



**WPI**



**The Qt  
Company**

DESIGN AND DEVELOPMENT OF A  
TRADITIONAL ANIMATION TOOL IN  
QT QUICK DESIGNER

A MAJOR QUALIFYING PROJECT REPORT

SUBMITTED TO THE FACULTY OF THE

**Worcester Polytechnic Institute**

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF BACHELOR OF SCIENCE

BY:

---

ADILET ISSAYEV

---

PATRICK LEBOLD

---

MAURIZIO VITALE

DATE: MARCH 3, 2017

APPROVED:

---

PROFESSOR MARK CLAYPOOL, ADVISOR

*This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>.*

# Abstract

Qt Creator is an IDE used for developing Qt GUI applications. While Qt Creator provides a graphical context for developing interfaces, the only way to create animations using the Qt framework is to code them. We developed an animation editing tool built inside of Qt Creator that eliminates the current requirement for designers to code animations. Additionally, by analyzing industry standard animation tools and developing a structured way to organize animations using existing Qt framework libraries, we defined a new schema for representing animations in Qt's front-end language, QML. Overall, our tool facilitated the process of developing animations in Qt Creator by providing an easy to use graphical interface, and is set to be included in future versions of the Qt software development kit.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	The Qt Company . . . . .	7
2.2	Qt Framework . . . . .	7
2.2.1	Advantages of Qt . . . . .	8
2.2.2	Signals and Slots . . . . .	8
2.2.3	QtWidgets . . . . .	9
2.2.4	Models and Views . . . . .	9
2.3	QML . . . . .	10
2.3.1	Advantages of QML . . . . .	10
2.3.2	Items and Attributes . . . . .	10
2.3.3	Interacting with Components . . . . .	11
2.3.4	Signals and Connections . . . . .	12
2.3.5	Loaders . . . . .	12
2.3.6	Animations . . . . .	13
2.4	Qt Quick Designer . . . . .	13
2.4.1	Introduction . . . . .	13
2.4.2	Document Manager and Model Node Structure . . . . .	14
2.4.3	View Manager . . . . .	15
2.4.4	Components . . . . .	15
2.5	Related Works . . . . .	17
2.5.1	Hype . . . . .	17
2.5.2	Google Web Designer . . . . .	18
2.5.3	Qt 3D Studio . . . . .	19
2.5.4	Adobe After Effects . . . . .	19
<b>3</b>	<b>Methodology and Implementation</b>	<b>21</b>
3.1	Requirements . . . . .	21
3.2	Design . . . . .	22
3.2.1	Wireframes . . . . .	22
3.2.2	Interaction Design . . . . .	24
3.3	Interfacing with Qt Quick Designer . . . . .	28
3.3.1	Creating a Component . . . . .	28
3.3.2	Registering the Component . . . . .	29
3.3.3	Linking the Document Model to Components . . . . .	30
3.3.4	Loading QML in a Component . . . . .	30
3.4	Developing a Timeline Model . . . . .	31
3.4.1	Model Requirements . . . . .	31
3.4.2	Defining a Timeline Schema in QML . . . . .	31
3.4.3	Defining the Model Structure . . . . .	32
3.4.4	Separating the Model from the View . . . . .	33
3.4.5	Linking the Model to QML . . . . .	34
3.5	Developing the Timeline Navigator . . . . .	34

3.5.1	Navigator Requirements . . . . .	34
3.5.2	Adding and Selecting Timelines . . . . .	35
3.5.3	Adding and Viewing Timeline Items . . . . .	35
3.5.4	Adding and Viewing Timeline Item Properties . . . . .	36
3.6	Developing the Keyframe Area . . . . .	36
3.6.1	Keyframe Area Requirements . . . . .	36
3.6.2	Implementing the Ruler . . . . .	37
3.6.3	Keeping Track of Time . . . . .	37
3.6.4	Keyframe Rows . . . . .	38
3.6.5	Adding Keyframes . . . . .	39
3.6.6	Keyframe Interactions . . . . .	39
<b>4</b>	<b>Results</b>	<b>41</b>
<b>5</b>	<b>Future Steps</b>	<b>43</b>
5.1	Form Editor Animation Playback . . . . .	43
5.2	Keyframe Editing in Property Editor . . . . .	44
5.3	Additional Animation Functionality . . . . .	44
<b>6</b>	<b>Conclusion</b>	<b>46</b>
<b>A</b>	<b>Qt for Native Client</b>	<b>49</b>
A.1	Introduction . . . . .	49
A.2	Background . . . . .	49
A.2.1	Native Client . . . . .	49
A.2.2	Pepper Plugin API . . . . .	49
A.2.3	Project Files and Qmake . . . . .	50
A.2.4	Qt Platform Abstraction . . . . .	50
A.3	Methodology . . . . .	51
A.3.1	Outdated Qt for NaCl . . . . .	51
A.3.2	Preparing the Compilation Environment . . . . .	51
A.3.3	Compiling QtBase and Qmake for NaCl . . . . .	51
A.3.4	Building Qt for NaCl Modules . . . . .	52
A.3.5	Building Sample Application for Qt for NaCl . . . . .	52
A.4	Results . . . . .	52
A.5	Future Steps . . . . .	53

# List of Figures

2.1	Signals and Slots . . . . .	8
2.2	A red rectangle item defined in QML . . . . .	10
2.3	A blue rectangle defined by the red rectangle's properties. . . . .	11
2.4	A button control item causes itself to change position on click . . . . .	11
2.5	Two items interact with each other through the use of signals and slots. . . . .	12
2.6	QML Loader . . . . .	13
2.7	Animation Item in QML . . . . .	14
2.8	The Navigator Component . . . . .	15
2.9	The Property Editor Component . . . . .	16
2.10	The Form Editor Component . . . . .	16
2.11	Hype Interface . . . . .	17
2.12	Hype, separation of keyframes by item property . . . . .	17
2.13	Hype property list . . . . .	18
2.14	Google Designer's toolbar . . . . .	18
2.15	Google Designer's animation interpolation . . . . .	19
2.16	Qt 3D Studio . . . . .	19
2.17	Adobe After Effects interface . . . . .	20
2.18	Adobe After Effects keyframe properties . . . . .	20
3.1	The Main Layout of the Timeline Tool . . . . .	22
3.2	The components of the Keyframe Area . . . . .	22
3.3	The navigator area wireframe . . . . .	23
3.4	The titlebar wireframe . . . . .	24
3.5	A keyframe being added to the keyframe area . . . . .	24
3.6	Moving keyframe by grabbing whole keyframe . . . . .	25
3.7	Moving keyframe by dragging keyframe handles . . . . .	25
3.8	Coupling keyframes . . . . .	25
3.9	Dialog for modifying keyframe values . . . . .	26
3.10	Ruler mouse click and current time change . . . . .	26
3.11	The Playback Buttons . . . . .	27
3.12	Clock control's interaction . . . . .	27
3.13	Process of changing timeline in navigator . . . . .	27
3.14	Add timeline dialog . . . . .	28
3.15	Add property to an item in Navigator . . . . .	28
3.16	The widgetInfo function . . . . .	29
3.17	Additions to the attachViewsExceptRewriterAndComponetView function . . . . .	29
3.18	Our Timeline Component's placement in the Qt Quick Designer Hierarchy . . . . .	30
3.19	Function which reloads QML . . . . .	31
3.20	This schema represents a basic timeline that animates two items. . . . .	32
3.21	The final component architecture . . . . .	33
3.22	QML code defining the timeline list and add timeline features . . . . .	35
3.23	The QML source of the step-back button. . . . .	38
3.24	QML Source code that defines the editing component of the title bar's clock. . . . .	38
3.25	The add keyframe slot . . . . .	39

4.1	The final look of the Timeline Tool . . . . .	41
-----	---	----

# Chapter 1

## Introduction

As the rate of new technologies appearing increases, companies of all sizes find themselves struggling to keep supporting their products on all of the newly available platforms. End users are no longer using applications solely on their computers, but also on phones, tablets, and embedded devices like car interfaces and home-security panels. Even within the same category of device such as phones, companies are often forced to create different versions of their app that conform with each type of phones interior structures. These hardware and software requirements create the need for multiple applications to be made, even if they all look the same to the end user. The Qt Company offers a solution to this problem by providing a platform that supports the development of cross-platform applications. End users can use The Qt Company's development platform, Qt Creator, to develop their applications.

The scope of our project lays within Qt Quick Designer, a plugin for Qt Creator that allows users to graphically design their applications by dragging and dropping widgets onto a form. When these widgets are placed, code is generated in the background that represents the corresponding widgets. Frequently, users will want to create interactive applications by adding animations involving these widgets. This however can not be done graphically; in order to create animations for Qt applications, users need to program them. Graphic designers typically have no knowledge of how to code, thus making creating animations a daunting task. The result of this project was the addition of a tool in Qt Quick Designer that allows users to design animations without having to write a single line of code.

This paper acts as a roadmap detailing the technologies used in this project, the requirements of this project, our design and implementation methodology for htis project, and the results and possibilities of future work regarding this project. Specifically, chapter two provides a background on The Qt Company, its framework Qt, Qt's front-end language QML, Qt Creator, an IDE used to create Qt applications, and other leading animation tools. Chapter three provides details on project requirements and a close look at our design and implementation methodology for each component of our tool. Chapters four and five discuss the results and opportunities for future work respectively.

# Chapter 2

## Background

In order to begin discussing the creation of an animation editing tool for Qt applications, it is important to understand a variety of background topics. The following section provides context for the The Qt Company, the Qt framework, the markup language used for creating interfaces, the development environment used for creating Qt applications, and other leading animation tools.

### 2.1 The Qt Company

The Qt Company, based in Espoo, Finland, is primarily responsible for the development and distribution of the Qt framework and manages an open governance model for the licensing and contributions of the framework. The Qt Company originally started as the company Trolltech, after the founders Haavard Nord and Eirik Chambe-Eng were tasked to write a database application for ultrasound imaging [1]. This application, written in C++, required a GUI toolkit which could run on Unix, Macintosh and Windows. This laid down the foundation for the Qt framework.

Haavard and Eirik started working on Qt in 1991, and their product gradually improved and expanded, featuring a framework design that worked around signals and slots [2]. With Qt software, they founded Trolltech company in 1994. Nokia Corporation acquired Trolltech in 2008 and renamed the company to Qt Software at Nokia. Nokia later sold the commercial licensing of Qt to Digia, resulting in the creation of “The Qt Company”, a wholly owned subsidiary whose purpose is for Qt development and market expansion. Just recently in 2016, The Qt Company separated from Digia.

The Qt Company’s licensed product is used in more than 70 industries worldwide for creating, building, and deploying millions of connected embedded devices and applications, and is used by 8 of the top 10 Fortune 500 companies [3].

### 2.2 Qt Framework

Qt is a comprehensive, open-source framework written in C++ used to design applications capable of running on any platform [3]. The framework divides itself into several modules that support 3D graphics, bluetooth connections, data visualizations, networking, and intermediary languages such as the Qt Markup Language (QML).



### 2.2.1 Advantages of Qt

The primary advantage of using Qt is its support for cross platform development [4]. Because of this support, developers do not need to concern themselves with platform specific details and data types, and can instead focus on higher level implementations of their application. Qt provides abstraction by resolving intermediate classes, such as QString, to their appropriate system-level representations.

Qt originally supported two versions, one for X11 and one for Win32 window systems, which resulted in a platform-dependant codebase. This made Qt as a framework hard to port to new systems, giving start to a new project called the Qt Platform Abstraction [5]. The results of this project made Qt flexible enough to be ported and tailored to any specific platform without the need to modify the entire codebase.

### 2.2.2 Signals and Slots

Because Qt is designed to be object oriented, it is critical that there is a system in place that notifies objects that other objects have been changed. A typical solution to notifying one object of a change in another object is the use of callback functions. That being said, callback functions have one primary disadvantage; callback functions are strongly coupled to the processing function. As seen in Figure 2.1, in order to link a callback function to a separate object, that object must be aware of the existence of the callback function's object. This forces developers to provide public or protected access to member functions that may otherwise be hidden.

