rectangle around the screen in a loop and changed the colour of the rectangle simultaneously the code in included in [Appendix A.1]. Below is a sample showing how simple it was, at the time it seemed complex but after becoming familiar with VivioIDE and how to write Vivio source code it became simple.

```
setViewport(0,0,WIDTH,HEIGHT,1);
setTPS(10);
start();
RecA = Rectangle2(0, 0, 0, redbrush, 0, 0, 50, 50);

while(1)
{
    RecA.setOpacity(255);
}
RecA.translate(250, 0, 30, 1, 0);
RecA.setOpacity(0, 30, 1, 1);
RecA.setBrush(bluebrush);

RecA.setBrush(redbrush);

Let me explain how this code gets from this point in the VivioIDE to one of the Vivio players. Figure 2 is a diagram of the process.

Figure 2

The Vivio source code [Appendix A.1] gets compiled and compressed into vcode. [Appendix A.2] Shows the vcode compiled from the Vivio source code. This file can be stored in a server and can then be accessed from any internet connection provided you know the link and that link is visible to your connection. Path one above shows how the vcode is stored on the developers file system and then run in Vivio from the VivioIDE. In path two the file is uploaded to a server (e.g. a TCD server) and is downloaded via a browser to a user’s file system and then played with Vivio. Path three shows how the VivioNP is used with a user’s browser to play the animation the fourth and final path shows a user playing the animation on a tablet using VivioApp.

Vcode files are usually very small even for complex animation like the dlx animation[Appendix B]. This means that it is a viable option to just download the file whenever you want to play the
animation and discard the file when finished. In most cases the file is so small that it will still be in the cache when you swap between two animations and if not downloading it again is so fast it’s rarely noticeable.

2.3 QT

"Qt ("cute") is a leading cross-platform application and UI development framework for leading desktop, embedded and mobile operating systems. Qt uses standard C++ and is widely used for developing software applications with a GUI and also for developing non-GUI applications with features such as file handling, database access, XML parsing, thread management and network support.” [5]

As stated in the above quote Qt is a cross-platform application and UI development framework that allows software and programs written in Qt to be deployed across a range of platforms if compiled for each platform, with the same source code. Qt is supported by many platforms including Windows, Linux, Mac OS and Symbian, other external ports of Qt have been developed by programmers that are not part of the Official Qt team. These ports include Android, Haiku and Palm Pre. Qt for iOS is also in development.

In most cases Qt applications are indistinguishable from applications written specifically for each platform as they run natively there is also little or no difference in performance of the application.

Qt is written in C++ and uses standard C++ with a moc (Meta Object Compiler) code generator to add Qt classes. There are over 800 classes in Qt. All Qt classes are based on QObject this means that all the classes are compatible with each other as each class can be viewed as its basic form, as a QObject, allowing for one of the most useful aspects of Qt, slots and signals. I discuss this in more detail on page 20.

Qt is mainly used for GUIs. In Qt GUIs are made up of widgets (windows gadget) and widgets within widgets. Complex GUIs can be created from simple widgets such text boxes, buttons and labels.

Qt supports other features such as OpenGL, SQL, XML and CSS.

Qt was created by Haavard Nord and Eirik Chambe-Eng who founded Quasar Technologies later known as Trolltech. Trolltech was bought by Nokia and Qt was ported to Symbian. When Nokia decided to go with windows 7 phone as the operating system for their smartphones, the Symbian OS was essentially a dead platform. For some reason Nokia decided not to port Qt to windows phones so Qt was sold to Digia.
2.4 Necessitas

Necessitas was developed by BogDan Vatra as a hobby project, using Qt 4.8, which he started in late 2010. His aim was to have a Qt port to android. He developed an IDE using Qt creator. Necessitas builds a Java shell around the Qt application and creates a .apk file that can be run on android. Necessitas is still in beta but will soon become part of the Qt Project.

*Together with BogDan Vatra we have agreed that he will contribute his work into the Qt Project. The Qt 5 Android port will be based on the work done in the Necessitas project and BogDan will also continue his work in this area within the Qt Project.* [6]

In order to get Necessitas working you need to have several components on your machine;

- Necessitas for Qt creator
- Android SDK and Android NDK
- Apache Ant
- Java JDK and JRE
- Qt 4.8.2

You also need to download Ministro to the device that you intend to run the application on, this is simplified by the fact that when you run a Qt app for the first time it will search for Ministro on the device and if it’s not found it will ask the user if they want to download Ministro, answering yes downloads Ministro.

*Ministro is an Android application which provides a system wide downloader and installer of the LGPL Qt shared libraries. You can find Ministro on the Android Market* [7]

Downloading the necessary files should have been made easier by the Necessitas installer [8]. But as I explain on page 19 this is not the case.
Chapter 3 – Original design solutions

3.1 Onscreen buttons

The original idea was to be able to control the app with very little movement just like with a mouse your hand shouldn’t need to move more than 100mm to achieve what you desire. Scroll the mouse wheel, left click or right click for a menu and select the option you want. These things are very easy and almost second nature to all computer users. Without a mouse what can you do easily? Buttons are easy to press, every user in the world can press a button. How many buttons are on a typical tablet? Power/lock button, 2 volume buttons, back, home, Recent Apps, menu and screen shot. All of these have uses that shouldn’t be overwritten by an application. The lock function is one of the most important functions of a tablet/smartphone when the user wants to turn off the screen without leaving the application. Although the animations do not use sound and therefore the volume buttons could be used to control functions of applications they are usually not in an easy location for the user to press frequently as usually the user would only set the volume once or twice an hour of device usage[*]. The back button is used by many applications for functionality generally to return to the previous menu. In my opinion the back button should almost always exit the application it is very frustrating to close a browser as the back button will bring you all the way back through the pages in your session history before finally closing the app. I am going to be developing this app for Android devices and these devices keep the closed program in memory so that it can be reopened quickly. If memory gets full the device removes data on a least recently used basis and so there are no memory overflow problems. So if you accidently close an app you can reopen it as if you had it open all the time and just jumped to the home screen. With this knowledge and my own opinion I have decided that when pressed the back button will exit my app. The home button does exactly what you would expect it goes to the home screen, the app is not closed it is just in the background, selecting another app will open that app and the pervious app will be added to a list of running programs. The Recent Apps button brings up a list of active apps and most recently closed apps. My application will not affect the functionality of either of these buttons or the screen shot button. So the only button left is the menu button which is there to be used by applications. Several apps do not use the button and the better ones hide it from view if it is not used, unfortunately not all apps hide it when not used meaning that some users don’t realise that it is useful and therefore are not in the habit of pressing it. This is the case on tablets but on smartphones due to the lack of onscreen space it is very useful and most users use it and is very easy to press.
So I have one button that I can use in my app. One of the alternatives for this is to have on screen buttons. On screen buttons have the advantage of being easy to see and the user notices them, the major flaw is that they take up space on screen and as you can see below in figure 3 this is not an option as the already reduced screen space compared with a laptop screen is reduced further so a point that on some phones the animation would have very little space. Figure 3 shows the reduction in space from the screen size on my laptop to the size of the screen on the tablet and a phone. On screen buttons would mean that in most cases it would not be possible to see any detail within the animation.

Figure 3

### 3.2 Gestures

As all Android devices are touch screen and on screen buttons are not an option as mentioned above touch gestures are the only alternative. The question then becomes which gestures should be used for each function of the app. There are many gestures and combinations of gesture that could be used. Android devices can detect a high number of individual touch points on the screen at one time. I tested this on the device that I was given access to by Jeremy Jones, the results suggested that there was a limit to the number of touch points close to 8. Further tests showed that in the case of 1-3 touches the points were always detected but the reliability of the detection of touch points greater than 4 was inversely proportional to the number of points.
3.3 Selecting the animations