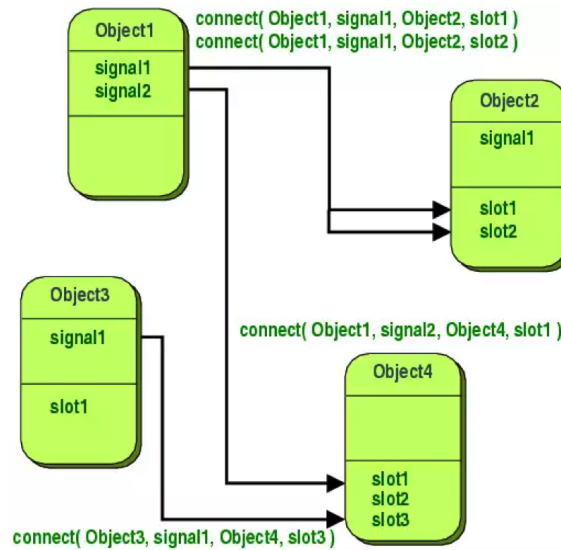


Figure 2.1: Signals and Slots  
[6]

The Qt framework circumvents this design flaw by utilizing a signal and slot system [6]. By defining a function labeled in a header file as either a signal or a slot, a developer can provide access for that function to be linked to another slot or signal. As demonstrated in Figure 2.1, this connection is established by calling the globally available connect function and providing the

appropriate objects, signal function, and slot function. This abstraction allows a signal function call to be paired to a slot function without the need for either object to have knowledge of the other's existence.

In order for this system to be effective, there are a few restrictions in place. First, the signal function must remain as a function prototype and can not be implemented. The sole purpose of a signal is to trigger the connected slots, and therefore no additional implementation of that function is permitted. In accordance with that, the return type of all signal functions must be defined as void, since there is no way for a signal to return any other value.

Any signal matched with a slot must contain the same number of parameters, with each parameter in the signal being the same type as each corresponding parameter in the connected slot. Unlike signals, slots can be implemented and called as normal C++ functions, and therefore do not have to have a matching void return type.

### 2.2.3 QtWidgets

Qt's windowing system, QtWidgets, provides dialogs, common layouts, and UI elements while still managing to keep a cross-platform compatibility layer [7]. Qt windows and forms can be added programmatically in C++ or an XML schema (in .ui format). This XML schema is used to define widgets, widget properties, and bindings to C++ variables and functions [8]. By utilizing Qt's cross-platform compatibility, widgets using the Qt framework can naturally capture any platform's native look and feel. Many properties of widgets such as font, color, border style, and size can be modified to further define the look and feel of an application. The Qt Company provides a tool called Qt Designer which allows graphic designers to create Qt windows and forms and modify properties graphically instead of programmatically.

### 2.2.4 Models and Views

The Qt framework uses a simplified Model-View-Controller design pattern; by combining the controller and view components, the Qt framework utilizes a Model-View pattern [9]. This combination results in a simpler framework which still separates the way data is stored from the way it is presented. Additionally, a Model-View pattern allows for higher modularity, as interaction layers do not have to be written in order to synchronize data between model and view layers.

The Qt framework introduces the concept of delegates, objects responsible for editing a model's data and dictating how this data is rendered [10]. A delegate object acts as a replacement for classic controller objects and relegates most controller duties from the view to itself. Delegates are modular by nature, being able to load data from different models and be instantiated into any view. One use case of a delegate is a spin box, an object with arrows that can increment or decrement any property in a model [11]. For each spin box included in an application, a single implementation of a delegate can be used to link each spin box to a distinct property.

The preferred method of communication between models, views, and delegates are signals and slots. Signals emitted from models inform delegates that data has changed. Signals from views provide delegates with interaction data including which items are involved in the interaction.

Signals from delegates inform models that their data will be modified and views that they need to be refreshed.

## 2.3 QML

Like most applications, applications written with the Qt frame have both backend and frontend components. Qt application backends are written in C++, while the frontends are written in Qt's internally developed markup language, QML.

### 2.3.1 Advantages of QML

QML is a markup language which is used to define the structure and styling of Qt applications [12]. QML is a hybrid of Javascript and JSON syntax that allows users to define and deploy complex interfaces, interactions, and animations with relative ease. QML provides bindings for C++ structures which opens the door to dynamic interfaces. By using QML, users can create networked applications, 3D showcases, games, and more.

### 2.3.2 Items and Attributes

By using QML's JSON-like syntax, users can add items such as shapes, buttons, lists of items, and more to their application's model [13]. Each type of item has a unique set of properties that can be assigned to them [14]. Items inherit a common set of properties from the QML Item class as well, giving it access to common properties such as position and dimensions. Users can define their own complex items by embedding basic items and defining custom properties and functions.

Figure 2.2 features a rectangle item in an application defined using QML. In this example, the window frame is 100px wide by 100px tall. The rectangle is given an id of `red_rect`, a position of (50,50), and a dimension of 50x50. This places the rectangle's left point at the center of the application and causes the rectangle to occupy the entire bottom-right quadrant of the app. The color of the rectangle is written in hexadecimal, however QML also supports color constants written as strings including `red`, `blue`, etc.

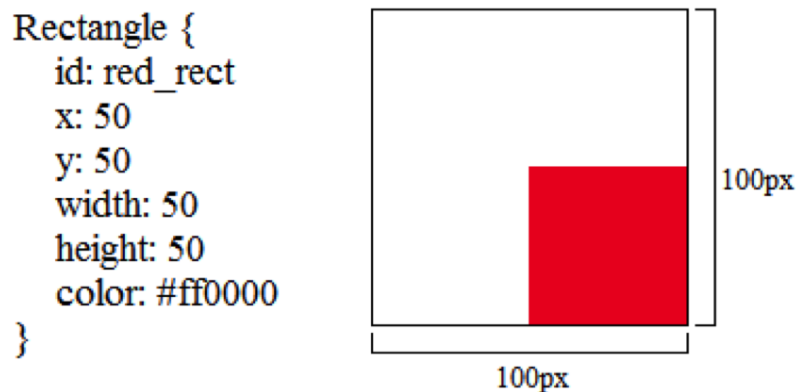


Figure 2.2: A red rectangle item defined in QML

By using basic Javascript syntax, the value of any property can be a variable. These variable properties can be relative to other item properties, variables passed in through C++ code, and environment variables. Figure 2.3 defines `blue_rect`, a blue rectangle which occupies the bottom-right corner of `red_rect`. This item uses `red_rect`'s properties to define its own position and dimension.

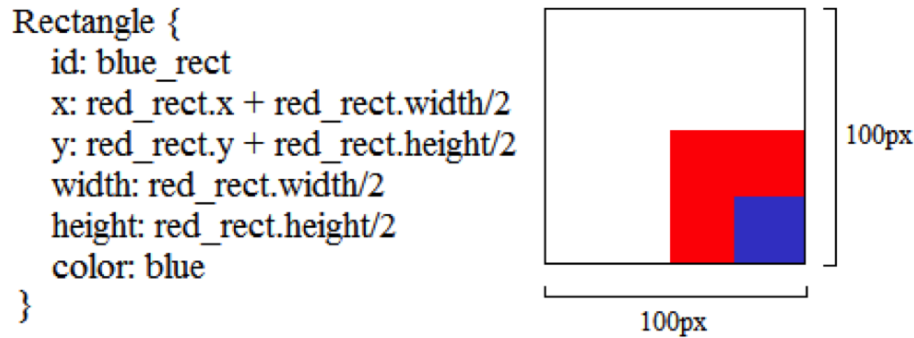


Figure 2.3: A blue rectangle defined by the red rectangle's properties.

### 2.3.3 Interacting with Components

QML provides a series of items called controls that allow users to interact with their applications. Control items such as buttons, menus, and sliders can be added to applications in the same way as other basic items and possess their own properties and inherited properties [15]. Unlike basic items, controls provide slots for users to add Javascript which gets executed when the control is interacted with.

Figure 2.4 presents a button control item with text "press". When the button is clicked, the `onClicked` slot is triggered and the inner Javascript is executed. In this case, when the button is clicked, the button's x position alternates between 0px and 50px.

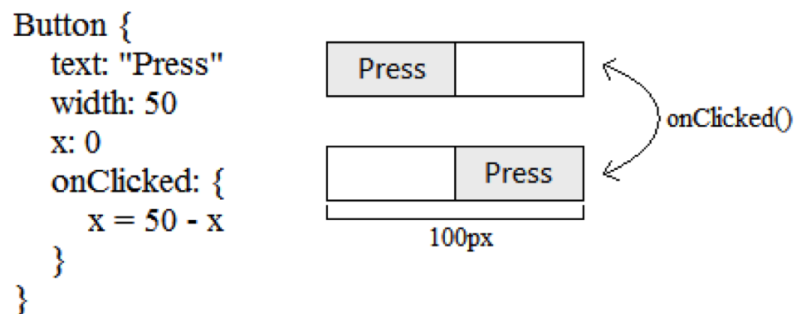


Figure 2.4: A button control item causes itself to change position on click

While the above Figure utilizes the button item, QML also provides a more generic mouse area item that allows users to interact with mouse events without being bound to a button item's look and feel. These mouse areas can be embedded into any item, occupy their own region, and support more mouse interactions than a button.

### 2.3.4 Signals and Connections

Like the Qt API, QML supports a system of signals and slots which allows users to connect different components of their model together. In 2.4 the user defined a slot which captured `onClicked`, a signal defined by the button item [16]. Users can also define their own signals inside items which can then be captured elsewhere in their QML [17].

Figure 2.5 features two items that interact with each other on a button press. The item with the id `myItem` defines a signal named `sendMessage`. When the button inside of `myItem` is clicked, `sendMessage` is called. The connections item featured in Figure 2.5 specifies `myItem` as its target. This allows the user to capture any signals defined inside of `myItem`, such as `sendMessage`. The `sendMessage` signal defines the parameter `msg`, which provides access to that variable inside of the `onSendMessage` slot. In this example, `textitem`'s text field is changed to the value of `msg`.

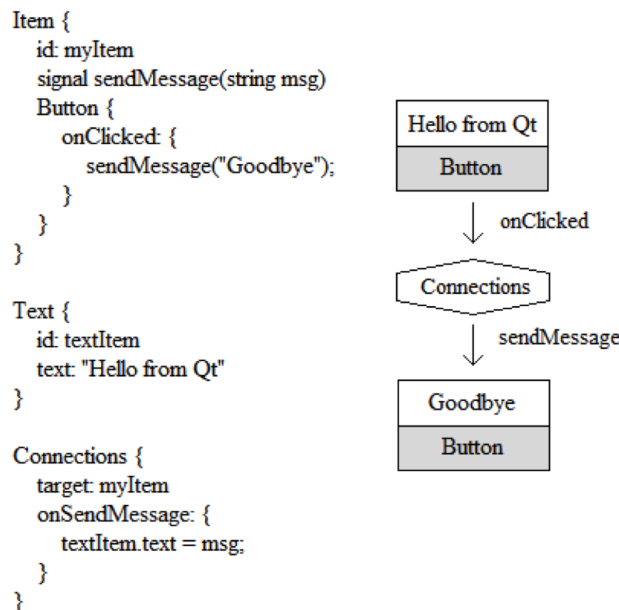


Figure 2.5: Two items interact with each other through the use of signals and slots.

### 2.3.5 Loaders

Applications typically contain multiple components that interact with one another. QML provides a loader item which allows QML models to contain other user-defined models as sub-elements [18]. Control items from separate models can be linked to parent models by embedding a connection item inside of a loader item.

Figure 2.6 portrays a basic example of a loader item being utilized. The file `MainApplication.qml` includes a loader item that loads in the contents of `MyItem.qml`. In this case, the content of `MyItem.qml` is the red rectangle seen in Figure 1. Users can modify the properties of items loaded in by a loader item by accessing the loader's item property. For example, a user wishing to set `red_rect`'s `x` value to 0 inside of the `MainApplication.qml` file would write: `myItemLoader.item.x`

= 0;". MyItemLoader is the id of the loader, item is the property that refers to the actual red\_rect item, and x is the property being modified.

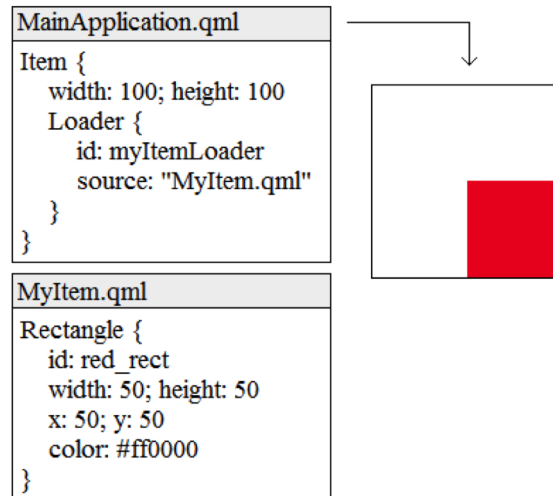


Figure 2.6: QML Loader

### 2.3.6 Animations

Animations are created in applications by modifying properties of items over a given time interval [19]. These property animations can be applied to basic properties of type int, double, color, point, etc. Animations on properties can be defined sequentially and in parallel [20, 21]. Figure 2.7 features an animation defined to have green\_rect move around the viewport clockwise in 1 second. To perform this animation, green\_rect's x property must move to the right, pause, move back to the left, and then pause again. This qualifies as a sequential animation because there are multiple steps involving this single property that must occur in order in order for the animation to look correct. While this x-property animation is running, a sequential animation for green\_rect's y property must also be run with a similar pattern. Since these two separate animations must be run simultaneously, they are bundled under a parallel animation object

## 2.4 Qt Quick Designer

While Qt applications can be developed using any text editor, the Qt framework provides enough functionality to warrant a development environment of its own.