The reason for this project is the fact that Android browsers don’t support plugins so, without a browser, how does the user choose the animation that they want to view? One option is to have a different app for each animation. This is a simple solution and does comply with one of the objectives of the project to have the ability to add future animations. All that would be needed is for the app to be rebuilt changing the source of the animation and creating the .apk file for users to download. In my opinion this solution conflicts with one of objectives of this project, to be user friendly. If the user has to download a new app for every animation then their home screen or app menu screen will be filled with several versions of the same app the only difference being the animation. The change would probably only need a few lines of code. This repetition is very wasteful and in my opinion this should be avoided.

Therefore the app must have knowledge of all the animations from one download, but now we still have the question of how to select the animation, having a gesture for each animation would be very complex both for the developer, me, and the user. I would have to find a new gesture for each animation these gestures would all need to be different so that they could be detected correctly. Another point is that the user would need to know what gesture selects which animation. Alternately there could be two gestures one to go to the next gesture and one to go to the previous gesture. The app would have a list of animations and to fulfil all of the requirements of the objectives the list could just be updated or a new app with a different list could be created. This solution is not optimal because if the user knows what animation he/she wants to view and this animation is in the middle of the list then he/she has to go through all of the animations until getting to the desired animation for lists longer than 5 animations this would become frustrating. The ability for the user to edit the list may solve this problem but it is not optimal.

The best solution is to have a menu of all the animations for the user to choose from. How does the user access this menu? A button of course, I’ve discussed earlier about having onscreen buttons and even thought this would only be a single button on the screen and it would not take up a lot of space it would still take some space away from the animation meaning that every animation would need to leave the same location free of information otherwise it would be lost to the user as most of the animations have already been created this could mean a reduction in quality of some of the animations so again I have decided against any onscreen buttons. Luckily there is one button that can be used the menu button, users would understand what it does as it does what you’d expect it displays the menu. So the menu would contain all of the animations possible, but what if a new animation was to be added to the menu, a developer would need to change the menu and create a
new app or an update to the original app and users would have to download the new version. This would be ok if a completely new set of animations was added to the Vivio collection but for a single addition it would be a pain for the developer and the users. So a dynamically constructed menu is the only option. The menu would be created each time the app opened and find all the animations needed for the app. This would be done by looking in an online directory and selecting the animations from there and adding them to the menu. This would give the option for a lecturer to create a directory and an app that looks in that directory, the lecturer could have an app for all the courses they use animations for or an app for each course. To update any set of animations the Vivio file would just be added to the directory. The app would only ever have to be downloaded once with no updates needed.

3.4 Gestures and menu

Depending on how frequently you use the app you may forget some of the features and gestures. The easy to use gestures that are used in all applications to perform similar actions will never be forgotten but other gestures that are used differently in each app can be. Therefore a way for the user to use all of the functions available to the application must be provided to the user. In the original Vivio this is done with a context menu that appears when a user right clicks the screen, on a tablet there is no mouse and so right clicks are not possible, this is also a problem for porting to Mac OS but that is outside the scope of this project. The menu button is already been used for selecting the animation so what are the other options?

The only option is to use the menu button and add an option to go to the animation selection menu, as we have previously ruled out the possibility of having onscreen buttons. A gesture to bring up the menu could also work but as we are providing an alternative way of controlling the animation, without gestures then using a gesture to bring up the menu would be silly as the user my forget this gesture.

3.5 The design

The final design therefore is to have a player that can be controlled with either gestures or by selecting the option from a menu. The menu appears after the menu button is pressed and disappears when an option is selected. An option to go to the animation selection menu is available from the menu. The selection menu will be dynamically construed on start-up of the app and contains all of the animations in the directory chosen by the app creator. Pressing the menu button while in either menu will return you to the animation screen. This design fulfils all of the objectives of the project.
Chapter 4 – Getting started

4.1 Qt or not Qt

Qt is not the only cross platform framework. The question is if it is the right framework to use for this project. When I started the project it seemed as if Qt was the only viable option. The project already used Qt for the player and other classes that are the core of the project. At the time I had a quick look at other options. When choosing a cross platform frameworks there are a few things to take into consideration.

- Abilities of the framework to achieve what you are intending to do. Newer frameworks tend to have limited capabilities in comparison to frameworks that have been around for years. I found some options, Titanium [9], Xamarin [10] or PhoneGap [11].
- The community size. The amount of developers using a framework is a good indication of how good that framework is. Frameworks need to be tested by very large numbers of programmers to find all the bugs. Learning how to use the framework is also dependent on the community size, more programmers that understand the framework means better tutorials, more programmers learning to use the framework means more tutorials. All the above frameworks have large communities including Qt, all of similar sizes.
- What the framework is most suited for. Some frameworks work best for games other for web applications and other are focused on mobile apps such as PhoneGap.
- The last and maybe the most important is what language the framework is based on. Using a framework based on a language that you are not familiar with would certainly be a greater challenge but making things harder for yourself is not advisable.

Titanium is written in JavaScript and Xamarin is written is C#. Qt is written in C++ and this is my preferred language. For these reasons Qt is the best choice for this project.

4.2 Necessitas problems

Steps for downloading Necessitas, 1 go to the web site and download the installer, 2 run the installer, 3 run Necessitas. If only it was that simple. Although the installer downloads most of the necessary files it doesn’t download the java JDK or JRE as I had a version of these installed on my system I thought it would be fine. The problem is that even though Necessitas doesn’t have what it needs to work, Qt creator runs. This means that it is very difficult to find the cause of the problem. There were many problems with my installation of Necessitas most of which were caused by my lack of experience with the software and compounded by the lack of information for Necessitas on
windows. It took me 2-3 weeks to finally get the basic version of VivioApp running on the device, there were no problems in connecting the device and running once I had solved the initial problems.

I’m not sure I what order I found the solutions to the problems, and discussing how I found each problem would be a waste of time. However I will list the solutions as a checklist for things you need to have in order to get Necessitas running.

- Download the latest version of the java JDK and JRE
- Download the latest version of Apache Ant, although the installer does download a version it doesn’t work.
- Add ANT_HOME and JAVA_HOME to your system’s environmental variables.
- Select the same Kit that you used in setting up the project in Qt creator in the build&run kits
- Make sure that you have a compiler selected from the dropdown menu in compiler.
- If you are using source code for your project in Qt creator right click on the project to add them.

Make sure that you build the project in the same directory as the source code for example if your source code is in ..vivio6.0/src/shared/ use ..vivio6.0/src/ as the directory for your project.
- Don’t have any spaces in file path names

4.3 Qt learning curve

As Qt is at the core of this project it was vital that I understand how it works and how to program in Qt. After getting Necessitas to work I developed a test application, a simple app with a QLineEdit and a QSlider. The task was to build the app in such a way that when the either the slider or the line edit are changed the other also changes. The sliders range was set to the default which is 0-99. Typing a number in the line edit changes the slider to that number and the reverse is also true. This was a good challenge to start because the signals and slots of the QLineEdit and the QSlider do not match.

Signals and slots: slots are functions that can be called in the same manner as regular functions but can also be connected to one or more signals. If any of the signals are emitted then the function is called. The slots and signals must be declared in the class header file.

```c++
signals:
    void name(Type);
    void name(Type, Type...);
{Type = void, bool, int, double , QString...}

public slots:
    Type name(Type);
    Type name(Type, Type...);
{Type = void, bool, int, double , QString...}
```
Signals can be connected to signals and to slots, but two slots cannot be connected to each other.

Below are a few examples of how to connect the lines with the strikethrough are not allowed and do not work in Qt.

```cpp
connect(objectA, SIGNAL(nameA()), objectB, SLOT(nameB()));
connect(objectA, SIGNAL(nameA(TypeA)), objectB, SLOT(nameB(TypeA)));
connect(objectA, SIGNAL(nameA(TypeA, TypeB)), objectB, SLOT(nameB(TypeA, TypeB)));
connect(objectA, SIGNAL(nameA(TypeA)), objectA, SLOT(nameB(TypeA)));
connect(objectA, SIGNAL(nameA(TypeA)), objectB, SIGNAL(nameB(TypeA)));
connect(objectA, SLOT(nameA(TypeA)), objectB, SLOT(nameA(TypeA)));
```

As you can see from the above code, you can connect a signal from one object to the slot of the other object, the signals and slots can have parameters. These parameters must be of the same type. An object’s signal can be connected to one of its slots. Signals can be connected to signals. If the signal has more parameters than the slot/signal it is connected to then the extra parameters are ignored. You cannot connect a signal with less parameters than the slot/signal it is connecting to. A slot cannot be connected to a slot.

For a very detailed description on the internal workings of slots and signals read this article[12].

When working with a Qt class the first thing a programmer should do is read the Qt documentation on that class. In the Qt Class Reference for QLineEdit[13]and QSlider[14]. You can see that the signal QLineEdit::textEdited ( const QString & text ) emits a QString and slot QSlider::setValue ( int ) takes an int. QSlider inherits setValue from QAbstractSlider[15]. As shown previously a signal with one type cannot connect to a slot with a different type. The solution is to create a new class and derive it from QSlider. In most Qt examples the new function is called My<class> so I created MySlider derived from QSlider.

```cpp
class MySlider : public QSlider{
    Q_OBJECT
public:
    explicit MySlider(QWidget *parent = 0);

signals:
    void valueChanged(const QString);
public slots:
    void setValue(QString);
    void send(int);
};
```
The new slider class can now take a QString to set the value and can emit a valueChanged signal with a QString. The code for this is in [Appendix C]. I connected the textEdited(const QString & text) to setValue(QString) which converts the QString to an int and calls setValue(int). This means that when the QLineEdit’s text is changed the MySlider changes its value. Then I connected the valueChanged(int value) signal inherited from QAbstractSlider to the send(int) slot I created. This slot converts the int to a string and then emits valueChanged(const QString) signal. Connecting valueChanged(const QString) to the QLineEdit’s slot setText(const QString &). I used the designer tool in Qt Creator to set up the connections. Figure 4 shows the connection interface.

<table>
<thead>
<tr>
<th>Sender</th>
<th>Signal</th>
<th>Receiver</th>
<th>Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>valueChanged(int)</td>
<td>ms</td>
<td>send(int)</td>
</tr>
<tr>
<td>ms</td>
<td>valueChanged(QString)</td>
<td>le</td>
<td>setText(QString)</td>
</tr>
<tr>
<td>le</td>
<td>textEdited(QString)</td>
<td>ms</td>
<td>setValue(QString)</td>
</tr>
</tbody>
</table>

Figure 4

I learned a lot about slots and signals in a short time using the designer tool, but I still needed to understand how to do it by code as that is how I would be creating VivioApp. With the lessons from this example I used online information and tutorials to try and learn the rest of the information I would need for the project, however this was slow and I didn’t learn enough for the time I spent researching. Online information is not simple to understand and follow. This book [16] helped me beat the learning curve of Qt and allowed me to continue with the project. For anyone wanting to learn Qt I highly recommend.
Chapter 5 – Implementation

5.1 Player control from menu

After becoming familiar with Qt I began developing the app. The main part of the app is the player. I used the same player that is used in the original Vivio player. Adding the player is very simple, just create and object in main.cpp and loading an animation is just as straightforward. Using the code below and building the project with Necessitas produces an app with the player as its main window.

```cpp
#include "stdafx.h"
#include <QtGui/QApplication>
#include "player.h"

QApplication *vApp;