### 2.4.1 Introduction

Qt Creator, an interactive development environment (IDE) developed by The Qt Company, is the primary tool used to develop Qt applications [22]. Like many common IDEs, Qt Creator provides tools for project management, coding applications, building and running applications, and testing applications. In addition to these features, Qt Creator contains a plugin infrastructure that allows developers to add additional functionality to the IDE. One such plugin is Qt Quick Designer.

```

Rectangle {
  id: green_rect
  x: 0; y: 0
  width: 50; height: 50
  color: green
  ParallelAnimation {
    loops: Animation.Infinite
    SequentialAnimation on green_rect.x {
      NumberAnimation { to: 50; easing.type: Easing.Linear; duration: 250 }
      PauseAnimation { duration: 250 }
      NumberAnimation { to: 0; easing.type: Easing.Linear; duration: 250 }
      PauseAnimation { duration: 250 }
    }
    SequentialAnimation on green_rect.y {
      PauseAnimation { duration: 250 }
      NumberAnimation { to: 50; easing.type: Easing.Linear; duration: 250 }
      PauseAnimation { duration: 250 }
      NumberAnimation { to: 0; easing.type: Easing.Linear; duration: 250 }
    }
  }
}

```

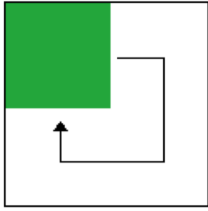


Figure 2.7: Animation Item in QML

Qt Quick Designer facilitates the application UI development process by providing a graphical context for users to design applications [23]. When items are added, modified, or removed in Qt Quick Designer, the corresponding changes are reflected in the QML file that is currently loaded. By providing this graphical context for designing applications, The Qt Company has successfully separated the responsibilities of programmers and graphic designers by removing the need for graphic designers to write QML code.

## 2.4.2 Document Manager and Model Node Structure

The document manager is in charge of generating an editable Qt model from a QML file. In order to link the standard Qt model defined in the QML file to a version usable by Qt Quick Designer, the document manager employs the use of an abstract syntax model tree. This allows users to modify the models without directly modifying the QML file. Once the generated model has been detached by the document manager, a tool called a rewriter propagates these changes back to the QML document.

The intermediary objects defined to represent items in a model are called model nodes. Each model node has knowledge of the properties and child nodes of the item it represents. When a model node is removed from the model, all of its children are also removed entirely from the model. Since model nodes are abstraction tools, end users such as graphic designers are not aware of their existence as an abstraction layer; that being said, model nodes play a large role in how our timeline tool interacts with loaded models.

### 2.4.3 View Manager

The view manager is in charge of rendering Qt Quick Designer's scene along with handling the environmental context of it. The view manager does this by keeping track of the model loaded by the document manager and calling signals to its components to keep each component up to date with consistent experience throughout the scene. In addition to tracking the model, the view manager also monitors the file path of the current model loaded; if the file path changes, the view manager prompts the document manager to load a new model.

The view manager employs a series of components in order to build Qt Quick Designer's interface. While these components are defined with their own style and interfaces, the view manager still plays a large role in the functionality of its components. Specifically, the view manager registers component's basic information such as title and preferred layout position, sets theming parameters, and handles loading and unloading components. By supplying these functionalities, components can be added and formatted to the view manager without having to write any code inside of the view manager.

### 2.4.4 Components

While the view manager contains many components that provide functionality to Qt Quick Designer, only a few are important in the context of this project. Three components that are strongly linked to this project are the navigator, property editor, and form editor.

**Navigator** The navigator component, as seen in Figure 2.8, is a core component of the Qt Quick Designer interface that allows users to view items that are present in the loaded QML file. When items are added, removed, or reparented, these changes are reflected accordingly in the navigator. In addition to displaying the items, the relationships between items is clearly portrayed; Child items appear directly below their parent item with a slight indent. When an item is selected in the navigator, the full set of the item's properties is available to view and modify in the property editor component.

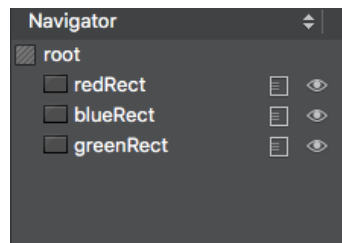


Figure 2.8: The Navigator Component

**Property Editor** The property editor component, seen in Figure 2.9 provides an interface for designers to set property values for the items included in their model. The property editor supports item-property inheritance and will display all available properties that can be set for an item. When an property is modified in the property editor, this change is automatically reflected in form editor component.



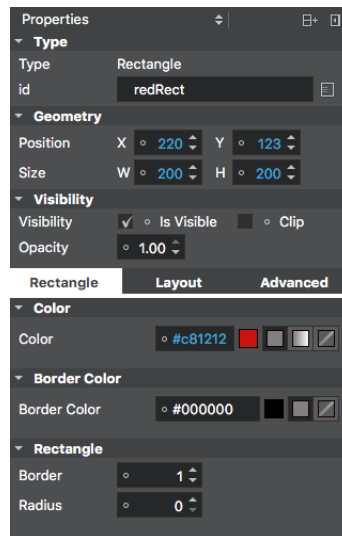


Figure 2.9: The Property Editor Component

**Form Editor** In order to graphically modify any items present in an application, users must directly interact with the form editor component. The form editor, seen in Figure 2.10 is the component in Qt Quick Designer that is responsible for displaying the items in a graphical context. By interacting with items in the form editor, graphic designers can change the positions and dimensions of items without having to code or manually enter the item's property values in the property editor. When items are added or removed in the form editor, the items are added to and removed from the navigator component accordingly. The same functionality applies when items are added to and removed from the navigator.

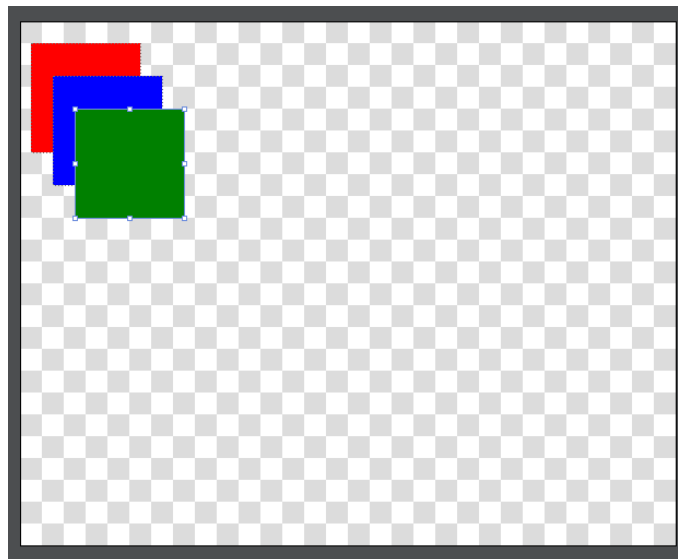


Figure 2.10: The Form Editor Component

## 2.5 Related Works

We looked at existing animation timeline technologies to provide us with a detailed scope of implementable features. We focused on tools that supported HTML5 animations since their scope is similar to QML animations. Four animation tools we used as models for our timeline are Hype, Google Web Designer, Qt 3D Studio, and Adobe After Effects.

### 2.5.1 Hype

The primary tool this project is inspired by is Hype by Tumult. Hype is an HTML5 animation tool with a large feature set and a timeline driven interface. As shown in Figure 2.11 Its primary interface is divided into three primary windows: a title bar window with animation control buttons, a clock, and a zoom feature, a navigator window with items and their properties, and an animation timeline window which displays keyframes.

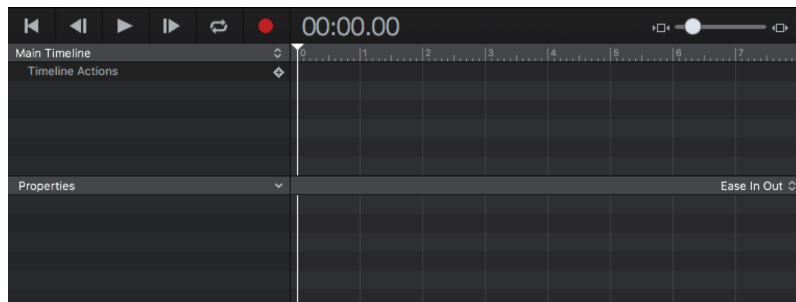


Figure 2.11: Hype Interface

One key feature of Hype is its separation of keyframes by property, Figure 2.12. The navigator window displays all items being animated alongside the specific properties of those items that are changed. The timeline window displays a group keyframe representing the overall changes between keyframes, while simultaneously displaying individual keyframes adjacent to the properties that are listed in the navigator window.

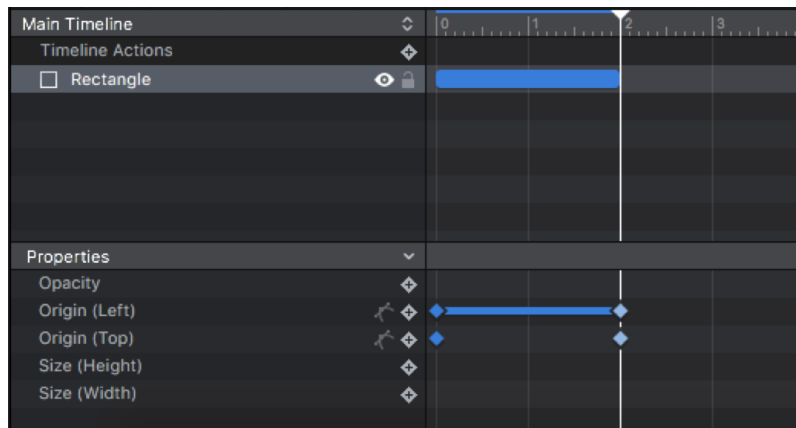


Figure 2.12: Hype, separation of keyframes by item property

Since items can, in some cases, possess over one hundred animatable properties, clutter in the navigator is a feature that needs to be handled delicately. Hype provides a clean solution to this problem by introducing a separate window including all item properties that can then be added to the animation. See Figure 2.13

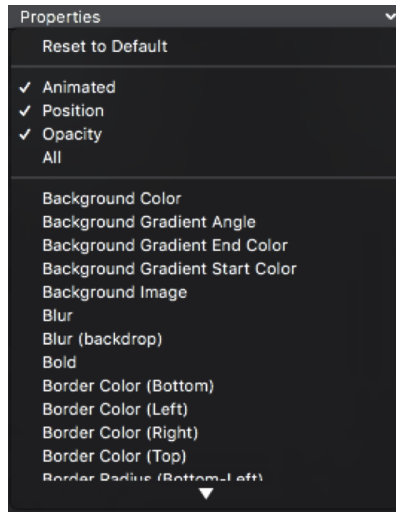


Figure 2.13: Hype property list

## 2.5.2 Google Web Designer

Google Web Designer is an HTML5 and CSS3 animation editor that presents interesting methods of abstracting and presenting animations. As flash animations started to become obsolete, it was necessary for other tools to adopt use cases that Adobe's flash editor originally provided. Google Web Designer provides many components that are similar in nature to Qt Quick Designer's existing layout, thus making it a compelling source of inspiration for this project.

One particular feature in Google Web Designer is the timeline's zoom bar which is seen in Figure 2.14. By being placed in the timeline's main toolbar as opposed to the property editor toolbar in Hype, the zoom functionality possesses a more natural feel.

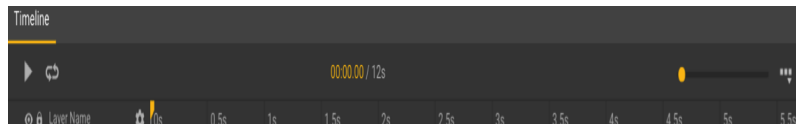


Figure 2.14: Google Designer's toolbar

One key difference between Hype and Google Web Designer is how each application handles animation interpolation. While Hype uses one single interpolation type per timeline, Google Web Design ties interpolation to its keyframes allowing for finer modularity. This aligns well with QML Animations, which also support interpolation types in a per animation basis as opposed to an overall timeline basis, as seen in Figure 2.15.

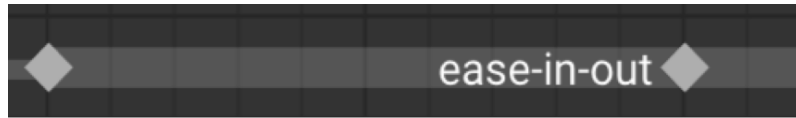


Figure 2.15: Google Designer’s animation interpolation

### 2.5.3 Qt 3D Studio

During the development of this project, The Qt Company received a code donation from NVidia called NVidia Drive Design, which later was redesigned and rebranded to Qt 3D Studio [24]. In order to keep this project’s look and feel consistent with the Qt 3D Studio team’s ambitions, the overall design and scope of this project is consistent with Qt 3D Studio, which is shown in Figure 2.16.

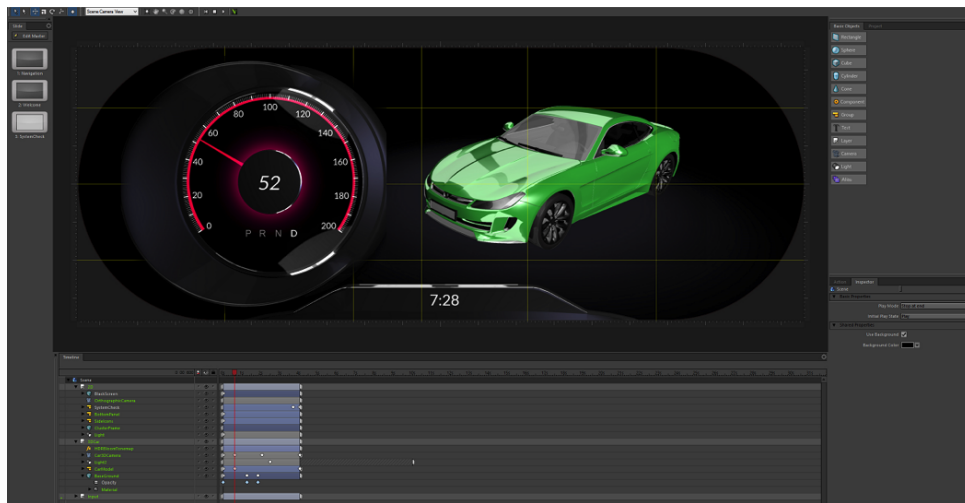


Figure 2.16: Qt 3D Studio

Qt 3D Studio’s primary purpose is to animate 3D scenes, and remains separate from Qt Quick Designer, so while this tool and the 2D timeline tool are two distinct tools, making sure user experiences for both tools was a major priority. One feature implemented by this project that was inspired by Qt 3D Studio is the presentation of properties as a collapsible submenu.

### 2.5.4 Adobe After Effects

Adobe After Effects, by Adobe Systems, is considered the industry standard of video animation software. While this tool is a video animation application as opposed to a dedicated HTML or markup-based language tool, many components of Adobe After Effects are relevant to this project. Figure 2.17

One example of an inspiration stemmed from Adobe After Effects is the way keyframe properties are handled, see Figure 2.18. In addition to offering a dedicated panel for the manipulation of specific properties, it also offers a simple design to allow for quick fine tuning of a property at a specific time. An additional convenience feature provided by this property view is the ability

to jump to the next keyframe in the property's timeline, which adds more precision to keyframe manipulation.

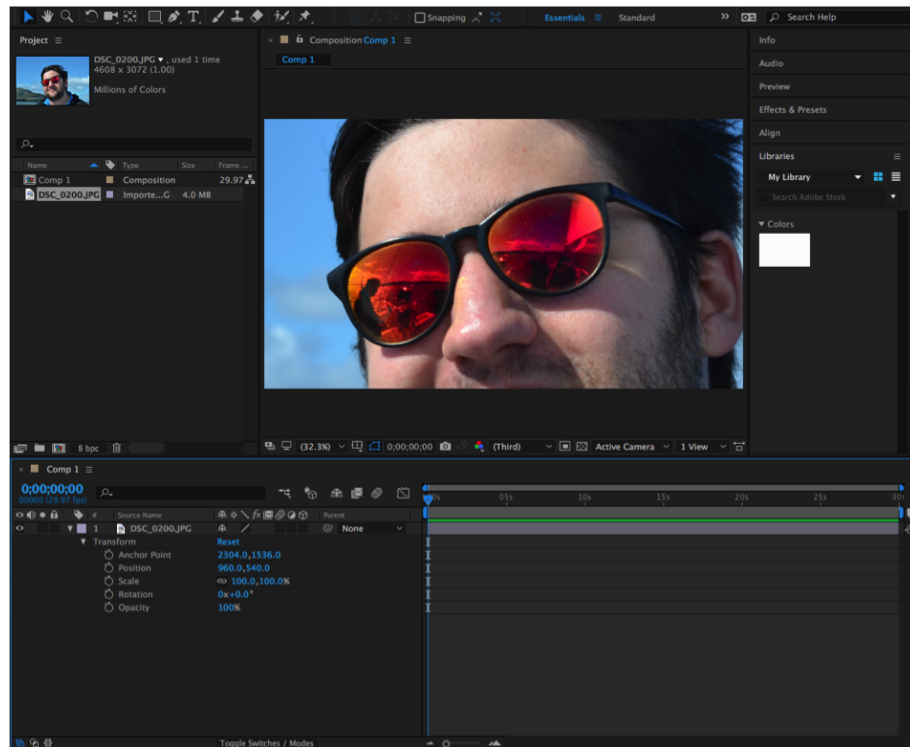


Figure 2.17: Adobe After Effects interface

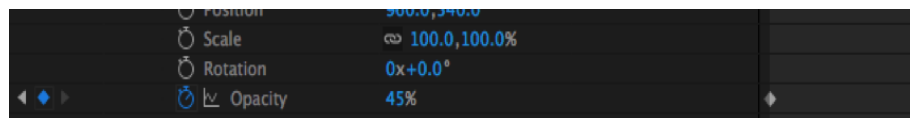


Figure 2.18: Adobe After Effects keyframe properties

## Chapter 3

# Methodology and Implementation

The primary objective of this project is to devise a tool for creating animations graphically in Qt Quick Designer. In order to do this, we opted to model our tool in the form of a timeline. In the scope of this project, a timeline is defined as a set of animations applied to a series of objects over a discrete period of time.

### 3.1 Requirements

During the conception of this project, we established several baseline requirements for our timeline tool. The first requirement states that the timeline tool must provide an organized way to animate items. While it is important that the tool provides basic functionality for creating animations, it is critical that the method which users create animations is both simple and concise.

A second requirement for this project is that the tool must provide functionality for modifying multiple timelines. In many cases designers will want to create separate animations involving elements of the same model depending on what actions are made. The ability to create multiple timelines for a single model is a core component of any animation tool, and thus must be included in the scope of this project.

In addition to the above requirements, our tool must support functionality for creating animations on a per-item, per-property basis. Some animation tools define animations as changes in state of the entire model. This is suboptimal as in many cases, only certain items in the model will be animated. By including functionality for creating animations with a more modular approach, our tool will be able to load animations faster and provide more explicit details on what items and properties are being animated.

Finally, while considering the above three requirements, our tool must also be supported by existing QML objects. As a rule, users should be able to create their own animations by writing QML code and still be able to load these animations into our tool to edit them graphically. While the graphical interface of our tool does not necessarily have to conform to the exact model in which animations are built in QML, it is important that any animation created using our tool can be exported to QML by using objects document in the official QML API.

## 3.2 Design

A critical component of our tool is the user interface. In order for designers to create animations using our tool, they have to be able to easily comprehend the tool’s user interaction (UI). The following section details our methodology for designing this UI.

### 3.2.1 Wireframes

The first step in designing the UI for the timeline tool was separating our requirements and determining what the best options to convey the timeline would be. Using the related works as inspiration, we determined that the timeline tool needed to include a detailed list of items, a breakdown of their keyframes, and a titlebar for playback and fine tuning. Figure 3.1 shows the proposed layout of the overall Timeline Tool. The keyframe area is the core feature of our tool, so it is the largest element of the scene, followed by the navigator area and the titlebar.

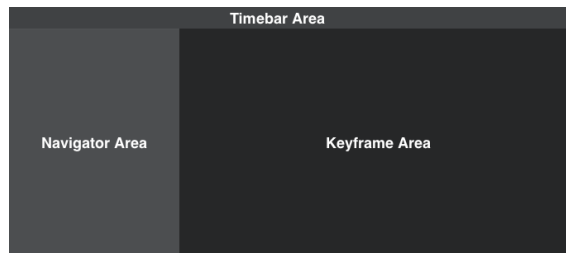


Figure 3.1: The Main Layout of the Timeline Tool

#### 3.2.1.1 Keyframe Area Wireframing

We opted to graphically present animations in the form of traditional keyframes. Because of the amount of space that keyframes occupy on a timeline, the keyframe area is the largest asset of the timeline tool.

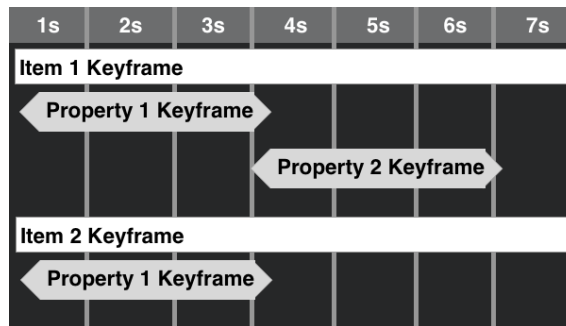


Figure 3.2: The components of the Keyframe Area

In order to divide keyframes by the item’s they are animating, we developed two structures: group keyframes and property keyframes. We decided that the group keyframe would be a large rectangle and placed above the property keyframes, while the property keyframes would be diamond shaped and placed below. Additionally, our UI needs to provide spatial context for the users so

that the position in time of the keyframes is easily distinguishable. To accommodate this need, we added a ruler, tickmarks, and a time indicator to the area.

### 3.2.1.2 Navigator Wireframing

The navigator component of our tool is responsible for providing all functionality that does not involve manipulating keyframes. Accordingly, every item related feature is included in the navigator area.

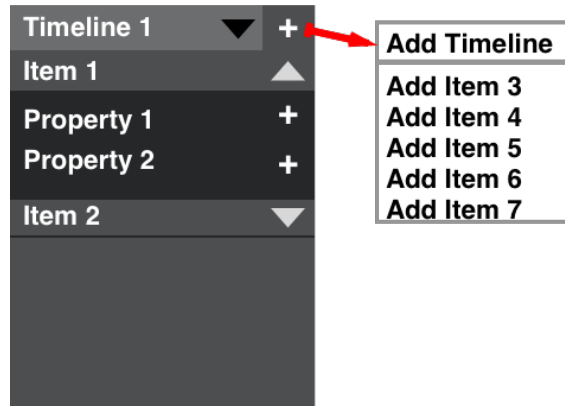


Figure 3.3: The navigator area wireframe

The first task in designing the navigator was determining how to portray multiple timelines, as the particular items that are animated vary per timeline. We decided that a collapsible drop down menu that contains the list of loaded timelines was the simplest choice. A big advantage of choosing this implementation is that now the navigator only has to display one timeline's information at a time.

The second task in designing the navigator was determining how to handle the addition of new timelines and items into the scene. We settled on a button with a plus icon that when clicked would open a menu that presents options for adding a new timeline or item, as seen in Figure 3.3. This implementation provides the benefit of being easily expandable should additional features be added that involve the addition of elements to the scene.

Finally, we had to determine a way to display the items and their related properties. We chose to use a label to display each item and a collapsible menu to list each item's properties. Having the properties detailed in this way allowed us to add button for adding keyframes, fulfilling the requirement of being able to add keyframes to the keyframe area.

### 3.2.1.3 Titlebar Wireframing

We determined that the titlebar, as a component of our tool, would provide functionality for convenience and precision related features. As seen in Figure 3.4, the titlebar includes playback icons that allow users to jump around the timeline. It also contains a clock, which can be edited to jump to a specific time in the timeline. Finally, the title bar includes a zoom slider. This slider allows users to drag keyframes with more precision.





Figure 3.4: The titlebar wireframe

### 3.2.2 Interaction Design

In order to provide functionality for all timeline related features, many components of our design needed to be interactive.

#### 3.2.2.1 Manipulating Keyframes

**Adding Keyframes** Since keyframes are specific to individual properties, the button to add new keyframes exists next to each property in the navigator's property list. Since the functionality of the button adds a keyframe to the keyframe area, each button is placed on the far right of the navigator so that they are adjacent to the keyframe area. Figure 3.5 displays the keyframe area before and after the button is pressed.