int main(int argc, char *argv[])
{
    vApp = new QApplication(argc, argv);
    Player *vplayer = new Player();
    vplayer->load("https://www.scss.tcd.ie/Jeremy.Jones/vivio6.0/sorts/BubbleSort.vcode", "", 0);
    vplayer->showFullScreen();
    int r = vApp->exec();
    delete vApp;
    return r;
}
```

The resulting app will load and display the BubbleSort animation this player would function in the same manner that the VivioNP functions if a mouse was connected to the tablet but as that is not possible more code is needed. The only function that does work is play/stop as touching the screen represents a mouse click and so touching the screen with any number of fingers will trigger the player to toggle the animation from playing to stopped. This resulted in a problem when it came to implementing gestures. Thankfully at this stage there were no problems with the player, but I needed to plan in what order I should proceed with the development of the app. Control of the player that loads and plays the animations (vplayer). Gestures in Qt are not straightforward and are a challenge to developers with far more experience than I had at that point so controlling vplayer from a menu (popupmenu) was the first step. Then selecting a menu for selecting the animations (metromenu), gestures and finally dynamic building of metromenu.

I set about enabling the control of the player from the popupmenu, in order to control the vplayer I created a new class and sub classed the vplayer in Qt fashion I call this new class MyPlayer.
class MyPlayer : public Player

This allowed the MyPlayer class to access the private functions of the vplayer but as the app will have three windows or widgets (vplayer, popupmenu and metromenu) I needed to create a class to control the switching between the widgets. I decided to sub class QStackedWidget.

class MyPlayer : public QStackedWidget

With the QStackedWidget you can have as many widgets in the class and switching between them hides the other widgets and displays the selected widget. I then added the three widgets to the MyPlayer class but this meant that the MyPlayer no longer had access to the private functions of the vplayer. As a workaround I just changed the source code so that the functions I needed were public I had planned to find a better way of accessing the vplayer’s private functions, what I should have done was to add the MyPlayer as a friend class of the vplayer but at the time I didn’t realise how much I would have to add/edit the vplayer class source.

Player *vplayer;
PopUpMenu *FUM;
Metroselectfile *MSF;

Above you can see that the MyPlayer contains the three pointers one to each of the widgets that will be created in the constructor. I can then connect signals from one widget to slots in the other widgets. Adding buttons to the popupmenu and then connecting signal generated when clicked to the correct slot of the vplayer gives functionality to the app. Connecting the same signal to the MyPlayer’s slot that shows the vplayer produces the effect of hiding the popupmenu when a button is clicked.

QPushButton *button2 = new QPushButton("Play forward");
connect(button2,SIGNAL(clicked()),this,SLOT(playF()));

The above code is taken from the popupmenu constructor, combined with the code below from the MyPlayer constructor shows how the buttons in the popupmenu are connected to the vplayer.

connect(FUM,SIGNAL(playF()),vplayer,SLOT(contextPlayForwards()));
connect(FUM,SIGNAL(playF()),this,SLOT(showplayer()));

All the buttons in the popupmenu are created in the same way. The result is the menu shown in figure 5.
5.2 Static menu

The menu above is plain and simple and because it is for control of the player simple makes sense but the menu for selecting the animation shouldn’t be just a plain old list. Taking inspiration from the Android app screen I wanted to create a similar metro style menu for the animations. I first designed a button label combo widget. The idea was to have an image of the animation or an image representing the animation on the button and the name of the animation on the label. At this point I was just trying to get a simple version of the menu so the buttons were just normal buttons with the name in the middle. Experimenting with the size and shape I found a solution I was happy with, the buttons are not too small so they can’t be seen nor to big so they take up too much space. I added all of the cache animations as well as the dlx and BubbleSort animations to the menu. This was done statically. Figure 6 is an image of this temporary menu, as you can see the buttons do not have an image on them I decide that it would be best to leave that until I implement the dynamic version of the menu.
5.3 Gestures

This was always going to be the hardest part of the project. There are 5 standard gestures in Qt: pan, pinch, swipe, tap and tapandhold. I tried testing the first three gestures, I also already knew that tap worked as my menus were working. I read through the Qt documentation on gestures and followed the guidelines on how to use the gestures. I wrote the code and tested it using the qDebug() function that writes lines to the debugger. I found that the pan and pinch both worked but that the swipe gesture did not. Both a pinch and pan was triggered when the user puts two fingers on the screen this means that panning and pinching simultaneously is possible. I looked at one app that had these features from the point of view of the user. One such app is the Navigation app already installed on my device. I found that panning and pinching simultaneously worked really smoothly and was very user friendly. I decided that this was a feature that I wanted to replicate in my project.

Reading more information on how to add new gestures and reading online forums I was able to implement a swipe gesture that uses the fact that when a user presses the screen a mouse down event is triggered and when the user takes away their finger a mouse up event is triggered. In order to detect a gesture you must create your own gesture recognizer class derived from QGestureRecognizer. The gesture recognizer looks at the events generated and creates a gesture if
the appropriate event is triggered, the gesture recognizer follows the gesture through its life cycle and can cancel the gesture if the correct sequence of events do not happen. So when the mouse down event is triggered a new gesture is created and when the mouse up event is triggered the recognizer looks at the start and end points of the gesture and if they are not far enough apart the gesture is cancelled.

So I now had pinch, pan and my version of the swipe been recognized. This is not enough gestures to control the vplayer I would need more gestures. As the mouse down event is triggered when any number of fingers touches the screen a pan/pinch will trigger a pan, pinch and swipe. I did not want this to happen so I had to change the recognizer to use touch inputs rather than mouse events as its input. I spent many hours researching and testing solutions for different touch screen problems to get a better understanding on how touch works. There was not a solution that worked the way I wanted so in the end I adapted the code for recognizing the swipe gesture to use touch input. My solution was far simpler than any of the examples I followed but it works, the code can be found in [Appendix F]. Not only did I need to create a gesture recognizer for each gesture but I had to create a gesture to go with them. Another key point is to allow the widget to receive touch events.

```
this->setAttribute(Qt::WA_AcceptTouchEvents);
```

The gesture recognizer needs to be initialised and the widget needs to grab the gesture type. The registerRecognizer function returns an enum value for the new gesture type.

```
Swipe1GestureType = QGestureRecognizer::registerRecognizer(new
Swipe1GestureRecognizer());
grabGesture(Swipe1GestureType);
```

My first swipe gesture recognizer used the QSwipeGesture and changed the values of its start and end points, but in order to have an option for a swipe gesture that could be either short (30 – 200 pixels) or long (200+ pixels), these numbers were chosen through trial and error, I had to create a swipe gesture derived from QGesture. I also tried just using QGesture as the gesture for my gesture recognizer but Qt wouldn’t allow this.

A slight adaptation to this swipe gave me a three fingered swipe gesture I now had a pan, pinch, swipe1(long or short) and swipe3 as my gestures. I thought of many other gestures for example holding one finger on the screen and taping a second finger off the screen, but these became too complex for the user to remember and for the programmer to implement. When it comes to user interfaces a rule of thumb is the simpler the better. So I didn’t add any more gestures.
Creating gestures and recognizing them is one part of using gestures the other is getting them to affect the program. This was straightforward enough for some of the gestures but for others it was very difficult to get the program to react to the gesture in a smooth manner. The basic concept is to recognise the gesture event and then use functions to deal with them, below is the code for handling the gesture events.

```cpp
if (QGesture *swipe = event->gesture(Swipe1GestureType))
    swipe1Triggered(static_cast<Swipe1Gesture *>(swipe));
if (QGesture *swipe3 = event->gesture(Swipe3GestureType))
    swipe3Triggered(static_cast<Swipe3Gesture *>(swipe3));
if (QGesture *pan = event->gesture(Qt::PanGesture))
    panTriggered(static_cast<QPanGesture *>(pan));
if (QGesture *pinch = event->gesture(Qt::PinchGesture))
    pinchTriggered(static_cast<QPinchGesture *>(pinch));
if (QGesture *tapandhold = event->gesture(Qt::TapAndHoldGesture))
    tapandholdTriggered(static_cast<QTapAndHoldGesture *>(tapandhold));
```

As you can see the enum value gesture type is used to determine what gesture event has been triggered. Below I have shown the simplest example of a function dealing with the gestures. The Swipe3Gesture is the simplest gesture it goes right or left so to determine what action to take we just find the direction.

```cpp
void MyPlayer::swipe3Triggered(Swipe3Gesture *swipe)
{
    if (swipe->state() == Qt::GestureFinished)
    {
        if (swipe->horizontalDirection() == Swipe3Gesture::Left)
            vplayer->contextSnapToPrevious();
        else if (swipe->horizontalDirection() == Swipe3Gesture::Right)
            vplayer->contextSnapToNext();
    }
}
```

One important note to make is that in most cases you should wait until the gesture is finished before acting on it.

The most difficult gestures to deal with were the pinch and pan. Pinch is used to zoom in and out, when a pinch is triggered it has a scale factor, this is the distance between the two touch points, and is set to 1 when the user touches the screen. If the user moves their fingers further apart the scale factor increases. The scale factor decreases when the user moves their fingers closer together than the original distance. The problem is that this change is very small the scale factor is very rarely greater than 1.14 or less than 0.95. Using the scale factor to directly affect the size of the viewport was not an option as I initialise a variable (zoomfactor) and increase or decrease this value depending on the scale factor of the pinch gesture. I also decided that the animation would never be smaller than the screen. The difficulty with the pan came from trying to make sure that the entire
screen was covered by the animation. Limiting the offset needed to be dependent on the zoom factor became very difficult until I had implemented the pinch gesture correctly.

Zooming had a very funny consequence because of the way the animations are rendered. Vivio animations only render the part of the animation that has changed. The solution is to call the RenderAll() function on the vplayer after setting the viewpoint.

A problem with all the gestures was that when the user touched the screen with any number of fingers the program registered a mouse down event. This caused the player to stop/start. This made the using the gestures really frustrating. To solve this I had to make more changes to the player source code. In Player::mousePressEvent(QMouseEvent *e) I added the code below, this only affects the VivioApp, when a mouse down event happens a timer is started and after 100ms the timer times out and sets click to false.