Figure 3.5: A keyframe being added to the keyframe area

**Moving Keyframes around** When moving keyframes, we had to consider three possible use cases: changing the start time, changing the duration, and changing both the start time and duration. We addressed these use cases by allowing the start keyframe, end keyframe, and the transition inbetween to be draggable.

We decided to associate changing just the start time with dragging the transition inbetween. By dragging the transition, both keyframes as well as the transition will follow the mouse, thus changing the start time. Figure 3.6 shows this interaction with detail.

By dragging the starting keyframe, the user can modify both the start time and duration of the keyframe pair. Dragging the second keyframe changes only the duration. This functionality can be seen in Figure 3.7.



Figure 3.6: Moving keyframe by grabbing whole keyframe



Figure 3.7: Moving keyframe by dragging keyframe handles

**Coupling Keyframes** One caveat of having draggable keyframes and the ability to have durations that reach zero seconds is that the user might make the keyframes overlap, resulting in the keyframes being stuck together making splitting them up difficult. We designed our tool so that when this situation arises, the keyframe handles are stacked vertically in the same space, which is demonstrated in Figure 3.8. Clicking on the top handle splits the keyframe in two and displaces the end handle to the right. Clicking and dragging the bottom handle moves the combined keyframe without causing a split.



Figure 3.8: Coupling keyframes

**Editing Keyframe Values** While dragging keyframes provides functionality for changing the start time and duration of keyframe pairs, it does not cover changing the actual values that these keyframes modify. We propose showing a dialog window, as shown in Figure 3.9, that contains

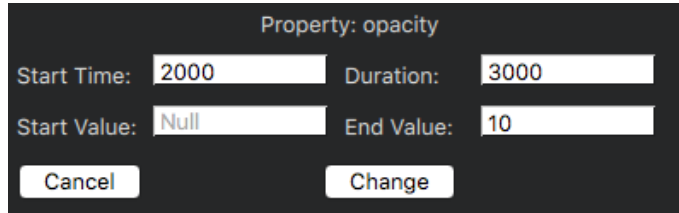


Figure 3.9: Dialog for modifying keyframe values

text fields for a keyframe's values when a user right clicks on said keyframe.

### 3.2.2.2 Traveling in Time

A core component of our keyframe area is representing the current time of the animation with an indicator. Our design offers three methods of interacting with this indicator

**Clicking on the Ruler** The keyframe area ruler contains an indicator for the current time along with tickmarks. Clicking on these tickmarks will set the current time to the tick that was clicked. In addition, dragging the mouse over the ruler is also supported and will set the current time to the the position of the mouse as the user drags. This interaction is highlighted in Figure 3.10.

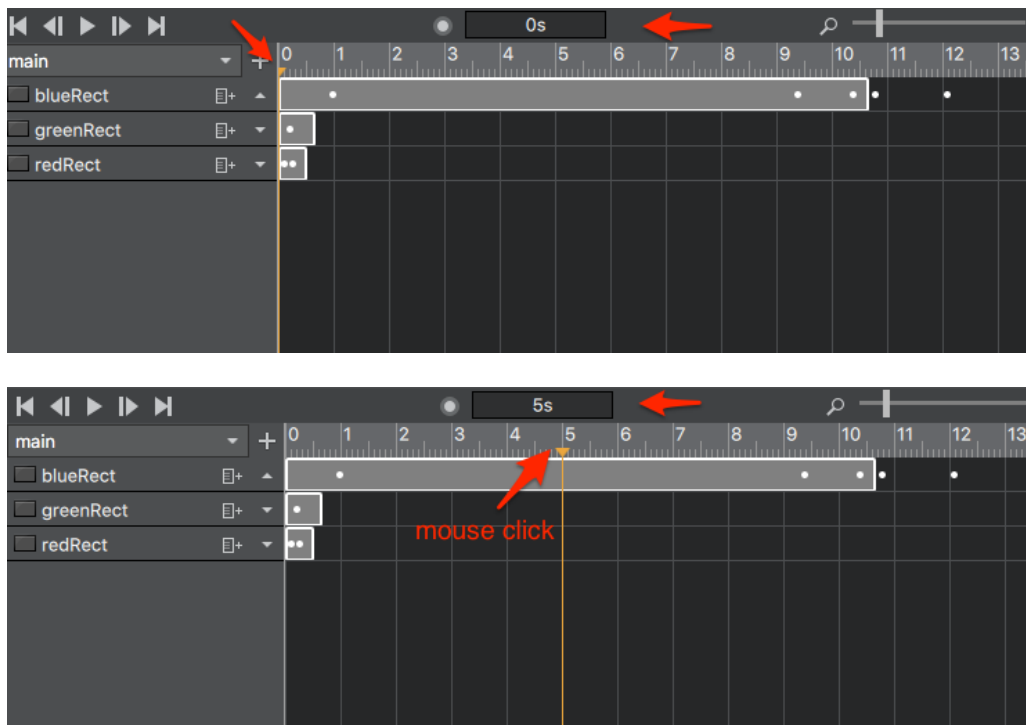


Figure 3.10: Ruler mouse click and current time change

**The Playback Buttons** Most animation tools we looked at for inspiration included buttons to allow users to increment the time without manually clicking on the ruler. We implemented buttons for jumping to the beginning and end of a timeline, stepping forwards and backwards through tick marks, and playing the animation. These buttons are shown in Figure 3.11.



Figure 3.11: The Playback Buttons

**The Clock Control** Finally, we present the most precise way to set the current time, a clock control. The clock control shows the current time in seconds to the user, along with providing a way to input a specific time in text. Clicking on the control toggles the editing mode. This functionality is outlined in Figure 3.12.



Figure 3.12: Clock control's interaction

### 3.2.2.3 Manipulating Timelines

**Switching timelines** Users can switch which timeline is currently loaded by selecting their timeline of choice in the navigator's dropdown menu. When a new timeline is selected, the navigator's item list and property lists should change to reflect the new timeline. This is seen in Figure 3.13.

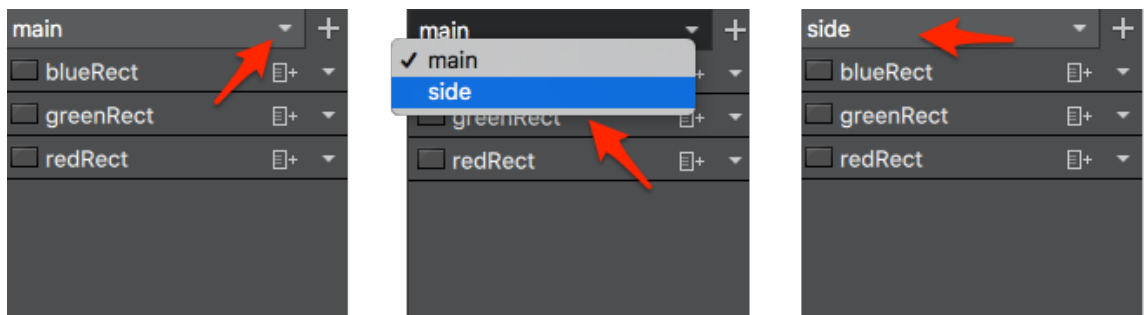


Figure 3.13: Process of changing timeline in navigator

**Adding timelines** In order to add timelines, users can click on a button that opens a dialog box. This dialog box provides a field for users to enter the name of their new timeline, as seen in Figure 3.14.

### 3.2.2.4 Manipulating Items

**Adding Items to the Timeline** The same button that allows users to add new timelines also provides functionality for adding items to the active timeline. The menu that opens when clicking

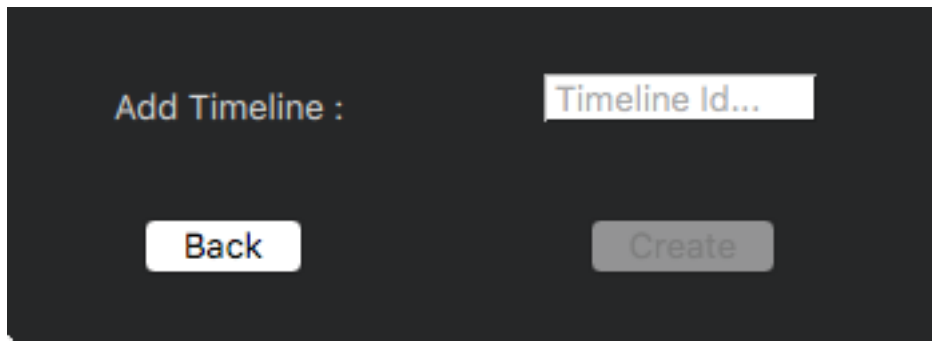


Figure 3.14: Add timeline dialog

on the button will provide a list of items not yet added to the timeline for users to choose from, which is illustrated in Figure 3.3.

**Adding Properties to Items** The final interaction in regards to items is adding properties to be animated to a timeline item in the navigator. This is implemented via a button next to each properties name. After the pressing this button, a list of properties is presented for the user to choose from. After selecting a property, that property will be added to the list of properties below the specified item in the navigator. This feature is highlighted in Figure 3.15.

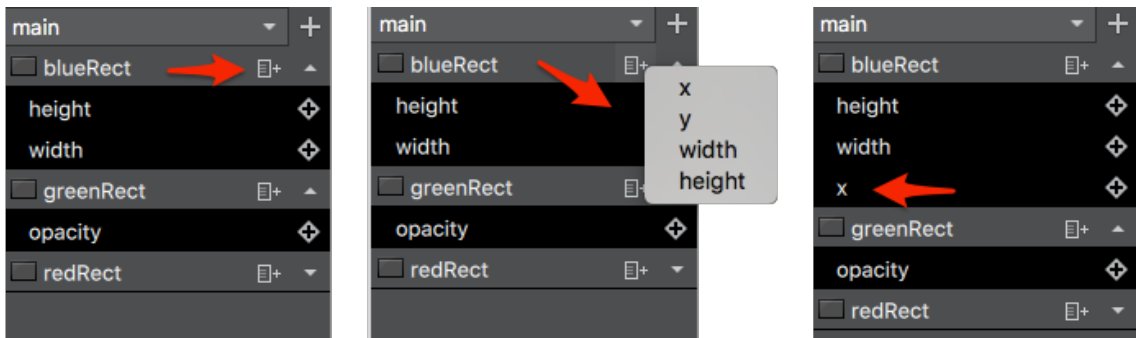


Figure 3.15: Add property to an item in Navigator

### 3.3 Interfacing with Qt Quick Designer

The initial step in implementing this project was finding a way to integrate our project into the existing Qt Quick Designer's codebase. Qt Quick Designer's ViewManager class was created to be modular and allows the addition of new components with the addition of only a few lines of code. The following section covers the steps necessary to interface with the entirety of Qt Quick Designer's view and document managers.

#### 3.3.1 Creating a Component

The minimum requirement for a Qt Quick Designer component is for it to have a view class which directly interfaces with the view manager. This view class must also contain a child QWidget object to be rendered. This view must inherit from the AbstractView class, a class that sets up default

slots for signals emitted from the view manager regarding the document and model's state. We accomplished this by modeling our component off of existing components such as the existing navigator component.