```cpp
#ifdef VIVIOAPP
    if (b==Qt::LeftButton)
    {
        if (mouseDownTimer->isActive() == 0)
            mouseDownTimer->stop();
        mouseDownTimer->start(100);
        click = true;
        return;
    }
#endif
```

This means that for VivioApp a click only happens if the mouse is down for less than 100ms and clicks are affect the animation on the mouse up event the following change was made to Player::mousePressEvent(QMouseEvent *e)

```cpp
#ifdef VIVIOAPP
    if (!click && b ==Qt::LeftButton) return;
#endif
```

I experimented with different delays for the click, at 200ms it was far too long but at 50ms I felt that it might be too quick for some users. So I erred on the side of caution and choose 100ms which sometimes is too long.
5.4 Dynamic menu

For the dynamic menu to meet the requirements set out in the objectives the menu had been created from a single line of code that when changed gave a new menu. There is already a list of animations found at [4]. The most important set of animations are the cache animations so I started working on a dynamic menu for them. Looking at the html source code [Appendix D] for Vivio 6.0/caches/ you can see a pattern all the names of the files are preceded by href=. So using the UrlReader class that is part of the player I can download the source of the page into a string.

\[
\text{char } \ast \text{ c}_{\text{str}} = \text{UrlReader::readURL(url, fsz);} \\
\text{QString string} = \text{QString(QLatin1String(c_{str})};
\]

By manipulating the string I can get a list of strings that contain the contents of the string after each point that href= appears in the original string.

\[
\text{QStringList pathlist} = \text{string.split("href=");}
\]

The output looks something like this

"MESI.gif">MESI.gif</a></td><…
"MESI.htm">MESI.htm</a></td><t…
"MESI.png">MESI.png</a></td…
"MESI.vcode">MESI.vcode</a></…

To get the file name all that is needed to do is split the string again this time using ". Unfortunately you cannot use " as an input character because it is used in C++ to show the start of a string. The value for this character is 34. The code below is the first version of SetUpButtons(QString url)

\[
\text{char spt} = 34; \\
\text{char } \ast \text{ c}_{\text{str}} = \text{UrlReader::readURL(url, fsz);} \\
\text{QString string} = \text{QString(QLatin1String(c_{str})}; \\
\text{QStringList pathlist} = \text{string.split("href=");} \\
\text{int possible} = \text{pathlist.count();} \\
\text{for(int i=6;i<\text{possible};i++)} \\
\{ \\
\text{QString test} = \text{pathlist.at(i)}; \\
\text{QStringList results} = \text{test.split(spt);} \\
\text{test} = \text{results.at(1)}; \\
\text{if(test.right(6)==".vcode")} \\
\{ \\
\text{test} = \text{url + test;} \\
\text{Buttons[\text{NofB}] = new MetroButton(test);} \\
\text{NofB++;} \\
\}
\]

Department of Computer Science, Trinity College Dublin
After splitting test we know that the file name is the first result and if the right six characters of the file name are .vcode then that is a file we are looking for and we add a new button to the list of buttons.

I then went back to the link [4] and looked at the source html and it follows the same pattern href= but this time there are folders and files. How do we know if it’s a folder? It has /at the end. So I modified the code above to look for folders as you can see below it is very similar.

```cpp
for(int i=6;i<possible;i++)
{
    QString test = pathlist.at(i);
    QStringList results = test.split(spt);
    test = results.at(1);
    if(test.right(1)=="/")
    {
        SetUpButtons(url + test);
        Heddings[NofL] = new QLabel(test);
        NofL++;
    } else if(test.right(6)==".vcode")
    {
        
    } 
```

I decide to add labels for each folder and recursively call the function. This would result in lots of labels that don’t have animations in them so I changed the function to return false if the url given to the function has no animations in it. If you look at the code [F] then you will see that I have also added some extra lines to deal with the layout of the menu. Below is the code for setting up the menu.

```cpp
if(SetUpButtons("https://www.scss.tcd.ie/Jeremy.Jones/vivio%206.0/"))
{
    int i=0;
    for(int j=0;j<NofL+1;j++)
    {
        while(i<NofB)
        {
            if(i==LabelAt[j])
            {
                grid->addWidget(Heddings[j],(j*2),0);
                break;
            }
            connectButtons[i],SIGNAL(click(QString)),this,SIGNAL(selectedfile(QString)));
            grid->addWidget(Buttons[i],(j*2)-1,i-(LabelAt[(j-1)]),1,1);
            i++;
        }
    }
```

In the above code i=number of buttons, j=current number of labels, NofB= total number of buttons, NofL= total number of labels, Heddings[] is an array of pointers to the labels, Buttons[] is an array of
points to the metro buttons, LabelAt[] is an array of integers that list at what point a label should be
placed. There is a LabelAt[] for each Heddings[].

The code checks if it should add a label and if it doesn’t then it adds a button to the menu and
connects the button’s click signal to the menu’s selectedfile signal. This signal is then connected to
the MyPlayer’s load slot.

The first line is the only line needed to be changed to make a different app just change the url to the
desired online directory, this must have the same formant as the directory used above[4], and
rebuild the app changing the name of the app may also be needed.

From reading the above code I have seen a bug that I had overlooked until now, if there are no sub
directories then there will not be any labels and so there will not be any buttons a simple fix to this is
to make sure that when you create a directory for the new app you make a parent directory for the
directory with the animations. Another is to add the following lines of code.

```cpp
if (NofL==0)
{
    NofL = 1;
    Heddings[0] = new QLabel("Animations");
    LabelAt[0] = 0;
}
for(int j=0;j<NofL+1;j++)
{
```

I tried to get the program to download an image and put it as the icon of the metro buttons but all
efforts at this failed. I spend many hours researching how to do this and wrote all the code to do it
but the line of code below always returned false. I didn’t have enough time to find out why it failed
but from looking at the output of bytes it appears that the image data is not stored that that location
and was just a pointer to the image that when opened with a browser it automatically fetches the
image I am not sure of this. More testing is needed.

```cpp
img.loadFromData(bytes)
```
Chapter 6 – VivioApp final result

The final result is an app that can play Vivio animations giving the user full functionally and control of the animations. VivioApp has all of the functionality that both Vivio and VivioNP have. VivioApp has one added feature the user can zoom in on the animations. This function is not necessary on the previous Vivio players but as Android devices have smaller screens zooming is a vital function of VivioApp. VivioApp has three views the main player, a popupmenu and a menu for selecting an animation (metro menu). In Figure 7 you can see the main screen and behind the main screen are the other two these are not visible by the user.

![Figure 7](image)

When in this screen the user can control the player using gestures, below is the list of gestures that can be used and a description of what happens when the user performs the gesture. The user can also press the menu button, circled in figure 7, this brings the popupmenu to the front of the screen and moves the player to the back.
<table>
<thead>
<tr>
<th>Gesture</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tap screen</td>
<td>Start/stop</td>
<td>This toggles the player between playing and stopped</td>
</tr>
<tr>
<td>Tap and hold</td>
<td>Single step mode</td>
<td>This toggles in and out of single step mode</td>
</tr>
<tr>
<td>Short wipe with one finger (not in single step mode)</td>
<td>Change speed</td>
<td>Right: increases the TPS of the player</td>
</tr>
<tr>
<td>Short wipe with one finger (in single step mode)</td>
<td>Single step</td>
<td>Right: moves the player to the next tick</td>
</tr>
<tr>
<td>Long swipe with one finger</td>
<td>Change direction</td>
<td>Right: play forwards, Left: play backwards</td>
</tr>
<tr>
<td>Swipe with three fingers (not in single step mode)</td>
<td>Snap to checkpoint</td>
<td>Left: snap to next checkpoint</td>
</tr>
<tr>
<td>Swipe with three fingers (in single step mode)</td>
<td>N/A</td>
<td>Right: snap to previous checkpoint</td>
</tr>
<tr>
<td>Pinch gesture*</td>
<td>Zoom</td>
<td>Makes the animation bigger/smaller</td>
</tr>
<tr>
<td>Pan gesture *</td>
<td>Pan</td>
<td>Moves the player around the screen</td>
</tr>
</tbody>
</table>

*Pan and pinch require two fingers.

Figure 8

In figure 8 above you can see the popupmenu clicking the top button will bring the metro menu to the front and moves the popupmenu to the back. clicking any of the other options will perform the
responding action on the player in the same way that would happen if the user used a gesture on
the main screen. Pressing the menu button will bring the main screen back to the front.

Figure 9

In figure 9, you can see the metro menu. This menu is automatically populated with labels for each
folder and buttons for each animation within the folders. Clicking on any of the metro buttons will
signal the player to load the appropriate animation and bring the player screen to the front and
move the metro menu to the back. Pressing the menu button will bring the player screen to the
front and move the metro menu to the back but the animation will not be changed.
Chapter 7 – Analysis and conclusion

7.1 Does VivioApp meet the Objectives?

- Vivio on Android – The primary objective of this project has been achieved. VivioApp runs on Android and has the full functionality of Vivio with the option to use the popupmenu of use touch screen gesture. Below is a table showing the functions of Vivio and difference between VivioApp and VivioNP.

<table>
<thead>
<tr>
<th>Command</th>
<th>VivioNP</th>
<th>VivioApp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play/stop player</td>
<td>Left mouse click</td>
<td>Tap the screen</td>
</tr>
<tr>
<td>Change speed</td>
<td>Scroll mouse wheel</td>
<td>Short swipe with one finger (normal mode)</td>
</tr>
<tr>
<td>Change playing direction</td>
<td>Scrolling the mouse wheel when the player is stopped</td>
<td>Long swipe with one finger</td>
</tr>
<tr>
<td>Play to checkpoint</td>
<td>Right click to access menu</td>
<td>Press menu button for menu</td>
</tr>
<tr>
<td>Snap to checkpoint</td>
<td>Right click to access menu</td>
<td>three fingered swipe</td>
</tr>
<tr>
<td>Single step</td>
<td>Scroll mouse wheel when player is stopped</td>
<td>Short swipe with one finger (singlestep mode)</td>
</tr>
<tr>
<td>Single step mode</td>
<td>N/A</td>
<td>Tap and hold screen to enter/exit singlestep mode</td>
</tr>
<tr>
<td>Zooming/panning</td>
<td>N/A</td>
<td>Use two fingers to zoom and/or pan</td>
</tr>
</tbody>
</table>

- Platform independent – Qt is platform independent and is in the process of being ported to iOS and other mobile platforms. This objective has partially been met but in the future VivioApp will meet this objective without any change to the source code.

- User-friendly – This objective highly depends on each user. Overall it achieves this objective as it can be controlled with or without gestures. Further tests need to be done to confirm whether or not this objective has fully been achieved.

- Flexible – The app is highly flexible as it can be run on any screen size because of the fact that the user can zoom in on the animation to get a better view, the animations are also very accommodating to zooming because they are vector based 2D animations meaning that they do not lose information when zooming in or out. Adding new animations is very easy and doesn’t require the user to download the app again. The creator of the new animation simply puts it into the Vivio directories. Creating a new app with a new set of animations is also very simple and requires very little change to the source code. This objective has been met.
7.2 Conclusions

Someone, wiser than I, once said to me “Every advantage is also a disadvantage”. Laptop users, particularly students, are gravitating to smart devices because of their smaller size and consequent portability but the small screen size makes for difficulty in running useful applications such as Vivio animations. This project set out to develop a player to allow Vivio animations to run on smart devices.

This project has been a success, a working version of VivioApp can now run on Android and if compiled for other platforms should run on them also. As you can see the project met all or most of each objective.

In my opinion this is a step in the right direction. The current challenge, for developers, is to produce applications that help to make popular devices more amenable to user’s needs.

The success of the final produce confirms that Qt was the right choice for this project. In my opinion Qt will continue to improve and grow. Sometime in the near future there will be a port of Qt to all the major operating systems including mobile operating systems. I am very happy with the outcome of the project.

Someone at IBM once famously said “In the future there will be a market for maybe 2 personal computers in the world”. We have come a long way since then with the proliferation not only of personal computers but laptops and now smart devices. Who knows what is coming down the road - Google glasses, smart hats? Whatever it is there will need to be developers working on applications to suit new devices. “But knowing how way leads on to way” (Frost) at some stage in the future this project will have lost its relevance but for now I believe it is a contribution to the current state of the art.