In order to actually register our view with the view manager, we had to implement a `widgetInfo()` function, a function that provides basic information on our component. As seen in Figure 3.16, this function provides the view manager with the widget's instance, the name of the component, our component's preferred location in the interface, and additional meta information for creating toolbars which were not required for this project.

```
WidgetInfo TimelineView::widgetInfo()
{
    return createWidgetInfo(
m_widget, // Widget instance
0, // Toolbar Data
QStringLiteral("Timeline"), // Unique ID
WidgetInfo::BottomPane, // Location Preference
0, // Placement Priority
tr("Timeline Editor")); // Component Name
}
```

Figure 3.16: The `widgetInfo` function

### 3.3.2 Registering the Component

While the `widgetInfo()` function abstracts our component's information from the view, there is no signal to call to attach a component to the view manager. Instead the components that Qt Quick Designer is comprised of need to be manually added by modifying the view manager's source code. The view manager does not directly keep track of its attached components, rather it simply registers views to the document model through a function named `attachViewsExceptRewriterAndComponetView`, as seen in Figure 3.17.

```
void ViewManager::attachViewsExceptRewriterAndComponetView()
{
    ...
currentModel()->attachView(&d->timelineView); // Our Newly Added Component
    ...
}
```

Figure 3.17: Additions to the `attachViewsExceptRewriterAndComponetView` function

Once our view is attached to Qt Quick Designer, the view manager has to factor in our widget's meta info to determine where our view is placed in Qt Quick Designer's interface. The view manager handles this task by passing our widget's info to it's own internal Qt Quick Designer widget. This internal widget then parses through the widget info of every Qt Quick Designer component, and places the components in their preferred locations. This hierarchy is portayed by Figure 3.18

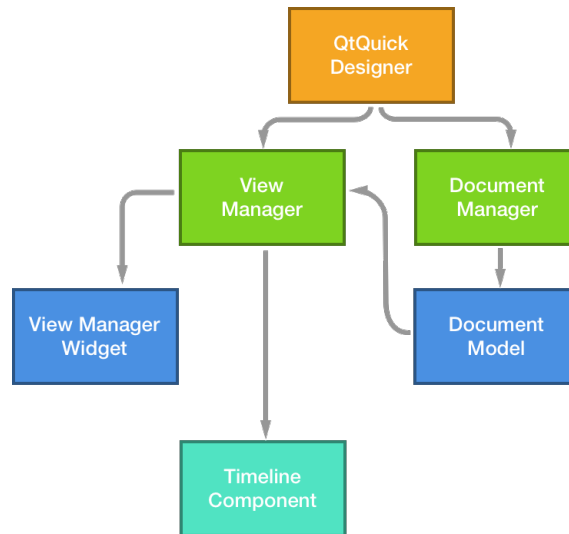


Figure 3.18: Our Timeline Component’s placement in the Qt Quick Designer Hierarchy

One of our design requirements was that the Timeline Component should be placed directly below the Form Editor Component, so we set our components location preference to be the bottom pane. Qt Quick Designer’s internal widget did not have a way to recognize this preference, despite the bottom pane being enumerated along with all of the other possible positions. In order to place our widget at the bottom of Qt Quick Designer, we created a function inside of the internal widget which modified the center area to include a bottom pane. This change allowed our newly created component to finally appear in the scene.

### 3.3.3 Linking the Document Model to Components

After the components of Qt Quick Designer are registered to the view manager, the model defined in the loaded QML file is broadcasted to all of the components via a "modelAttached" signal. Since our timeline tool is closely related to the model, we implemented a slot to pick up this signal. This allowed us to connect all of the components of our tool to the document model and provided us with the tools necessary to implement a robust C++ backend.

### 3.3.4 Loading QML in a Component

While a standard QWidget is the baseline requirement for a view that can be rendered as a component in Qt Quick Designer’s scene, we chose to implement a QQuickWidget to draw our project’s interface. This QQuickWidget creates a QML context along with providing bindings to variables from C++ to QML. We felt that QML was the correct choice in implementing the interface as it allowed us to easily interact with our model.

In order to load QML files of our choice we created a function that links QML source files to our tool’s resource folder, and then to our QQuickWidget. To aid in the development of this project, we made this function a public slot and connected it to a keyboard input signal, thus allowing us to reload our view’s QML sources without having to recompile the C++ binaries. An abbreviated version of this function can be seen in Figure 3.19.

```

void TimelineWidget::reloadQmlSource()
{
    QString timelineQmlFilePath = "/QtQuick/timeline.qml"; //file path
    setSource(QUrl::fromLocalFile(timelineQmlFilePath)); //set widget's source
    emit qmlReloaded(); //update the widget
}

```

Figure 3.19: Function which reloads QML

Finally, to provide the document model to our QML frontend, we first converted the root item of the model to an `AbstractModelItem`, an object that can be interpreted by QML as an associative array. After doing this, we linked the abstract model item to an environmental context variable that can be accessed globally inside any QML source that we include in our widget.

## 3.4 Developing a Timeline Model

The primary goal of this project is the development of a tool that removes the burden of writing QML code to design Qt animations. In order to accomplish this task, we developed an abstraction layer that allowed us to easily connect backend QML to a frontend UI. This abstraction layer took the form of a Qt Model defined in C++ that can be ported to QML for display.

### 3.4.1 Model Requirements

In order to create a successful abstraction layer, our model had to comply with two main requirements. The first requirement is that any model we create must be easily understood and usable in a graphical context by designers looking to design animations. The second requirement is that any model we create must be easily portable to a QML schema.

### 3.4.2 Defining a Timeline Schema in QML

The first step to defining a schema for our timeline was breaking down what our timeline represents. In our case, a timeline is a series of series of items with properties that are animated both sequentially and in parallel. A timeline can be triggered by multiple events, and multiple timelines can exist that include different items or properties being animated. Additionally, a single property belonging to one item can be animated multiple times in one timeline with pauses filling gaps between animations.

To accommodate this definition of a timeline, we defined the following schema. At the lowest level, a sequential animation is used to represent a single property of an item being animated. This sequential animation includes `PauseAnimations` and `PropertyAnimations` dictating the specific changes of that property, in order, throughout the timeline. The sequential animations are then grouped by their parent item under parallel animation objects, which will run each property's animation in parallel. Finally each of these item's parallel animations are bundled under a root parallel animation which represents the timeline as a whole.

Figure 3.20 presents QML source code formatted in our schema. In this example, a timeline named `timeline_1` animates two items in parallel: `item_1` and `item_2`. First `item_1`'s `y` value is



changed from 0 to 10 over 1 second. At the same time, item\_2's x value is changed from 30 to 40. After one second, item\_1's x value is changed from 70 to 80 over one second.

```
// Timeline Tag
ParallelAnimation {
  id: timeline_1
  // Item 1 animations
  ParallelAnimation {
    // X property animation
    SequentialAnimation {
      PauseAnimation { duration: 1000 }
      PropertyAnimation {
        target: item_1; property: x;
        duration: 1000; from: 70; to: 80
      }
    }
    // y property animation
    SequentialAnimation {
      PropertyAnimation {
        target: item_1; property: y;
        duration: 1000; from: 0; to: 10
      }
    }
  }
}

// Item 2 animations
ParallelAnimation {
  // X property animation
  SequentialAnimation {
    PropertyAnimation {
      target: item_2; property: x;
      duration: 1000; from: 30; to: 40
    }
  }
}
}
```

Figure 3.20: This schema represents a basic timeline that animates two items.

### 3.4.3 Defining the Model Structure

Based on our research, keyframe objects are the unanimous method of choice for defining animations in timeline tools. A single keyframe represents the state of the properties of an item at a specific time in an animation. While Qt animations are primarily defined by transitions and not states, we determined that the use of keyframes as a basis for our model satisfies both our usability and portability requirements.

Accordingly, the lowest level element of our model is defined as a property keyframe pair. This item closely resembles the structure of a property animation object, but in the context of two keyframes. Each property keyframe pair represents a single state change of a single property belonging to a single item. Members of this object include the time of the first keyframe, the first

keyframe's value, the duration of time until the second keyframe, the second keyframe's value, and the property that the property keyframe pair is associated with.

The second, mid-tier element of our model is the timeline item object. This object represents an item in the loaded QML model that contains properties that are being animated in the timeline. Members of this object include the id of the item being represented, the type of item being represented, and a map linking property names to lists of property keyframe pairs. When new keyframes are generated, they are added to the corresponding timeline item in our model.

The highest level element of our model is the timeline model object. This object acts as a link between QML designs and C++ implementations, and contains functions that provide high level interactions with the model as a whole. These functions range from adding new items to be animated to updating the qml implementation of the model in the view. The primary member of this object is a list of timeline items included in the current timeline animation.

### 3.4.4 Separating the Model from the View

To provide abstraction between the timeline model and the view, we developed a backend layer dedicated to handling interactions between the two. This timeline backend is initialized in the view, and interacts with the model through public functions called in the view and the widget. The backend also provides slots that are linked to signals emitted from the view's QML component. The hierarchy of these components and their interactions is portrayed in Figure 3.21.

The timeline backend's primary purpose is to reflect changes in the model to the view and vice versa. In order to accomplish this task, the backend provides functionality for constructing new models, modifying existing models, exporting models to the defined QML schema, handling interactions in the view, and updating the view when the model is changed.

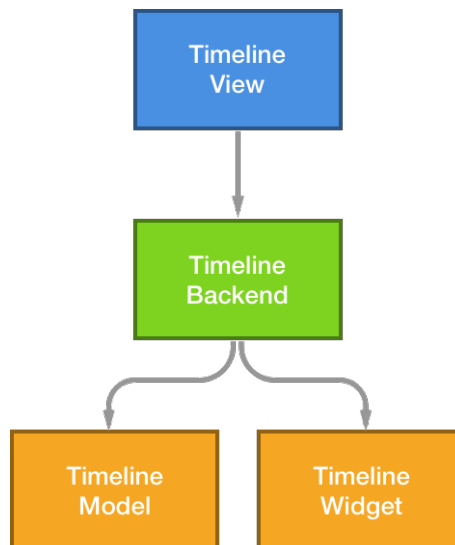


Figure 3.21: The final component architecture

### 3.4.5 Linking the Model to QML

A critical requirement of all Model-View systems is the synchronization of both the model and the view at all times. In the context of this project, we had to ensure that all data related to timelines was consistently passed from our C++ model to the QML in our view. In addition, any changes of the timeline in our view had to be reflected in our C++ model. We accomplished this task by using two features provided by the Qt framework: QProperties and context variables.

QProperties are members of QObjects that can be read and written inside QML source code. By establishing property keyframe pairs, our timeline model object, and our timeline backend as QObjects, it became possible to pass members of those objects to our QML view as QProperties with both read and write access. Some model properties passed as QProperties are keyframe start times, durations, start values, and end values.

In cases where information that was not explicitly a member of our model needed to be passed to the view, we passed them as context variables. Context variables are a form of environment variable belonging to a widget that can be globally accessed inside of QML source files. These variables are added in C++ code by linking a string name in the widget's context to any QObject. Some objects passed to the view as context variables are a list of timelines that can be loaded from the document model, a list of items that can be added to the active timeline, and the current time selected inside of the timeline area.

## 3.5 Developing the Timeline Navigator

Even though a navigator component already exists in Qt Quick Designer, a separate navigator tailored to our timeline is needed to provide functionality such as adding and selecting timelines, adding items to timelines, and adding properties to timeline items. While developing the navigator, we aimed to implement features based on our schema with the highest level feature, and the first feature we implemented, being adding and selecting timelines.

### 3.5.1 Navigator Requirements

While designing the timeline navigator, we had to abide by three primary requirements. First, the navigator must be feature complete. In the case of our tool, we define feature complete as the inclusion of functions for adding timelines, adding timeline items, adding properties to be animated for timeline items, and maintaining a list of items and properties that are available to be added to a timeline.

The second requirement of our navigator was that its style be similar in nature to the existing Qt Quick Designer navigator component. While the two components are separate in purpose, they belong to the same overall program, Qt Quick Designer, and therefore the user experience for both components must be roughly the same.

The final requirement that dictated the design of our navigator was a matter of ease of use. A shallow learning curve is an important trait of any GUI-based tool, and the timeline navigator component of our tool is no exception.

### 3.5.2 Adding and Selecting Timelines

The first feature we added to the navigator timeline tool was the ability to add and select timelines to be loaded into the tool. We considered this feature to be the highest-level feature of the navigator as the items and properties that are loaded are directly related to which timeline is currently active in the tool.

This feature is implemented in the view via a combo box and a menu button, both of which are adjacent to each other and located at the top of the navigator, the code of which being outlined in Figure 3.22. The combo box is filled with all timelines present in the QML source file loaded in by the document manager. When a new source file is loaded or changes to an existing timeline are exported to QML, the combo box updates to ensure that no new timelines are left out. When an item in the combo box is selected, a signal is sent from the QML view to the C++ backend to set the current timeline to the name of the item selected. The backend then prompts the entire timeline tool to update to reflect the newly selected timeline.

```
RowLayout {
    ComboBox {
        id: timelines
        model: timelineList
        onActivated: {
            navigator.setTimeline(timelines.textAt(index));
        }
    }
    BarButton {
        iconSource: "image://icons/plus"
        onClicked: {
            addMenu.popup();
        }
    }
}
```

Figure 3.22: QML code defining the timeline list and add timeline features

The menu button adjacent to the combo box provides an option for adding timelines. When this option is selected, a dialog box opens that prompts the user to enter a name for the new timeline being created. After entering a name, the user can either press a confirm button to proceed with creating the new timeline or press a cancel button to back out of the creation process. After the creation of a new timeline is confirmed in the dialog box, the same QML signal that is called by the combo box is called with the name of the new timeline as a parameter. The C++ backend then recognizes that the new selected timeline does not exist, generates a new timeline object with the selected name, and updates the combo box to reflect the new list of timelines.

### 3.5.3 Adding and Viewing Timeline Items

The second feature we added to the navigator was functionality for registering timeline items to the timeline model and viewing timeline items in the navigator. In order to fulfill our requirement of maintaining a constant user experience between the use of the standard navigator component and

the timeline navigator component, we opted to display timeline items in a vertical list view directly below the combo box containing the name of the active timeline. This list view is repopulated by the C++ backend whenever a new timeline is loaded or a new item is added.

The menu button adjacent to the timeline combo box provides an option for adding items to the active timeline. When this option is selected, a new menu opens with a list of items that are in the document model that can be added to the timeline. In order to ensure that this menu only includes items that are not already included in the timeline, the C++ backend generates a list and passes it into the QML view as a context variable. When an item is selected in this new menu, a signal is sent to the C++ backend to add that item to the timeline. When this signal is received, the item is added to the timeline model, the context variable which tracks what items have not yet been added is updated and resent to the QML view, and the navigator is updated to reflect the new item in the timeline item list view.

### 3.5.4 Adding and Viewing Timeline Item Properties

The third feature we added to the navigator was functionality for registering animatable properties for timeline items to the timeline model and viewing these properties in the navigator. While most functionality regarding keyframes is relegated to the keyframe area, it is the navigator's duty to define and display which properties are currently available to have keyframes added to.

In order to condense the navigator, property views of timeline items are minimized. Each timeline item in the navigator contains a button to expand the timeline item view to contain a vertical list view of each property that is actively primed for animation. Pressing this button a second time minimizes the timeline item view. This functionality exists solely in the navigator's QML source code.

In addition to the expansion button, the timeline item view also contains a button for adding property views to the respective timeline item. Pressing this button opens a menu displaying a list of all properties that are available to be animated and added to the timeline item view. In the current implementation of this project, only x, y, width, and height properties are supported. When a property is selected in the menu, a signal is sent to the C++ backend to add that property to the corresponding timeline item. The backend proceeds to add a new entry to that timeline item's map for the property specified. Next, the backend updates the navigator to reflect the new property that was added to the view.

## 3.6 Developing the Keyframe Area

The keyframe area is a component of our tool that is strongly linked to the timeline navigator component. The two primary features of this component are functionality for adjust the time indicator and functionality for modifying and interacting with keyframes.

### 3.6.1 Keyframe Area Requirements

In addition to the user experience and ease of use requirements defined by the the timeline navigator, the keyframe area has its own set of unique requirements.

One unique requirement of the keyframe area is the need for an accurate representation of the keyframes in the active timeline model at all times. More specifically, any modifications to the keyframes in the timeline model that are performed in the C++ backend or the QML source must be automatically reflected in the keyframe area view.

In addition to the synchronization of the keyframe area to the model, it is also required that the keyframe area is always synchronized with the timeline navigator. For example, when timeline item properties are expanded and minimized, the corresponding rows in the keyframe area must do the same.

### 3.6.2 Implementing the Ruler

The primary purpose of the ruler is to set a scale for the keyframe area and provide a visual context for time. As seen in Figure 3.2, the ruler resides at the top of the keyframe area and occupies the entire width of the component.

The Ruler was implemented as a QML Repeater item that draws lines at selected intervals that represent tick marks. The height of each tick mark is determined by running the index of the tick through a modular function; tick marks that represent a time divisible by one half of a second are taller than other tick marks. The spacing between the tickmarks is directly linked to the value of the zoom slider which resides inside of the titlebar. This value is propagated to the ruler through a QML variable binding.

### 3.6.3 Keeping Track of Time

The timeline area provides three methods for interacting with the current time index. In order to supply this functionality, the C++ backend hosts an integer time variable that is passed to the QML source as a context variable. Each time interaction feature interacts with this variable in some way to perform its functionality.

The first method of changing the time index is clicking on the ruler at the desired time. To accomplish this, a MouseArea was overlaid on top of the ruler. When this MouseArea is pressed and dragged, the time indicator's x coordinate is changed to the current x value of the mouse. When the mouse is released, a signal is sent to the C++ backend indicating that the time has changed to the current position of the indicator. This change is then reflected in the context variable that is passed to the QML source.

The second method of modifying the time index is by interacting with the buttons in the title bar. Buttons are provided for jumping to the start and end of the timeline, stepping forward and backward by one tick mark, and playing the timeline. After pressing the button to jump to the beginning of the timeline, a signal is sent to the C++ backend to set the time to 0. Pressing the button to jump to the end of the timeline sends a signal to set the time index to the time of the last keyframe. Pressing the skip forward or backward buttons send a signal to the C++ backend to change the time the current time's closest tick mark plus or minus 100ms respectively. An example of the step back button is seen in Figure 3.23. Functionality for the play button is not currently supported for reasons outlined in the future works section.

```

BarButton {
    iconSource: "image://timeline/step-backwards"
    tooltip: "Step Back"
    onClicked: {
        if (currentTime > 0) {
            titlebar.setCurrentTime((currentTime - currentTime%100) - 100);
        }
    }
}

```

Figure 3.23: The QML source of the step-back button.

The final method of interacting with time is provided by a clock object in the title bar. When the clock is clicked, it is converted to a text field that allows the user to enter a time in milliseconds. After the user presses enter, the text field is converted back to a clock and a signal is sent to the C++ backend to set the time to the time specified by the user. Figure 3.24 illustrates how this item was created. The design of this feature in the view is shown in Figure 3.12.

```

TextField {
    id: editView
    height: 20
    text: time
    visible: edit ? true: false
    horizontalAlignment: TextInput.AlignHCenter
    onEditingFinished: {
        edit= false
        editView.focus = false
        clockControl.setCurrentTime(editView.text)
    }
}

```

Figure 3.24: QML Source code that defines the editing component of the title bar's clock.

### 3.6.4 Keyframe Rows

According to our model, keyframes are specific to individual properties inside individual items. Additionally, any property can be modified by an unlimited amount of keyframes. In order to represent these two aspects of our model, we chose to display all keyframes that belong to the same property of the same item in a single row alongside the name of that property in the navigator. Special rows are designated to fill the space next to the names of items in the navigator by including rows for group keyframes.

Property keyframes rows are filled with a horizontal list of keyframes pairs. The graphical representation of a keyframe pair includes two icons resembling the start keyframe and end keyframe. The transition between each keyframe in a pair is represented by a rectangle with a width equaling the duration of the keyframe pair.

Group keyframe rows contain a single group keyframe item that represents the combination of all property keyframes a timeline item contains. This group keyframe item takes the form of a

rectangle, and has a width equal to the time between the first keyframe and last keyframe. Circles are overlaid on top of the group keyframe item that represent the location in time of all property keyframes possessed by the group keyframe's corresponding timeline item.

### 3.6.5 Adding Keyframes

```
void TimelineQmlBackend::addKeyframe(QString itemId
, QString propertyName
, int time) {
    ...
    TimelineItem *item = m_timelineModel->getItemById(itemId);
    PropertyKeyframePair *keyframe =
    new PropertyKeyframePair( propertyName
    , time,0,startValue
    , startValue
    ,0);
    item->addKeyframe(keyframe); \\add the new keyframe to the item
    ...
}
```

Figure 3.25: The add keyframe slot

While keyframes live in the keyframe area, functionality for adding keyframes is a component of the navigator. Each property item in the navigator contains a button that is used to add a keyframe for that specific property. When this button is pressed, a signal is sent to the C++ backend to request the addition of a keyframe for the specific item/property pair as seen in Figure 3.25. Upon receiving this request, the backend creates a new property keyframe pair linked to the current time index specified in the keyframe area. The backend then proceeds to add the new keyframe to the model and update the keyframe area to reflect the new changes.

### 3.6.6 Keyframe Interactions

In addition to adding keyframes, the timeline tool also offers functionality for modifying aspects of property keyframe pairs by making keyframes in the view interactable. Overall, four properties of keyframe pairs are modifiable: start time, duration, start value, and end value.

The start time and duration properties can be modified by dragging elements of the property keyframe. If the starting keyframe icon is dragged, both the start time and the duration of the keyframe pair will be modified. For example, if the original start time of a keyframe pair is two seconds, the original duration is two seconds, and the starting keyframe of the pair is dragged to the one second mark, the start time of the pair will be changed to one second and the duration will become three seconds. The start time of the keyframe pair can be modified without affecting duration by dragging the bar between the two keyframes. The duration can be changed without affect the start time of the pair by dragging the final keyframe.

As an additional feature, the icon representing the keyframe pair changes when duration becomes 0 to allow for both keyframes to be displayed as opposed to both keyframes being in the same position. This functionality was accomplished by overlaying a mouse area on top of the keyframe



pair and along the keyframe area. When a mouse event is triggered inside of the keyframe area, QML code determines which item was selected, and changes the features of the pair in the QML source accordingly. Once the mouse is released, the property changes are written to the C++ model.

The process of changing the start and end values of keyframe pairs is different. When a keyframe pair is right clicked, a menu will open providing the user an option to modify that keyframe pair's properties. Upon selecting this option, a dialog box will open prompting the user to enter values for the start time, duration, start value, and end value of the keyframe pair.

Finally, the keyframe area provides functionality for expanding and collapsing property keyframes. When a group keyframe is clicked, the property keyframes that are a part of the item the group keyframe represents either expand or collapse based on their current state. Additionally, a signal is sent to the navigator informing the navigator of the item that has been expanded or collapsed so that the navigator can perform the same action.

# Chapter 4

## Results

Overall, our project was a success and was positively received by our direct advisors and general staff at The Qt Company. Figure 4.1 shows the final look of our tool, running inside of Qt Creator. Through the use of our tool, graphic designers can easily create and edit animations using QML without having to write a single line of code.

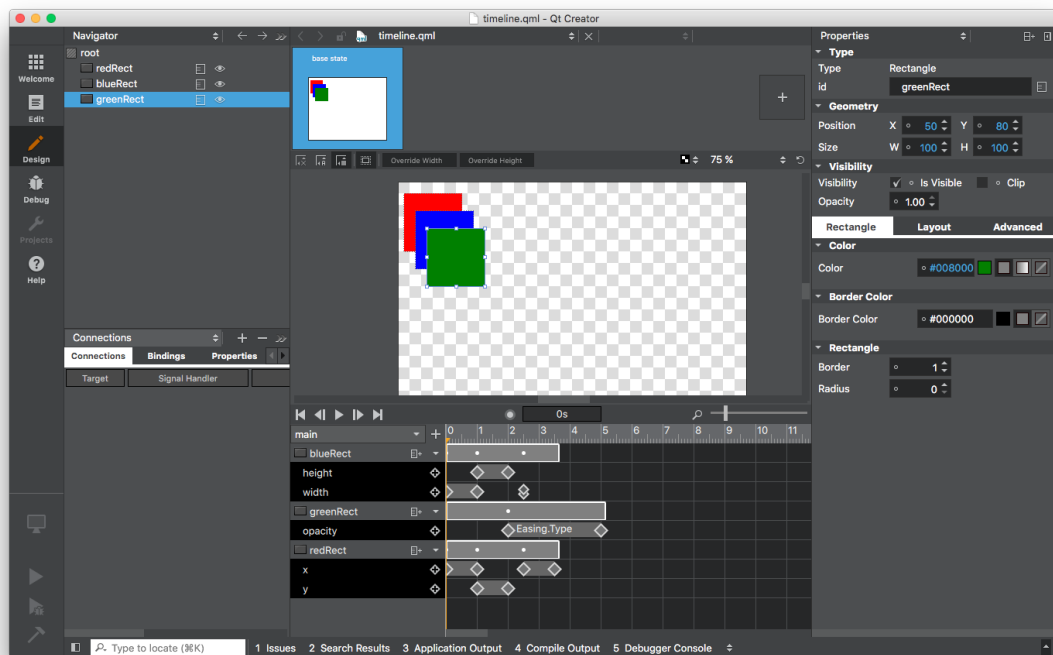


Figure 4.1: The final look of the Timeline Tool

When we began working on this project, we established a series of requirements for our final product; which we met. The first requirement for this project was that our timeline tool must provide an organized way for animating Qt items. We satisfied this requirement in two different ways. First, by developing a simple QML schema for timelines, users can easily browse through the source code of animations our programs generate. Additionally, by designing our timeline tool's

view with easy to grasp, modular components, the learning curve for working with our tool is very shallow.

The second requirement for this project stated that our tool must provide functionality for modifying multiple timelines. Our timeline schema paved the way to supporting multiple timelines by defining a contained timeline inside of a root Parallel Animation. This requirement was fully satisfied by our implementation of a tool in our navigator component to select multiple timelines from the loaded QML source. When a new timeline is selected, the old timeline is saved and the new timeline is loaded into the tool. Finally, when the QML is exported, the multiple timelines get saved individually as described in our timeline schema.