Chapter 8 – Recommendations for future work

8.1 Deployment tests

As there are many different combinations of Android versions and processors, VivioApp needs to be tested on as many of these combinations as possible, I have asked some friends and family who have Android phones to download VivioApp and I got mixed results. As far as I can tell the more recent Android versions do better (ice cream sandwich). A more important factor is the users’ patience.

8.2 Usability tests

Throughout the project I have tried to make VivioApp as user friendly as possible but I am just one user and I will admit to having opinions on how the apps I use should operate. It is very easy to see that other programmers and users have different opinions on this subject, because of the number of apps that I don’t like. I would like to conduct some independent tests to determine the usability of the gestures, can most users perform all of the gestures, and do they always work. I would also like to ask the users if they could change the app what changes they would make.

In the metro menu the buttons are laid out in a 2D space independent of the screen size. The user can use two fingers to pan around the screen left/right and up/down. I had thought of implementing it so that the menu wouldn’t be wider than the screen, so the user would just use two fingers to scroll up/down. I had implemented some of the code for this to test if it was possible and it is. The code in [Appendix E] shows how this would be implemented.

8.3 Look of VivioApp

I didn’t spend a lot of time worrying about how the app looked, as I was focused on getting the functionality working correctly, I’m not sure if it looks good. I personally like the look of the app, blue is my favourite colour and I like simple clean looking apps. I’m sure there are people that disagree and I would like to get as many opinions about the look of VivioApp as possible.
8.4 Bug fixes and features

Fixing the bug mentioned on page 32 would be the first priority. There is one other known bug in VivioApp, if the user clicks more than one option in the metro menu then the app will crash. I would have fixed this bug but I was in the middle of implementing other futures and had decided to fix it at the end but time ran out and it never got fixed.

Offline mode:
During the project I had thought about having the app store the 5 most recent animations on the device but as most apps require the user to be online I decided that it wasn’t necessary. This wouldn’t be too difficult to implement but it would involve having a cache for the animations and implementing a replacement policy. In my opinion it is not necessary as free Wi-Fi is very common in Dublin, coffee shops, fast food outlets and even busses and trains have Wi-Fi on board. As of 31/1/2013 Dublin has three free public Wi-Fi hotspots and several more are planned.

“Lorna Maxwell, from the council’s economic development unit, said free internet will be available to users in 12 areas of the city centre by the end of April including Grafton Street, O’Connell Street and Temple Bar.” [17]

Placing an image on each button in the metro menu is something I would like to see in the future as I was unable to implement this. Lack of time meant that I was unable to test my code and find the reason that it didn’t work.
References


Dr. Jeremy Jones, Dept. Computer Science, Trinity College, Dublin 2, Ireland.


[6] “Necessitas Android Port Contributed to the Qt Project”, Tuukka Turunen, Qt blog
http://blog.qt.digia.com/blog/2012/11/08/necessitas-android-port-contributed-to-the-qt-project/


[12] “How Qt Signals and Slots Work”, Olivier Goffart
http://woboq.com/blog/how-qt-signals-slots-work.html

Qt documentation, http://qt-project.org/doc/


[16] “C++ GUI Programming with Qt 4” second edition, Jasmin Blanchette and Mark Summerfield
[17] “Parts of Dublin get free public WiFi”, Dan Griffin, The Irish Times  

[*] Estimated without independent research
Appendix A.1 – Vivio source code

const int red = rgba(255, 0, 0);
const int green = rgba(0, 255, 0);
const int blue = rgba(0, 0, 255);
const int yellow = rgba(255, 255, 0);

Brush redbrush = SolidBrush(red);
Brush greenbrush = SolidBrush(green);
Brush bluebrush = SolidBrush(blue);
Brush yellowbrush = SolidBrush(yellow);

const int WIDTH = 300;
const int HEIGHT = 200;

setViewport(0, 0, WIDTH, HEIGHT, 1); // (_._,L3,L4, [0]fill screen [1]const size)

setTPS(10); // initially 1 tick per second
start(); // start animation

RecA = Rectangle2(0, 0, redbrush, 0, 0, 50, 50);

while (1)
{
    RecA.setOpacity(255);
    RecA.translate(250, 0, 30, 1, 0);
    RecA.setOpacity(0, 30, 1, 1);
    RecA.setBrush(bluebrush);

    RecA.setOpacity(255);
    RecA.translate(0, 150, 20, 1, 0);
    RecA.setOpacity(0, 20, 1, 1);
    RecA.setBrush(greenbrush);

    RecA.setOpacity(255);
    RecA.translate(-250, 0, 30, 1, 0);
    RecA.setOpacity(0, 30, 1, 1);
    RecA.setBrush(yellowbrush);

    RecA.setOpacity(255);
    RecA.translate(0, -150, 20, 1, 0);
    RecA.setOpacity(0, 20, 1, 1);
    RecA.setBrush(redbrush);
}
Appendix A.2 – vcode

!! 00000000 04ff 0000 ffff
!! 00000003 04fc
!! 00000004 0115
!! 00000005 04ff ff00 ff00
!! 00000008 04fc
!! 00000009 0215
!! 0000000a 04ff 00ff ff00
!! 0000000d 04fc
!! 0000000e 0315
!! 0000000f 04fe ff00
!! 00000011 04fc
!! 00000012 0415
!! 00000013 0104
!! 00000014 00f8 4348
!! 00000016 00f8 4396
!! 00000018 02bf
!! 00000019 50fc
!! 0000001a 0a04
!! 0000001b 79fc!! 0000001c 74fc
!! 0000001d 03bc
!! 0000001e 00f8 4248
!! 00000020 00f8 4248
!! 00000022 02bf
!! 00000023 0109
!! 00000024 03bf
!! 00000025 0009
!! 00000026 90fc
!! 00000027 0515
!! 00000028 03bc
!! 00000029 00f8 437f
!! 0000002b 0509
!! 0000002c 2bfc
!! 0000002d 00bd 1e01
!! 0000002f 00f6
!! 00000030 00f8 437a
!! 00000032 0509
!! 00000033 3cfc
!! 00000034 01bd 1e01
!! 00000036 00f6
!! 00000037 0509
!! 00000038 2bfc
!! 00000039 0305
!! 0000003a 0509
!! 0000003b 26fc
!! 0000003c 03bc
!! 0000003d 00f8 437f
!! 0000003f 0509
!! 00000040 2bfc
!! 00000041 00bd 1401
!! 00000043 00f8 4316
!! 00000045 00f6
!! 00000046 0509
!! 00000047 3cfc
!! 00000048 01bd 1401
!! 0000004a 00f6
!! 0000004b 0509
!! 0000004c 2bfc
!! 0000004d 0205
!! 0000004e 0509
!! 0000004f 26fc
!! 00000050 03bc
!! 00000051 00f8 437f
!! 00000052 0509
!! 00000053 2bfc
!! 00000054 00bd 1e01
!! 00000055 00f6
!! 00000056 0509
!! 00000057 01bd 1401
!! 00000058 00f8 c37a
!! 00000059 0509
!! 0000005a 3cfc
!! 0000005b 01bd 1e01
!! 0000005c 00f6
!! 0000005d 0509
!! 0000005e 2bfc
!! 0000005f 0405
!! 00000060 0509
!! 00000061 2bfc
!! 00000062 0509
!! 00000063 26fc
!! 00000064 03bc
!! 00000065 00f8 437f
!! 00000066 0509
!! 00000067 2bfc
!! 00000068 00bd 1401
!! 00000069 00f8
!! 0000006a 0509
!! 0000006b 3cfc
!! 00000070 01bd 1401
!! 00000071 00f6
!! 00000072 0509
!! 00000073 2bfc
!! 00000074 0105
!! 00000075 0509
!! 00000076 26fc
!! 00000077 b0a8
Appendix B – DLX animation
Appendix C – Qt practice code

//myslider.h
#ifndef MYSLIDER_H
#define MYSLIDER_H

#include <QSlider>
#include <QWidget>

class MySlider : public QSlider
{
    Q_OBJECT
public:
    explicit MySlider(QWidget *parent = 0);

signals:
    void valueChanged(const QString);

public slots:
    void setValue(QString);
    void send(int);
};
#endif // MYSLIDER_H
//eof

//myslider.cpp
#include "myslider.h"
#include <QString>

MySlider::MySlider(QWidget *parent) : QSlider(parent)
{
}

void MySlider::setValue(QString s)
{
    bool ok = true;
    int x = s.toInt(&ok,10);
    setSliderPosition(x);
}

void MySlider::send(int i)
{
    QString s;
    i++;
    s.append(QString("%1").arg(i));
    emit valueChanged(s);
}
//eof
Appendix D - HTML source code

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
<title>Index of /Jeremy.Jones/vivio 6.0/caches</title>
</head>
<body>
<h1>Index of /Jeremy.Jones/vivio 6.0/caches</h1>
<table>
<tr><th><img src="/icons/blank.gif" alt="[ICO]">
</th><th><a href="?C=N;O=D" >Name</a></th><th><a href="?C=M;O=A">Last modified</a></th><th><a href="?C=S;O=A">Size</a></th><th><a href="?C=D;O=A">Description</a></th></tr>
<tr><th colspan="5"><hr></th></tr>
<tr><td valign="top"><img src="/icons/back.gif" alt="[DIR]"></td><td><a href="/Jeremy.Jones/vivio%206.0/">Parent Directory</a></td><td></td><td align="right">-</td><td></td></tr>
<tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"></td><td><a href="ALL%20protocols.htm">ALL protocols.htm</a></td><td align="right">02-Jan-2007 10:32 </td><td align="right">1.5K</td><td></td></tr>
<tr><td valign="top"><img src="/icons/image2.gif" alt="[IMG]"></td><td><a href="MESI.gif">MESI.gif</a></td><td align="right">17-Nov-2009 11:56 </td><td align="right">15K</td><td></td></tr>
<tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"></td><td><a href="MESI.htm">MESI.htm</a></td><td align="right">02-Jan-2007 10:32 </td><td align="right">756 </td><td></td></tr>
<tr><td valign="top"><img src="/icons/image2.gif" alt="[IMG]"></td><td><a href="MESI.png">MESI.png</a></td><td align="right">16-Nov-2009 23:57 </td><td align="right">28K</td><td></td></tr>
<tr><td valign="top"><img src="/icons/unknown.gif" alt="[ ]"></td><td><a href="MESI.vcode">MESI.vcode</a></td><td align="right">22-Feb-2013 09:04 </td><td align="right">2.8K</td><td></td></tr>
</table>
href="MESI.vsd" MESI.vsd</a></td><td align="right">17-Nov-2009 11:55</td><td align="right">53K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"/></td><td><a href="MESIHelp.htm">MESIHelp.htm</a></td><td align="right">19-Jan-2012 23:20</td><td align="right">9.7K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"/></td><td><a href="cache.htm">cache.htm</a></td><td align="right">02-Jan-2007 10:32</td><td align="right">545</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/unknown.gif" alt="[ ]"/></td><td><a href="cache.vcode">cache.vcode</a></td><td align="right">22-Feb-2013 09:04</td><td align="right">3.4K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"/></td><td><a href="cacheHelp.htm">cacheHelp.htm</a></td><td align="right">10-Sep-2007 22:53</td><td align="right">3.8K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/image2.gif" alt="[IMG]"/></td><td><a href="firefly.gif">firefly.gif</a></td><td align="right">06-Dec-2002 08:08</td><td align="right">9.7K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"/></td><td><a href="firefly.htm">firefly.htm</a></td><td align="right">02-Jan-2007 10:32</td><td align="right">768</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/unknown.gif" alt="[ ]"/></td><td><a href="firefly.vcode">firefly.vcode</a></td><td align="right">22-Feb-2013 09:04</td><td align="right">2.9K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/unknown.gif" alt="[ ]"/></td><td><a href="firefly.vsd">firefly.vsd</a></td><td align="right">06-Dec-2002 08:09</td><td align="right">90K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"/></td><td><a href="fireflyHelp.htm">fireflyHelp.htm</a></td><td align="right">10-Sep-2007 22:54</td><td align="right">16K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/unknown.gif" alt="[ ]"/></td><td><a href="t3.doc">t3.doc</a></td><td align="right">07-Dec-2001 09:01</td><td align="right">57K</td>&nbsp;</tr></td></tr><tr><td valign="top"><img src="/icons/image2.gif" alt="[IMG]"/></td><td><a href="tcd.gif">tcd.gif</a></td><td align="right">11-Jun-2002 22:16</td><td align="right"></td>&nbsp;</tr></td></tr>
align="right">3.1K</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/image.gif" alt="[IMG]"/>vivio0.png</a></td><td align="right">22-Feb-2013 09:04</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/image.gif" alt="[IMG]"/>vivio1.png</a></td><td align="right">22-Feb-2013 09:04</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/image.gif" alt="[IMG]"/>vivio2.png</a></td><td align="right">22-Feb-2013 09:04</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/unknown.png" alt="[ ]"/></td><td align="right">06-Dec-2002 08:23</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/image.gif" alt="[IMG]"/>writeOnce.gif</a></td><td align="right">07-Jan-2003 07:43</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/text.gif" alt="[TXT]"/></td><td align="right">02-Jan-2007 10:32</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/unknown.png" alt="[ ]"/></td><td align="right">22-Feb-2013 09:04</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/text.gif" alt="[TXT]"/></td><td align="right">10-Sep-2007 22:54</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/image.gif" alt="[IMG]"/>writeThrough.gif</a></td><td align="right">06-Dec-2002 05:56</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/text.gif" alt="[TXT]"/></td><td align="right">02-Jan-2007 10:32</td><td>&nbsp;</td></tr>

<tr><td align="right"><img src="/icons/unknown.png" alt="[ ]"/></td><td align="right">22-Feb-2013 09:04</td><td>&nbsp;</td></tr>
<tr><td valign="top"><img src="/icons/unknown.gif" alt="[ ]"></td><td><a href="writeThrough.vsd">writeThrough.vsd</a></td><td align="right">13-Feb-2003 06:36</td><td align="right">85K</td><td>&nbsp;</td></tr>

<tr><td valign="top"><img src="/icons/text.gif" alt="[TXT]"></td><td><a href="writeThroughHelp.htm">writeThroughHelp.htm</a></td><td align="right">10-Sep-2007 22:55</td><td align="right">7.3K</td><td>&nbsp;</td></tr>

<tr><th colspan="5"><hr></th></tr>

<address>Apache/2.2.16 (Debian) Server at www.scss.tcd.ie Port 443</address>

</body></html>
Appendix E – Menu option code

//This is the only function that needs to be changed
void Metroselectfile::SetUpMenu()
{
    QGridLayout *grid = new QGridLayout();
    QScrollArea *area = new QScrollArea();

    const int width = (QApplication::desktop()->width());
    const int MaxBPL = (int) width/130;

    int count = 0, extra = 0;
    NofB = NofL=0;
    if(SetUpButtons("https://www.scss.tcd.ie/Jeremy.Jones/vivio%206.0/"))
    {
        int i=0;
        for(int j=0;j<NofL+1;j++)
        {
            while(i<NofB)
            {
                if(i==LableAt[j])
                {
                    grid->addWidget(Heddings[j],(j*2)+extra,0);
                    count =0;
                    break;
                }
            }
            connect(Buttons[i],SIGNAL(click(QString)),this,SIGNAL(selectedfile(QString)));
            grid->addWidget(Buttons[i],((j*2)-1)+extra,i-(LableAt[(j-1)])),1,1);
            i++;
            count++;
            if(count == MaxBPL) count=0;
        }
    }

    sub->setLayout(grid);
    area->setWidget(sub);

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(area);
    this->setLayout(layout);
}
Appendix F – Project code

The following files are included in this appendix.

main.h
main.cpp
myplayer.h
myplayer.cpp
GestureRecognizers.h
GestureRecognizers.cpp
Popupmenu.h
Popupmenu.cpp
Metroselectfile.h
Metroselectfile.cpp