The third requirement for this project highlighted the need for our tool to support the creation of animations on a per-item, per-property basis. By defining a system of property specific keyframes and requiring users to explicitly select which properties of which items they are choosing to animate at any given time, we were able to ensure that timelines were always defined by item-specific and property-specific animations.

The final requirement for this project stated that all animations created using our tool must be able to be exported to QML using existing QML objects. We satisfied this requirement by utilizing the existing parallel and sequential objects as critical components of our QML schema. By designing our timeline's C++ model to be structurally similar to our QML schema, our tool was able to easily convert timelines between both formats.

# Chapter 5

## Future Steps

While the current implementation of our tool is functional and satisfied all of our requirements, there are features that could be added to our tool to provide a more complete user experience. This section outlines features that may be the next steps in advancing the Qt Quick Designer animation editing tool.

### 5.1 Form Editor Animation Playback

In the current implementation of our tool, animations that are created are played by compiling and running the application that the animation is a part of. While this does allow designers to view their animations, it takes a relatively long time and makes modifying animations in a minor way difficult. One of the desired features we wished to implement was for animations to be playable inside Qt Quick Designer's form editor component. As of right now, there are limitations in the Qt framework's current QML and form editor implementations that hinder the development of such a feature.

The first limitation is that the current implementation of QML implementation does not keep track of time. While it is possible to pause and stop animations using signals, it is not possible to specify a specific time of an animation and show the current state of the animation at that point. This feature can be implemented via the development of a function that takes in the start value of an animation, the end value of an animation, and the desired intermediary time and outputs the value at that time by using the animation's interpolation function.

Another limitation is that the form editor was not built around playback, but rather for being able to change an item's properties. Manipulating one of these properties programmatically to feign motion would result in direct changes to the document model, which is undesirable. One workaround to this behavior is to modify a model node's property using the `setCustomProperty` function, which would override a specific property without writing them to QML code. The Qt Designer team advised us to not concentrate in this workaround as it has not been extensively tested.

## 5.2 Keyframe Editing in Property Editor

The current method of editing the values of keyframes in our tool is opening a dialog box by right clicking on a keyframe. Inside the dialog box, a user can modify attributes of the keyframe pair. While this approach is functional, it is not slick and it is not easily scalable to handle additional keyframe attributes.

To address these issues, we recommend that functionality for editing keyframe attributes be linked to the existing property editor component that lives inside of Qt Quick Designer. The property editor, as it stands, provides a simple interface for modifying the properties of items in a model. Additionally, the property editor groups properties into sections which can be expanded and collapsed, thus reducing clutter and making the component easily scalable for any amount of properties that belong to an item.

The existing implementation of the property editor only supports editing properties that belong to items that exist in a document model; this excludes our timeline model and by extension, our keyframe objects. One solution that would allow keyframe properties to be edited in the property editor would be to convert keyframe objects into pseudo document model nodes. By creating skeleton model nodes with keyframe property tags, keyframes could be edited in the property editor component. A less optimal solution would be to modify the property editor so that it can support properties of objects not included in a document model.

## 5.3 Additional Animation Functionality

As of right now, our tool only supports a limited set of features in regards to creating and modifying animations. One such feature is animation easing. In many cases, graphic designers wish to augment the movement of items to create smoother transitions during the beginning and end of animations. The front end of our tool currently provides a section for each keyframe pair that displays the animation's easing type, however the backend of our tool automatically assigns a linear easing to every animation created. We recommend that support for different easing types be added to our tool.

Currently, while the backend of our tool supports all animation types, the frontend of our tool only supports the creation of animations for x, y, width, and height properties of items. We recommend the addition of a feature to the frontend of our tool that allows users to create animations of any type. In order for this to be accomplished, a list of all animatable properties for a given item type needs to be propagated from the backend of our tool to the frontend.

Finally, Qt Quick 5.2 introduced a new type of animation item named Animator. As opposed to a property animation which simply interpolates values in a given amount of time using the CPU, the Animator is executed as a part of Qt's scene graph and is hardware accelerated, making animators faster than their animation counterpart. Unlike property animations, animators have different implementations per property. For example, instead of using a Property Animation on x, one could use an XAnimator instead.

We propose the addition of a simple switch case in our C++ backend which can detect if a specific property which has an Animator counterpart is being animated, and exporting the item as such Animator. Our import function already supports Animators, so not much work has to occur in that regard.

## Chapter 6

# Conclusion

Before this project began, designing animations using the Qt framework required writing code. The end result of this project was the development of a tool that allows graphic designers to create Qt framework animations inside of The Qt Company's IDE, Qt Creator, without having to write a single line of code. The tool is inspired by other leading animation editors which use a keyframe-based system for defining animations. While QML, the frontend language for Qt applications, does not support keyframes, our tool generates valid QML code, while also exposing QML animations in a keyframe-based system.

# Bibliography

- [1] Summerfield M Blanchette J. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006.
- [2] Meet qt - leading cross platform application and ui framework. presentation.
- [3] Qt overview. Available at [www.qt.io](http://www.qt.io). web. accessed 02/28/2017.
- [4] <https://www.qt.io/qt-for-application-development/>. web. accessed 02/28/2017.
- [5] [https://wiki.qt.io/Qt\\_Platform\\_Abstraction](https://wiki.qt.io/Qt_Platform_Abstraction). web. accessed 02/28/2017.
- [6] <http://doc.qt.io/qt-4.8/signalsandslots.html>. web. accessed 02/28/2017.
- [7] <http://doc.qt.io/qt-5/qtwidgets-index.html>. web. accessed 02/28/2017.
- [8] <http://doc.qt.io/qt-5/designer-using-a-ui-file.html>. web. accessed 02/28/2017.
- [9] <http://doc.qt.io/qt-5/model-view-programming.html>. web. accessed 02/28/2017.
- [10] <https://doc.qt.io/archives/4.6/model-view-delegate.html>. web. accessed 02/28/2017.
- [11] <http://doc.qt.io/qt-5/qtwidgets-itemviews-spinboxdelegate-example.html>. web. accessed 02/28/2017.
- [12] <http://doc.qt.io/qt-5/qtqml-index.html>. web. accessed 02/28/2017.
- [13] <http://doc.qt.io/qt-5/qml-qtqml-component.html>. web. accessed 02/28/2017.
- [14] <http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html>. web. accessed 02/28/2017.
- [15] <http://doc.qt.io/qt-5/qtquickcontrols-index.html>. web. accessed 02/28/2017.
- [16] <http://doc.qt.io/qt-5/qtqml-syntax-signals.html>. web. accessed 02/28/2017.
- [17] <http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes>. web. accessed 02/28/2017.
- [18] <http://doc.qt.io/qt-5/qml-qtquick-loader.html>. web. accessed 02/28/2017.
- [19] <http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html>. web. accessed 02/28/2017.
- [20] <http://doc.qt.io/qt-5/qml-qtquick-parallelanimation.html>. web. accessed 02/28/2017.



- [21] <http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html>. web. accessed 02/28/2017.
- [22] <http://doc.qt.io/qtcreator/>. web. accessed 02/28/2017.
- [23] <http://doc.qt.io/qtcreator/creator-using-qt-quick-designer.html>. web. accessed 02/28/2017.
- [24] <http://www.qt.io/qt-news/qt-company-adopts-nvidia-drive-design-studio/>. web. accessed 02/28/2017.

# Appendix A

## Qt for Native Client

### A.1 Introduction

The following section details a project which we worked on during the beginning of our time at The Qt Company that is separate from the Qt Quick Designer timeline tool.

A primary feature of Qt is its cross-platform compatibility. New coding environments and platforms now become readily available; Accordingly, The Qt Company is interested in expanding their platform to ensure that it runs on any and all devices. One such platform is Google's Native Client .

This project follows a previous attempt to port Qt to Native Client, provides context on how to run applications on Native Client using outdated versions of Qt, and outlines further steps needed in order to port a current implementation of Qt to browser-based platforms.

### A.2 Background

#### A.2.1 Native Client

Native Client (NaCl) is a sandboxed environment inside Google Chrome that brings the performance and low level control of native code to a web browser without sacrificing security and portability . Code written for NaCl is first compiled as intermediary low level virtual machine (LLVM) bytecode and is considered a Portable Native Client (PNaCl) executable (.pexe). This platform-specific bytecode is then passed to Google Chrome which in turn converts the LLVM bytecode into a NaCl executable (.nexe). In-browser compilation can be avoided if developers instead compile their code for a specific platform and run the resulting executables through NaCl security checks, thus creating their own NaCl executables. NaCl executables are bound to webpages via HTML embed tags.

#### A.2.2 Pepper Plugin API

Due to the nature of NaCl's sandboxed environment, it is impossible to make system calls inside of a NaCl module. The Pepper Plugin API (PPAPI) exists for this purpose, allowing NaCl to

access platform-abstract system calls without sacrificing existing security and portability . The PPAPI permits NaCl applications to perform file I/O, network functionality, mouse and keyboard integration alongside 2D and 3D graphic calls with hardware acceleration, allowing for richer and visually appealing applications .

### A.2.3 Project Files and Qmake

The Qt Company uses a variety of company-made utilities and file formats to assist in compiling their codebase and their end users' applications. These tools are components of Qt libraries and can be used to compile and make otherwise non-qt related software.

Qt applications include a specialized project file format (.pro) which contains all of the information required to build the given application . The file format features a declarative language used to define most standard variables such as the included header files and source files. Control flow structures can also be added to help define more complex projects. Project files can be configured to create release or debug versions of the same project, amongst other things, however this specification can also be made during compile time with in-line arguments. Libraries and other included files are also specified here.

In order to compile applications, The Qt Company and end users use a tool called qmake. Qmake helps simplify the build process by generating Makefiles based on the contents of the application's project file . The qmake command supports a wide array of options that can be used to specify levels of debugging, the environments the app will run in, and more. Typically qmake is only used for the creation of Makefiles, but it can also be used to generate project files when given the appropriate runtime arguments.

### A.2.4 Qt Platform Abstraction

A platform abstraction layer is a codebase that facilitates the development applications that run on multiple platforms. The Qt Platform Abstraction (QPA) provides the infrastructure necessary to develop Qt applications for any platform . Users who wish to run their applications on a unique platform first have to create a plugin using the QPA which resolves Qt objects such as QString and QList to the appropriate native structures . Qt provides a configuration tool where, when defining what platform Qt is going to be built for, it will generate a custom qmake executable specific to the specified platform .

In addition to writing the plugin described above, users must write a makespecs configuration file. This file outlines which C/C++ compilers to use when compiling an application on the specified platform. Additional make specs that can be defined are other necessary header files, libraries, and environment variables. Complex configurations can be created by linking to other makespec files which will also be evaluated.

## A.3 Methodology

### A.3.1 Outdated Qt for NaCl

In order to satisfy the primary goal of porting Qt version 5.8 to NaCl, we opted to begin by replicating the steps previous developers had outlined for Qt versions 5.4 and 5.6. During this process we were in contact with Morten Sørvig, the developer for The Qt Company who was in charge of porting previous versions of Qt to NaCl. Previous attempts to establish Qt for NaCl were not a part of Qt's supported API, so all existing related repositories lived inside of Morten's personal git repository .

### A.3.2 Preparing the Compilation Environment

The preferred method to compile Qt-NaCl is to perform a shadow build, a process that involves compiling code from a build directory that is outside of the source directory. Qt is typically compiled by running a configuration script which first compiles the QtBase module, the main module behind Qt which contains core functionality and the QPA layer. This script then locates all available modules in the source directory and proceeds to compile them using the qmake script built in QtBase. Because Qt for NaCl is unsupported, most modules can not be compiled for Qt and this automated compiling process can not be used. Instead, after compiling QtBase for NaCl, it is necessary to hand-pick the other modules that can be compiled and build them separately.

We created a build directory with one folder dedicated to each NaCl supported module: QtBase, QtDeclarative, and QtQuickControls. The QtDeclarative module provides a QML interpreter which allows for easy interface programming. The QtQuickControls module provides user input fields and controls for use with QtDeclarative.

The modified QtBase source was available on Morten's git repository . In order to compile QtDeclarative for NaCl, we pulled the QtDeclarative module source code from the Qt 5.4 and 5.6 releases and applied a NaCl compatibility patch to it . We followed the same process to obtain the appropriate QtQuickControls source except no patches had to be applied. The final step of preparing our environment was downloading the NaCl SDK from Google . After adding the root directory of this SDK to our environment's PATH variable, we were able to begin the process of compiling our modules.

### A.3.3 Compiling QtBase and Qmake for NaCl

Morten's implementation of QtBase provides a tool named nacl-configure that when run, generates a makefile that compiles QtBase and creates a qmake executable. The tool supports compiling for native 32-bit and 64-bit versions, and additionally provides an option to compile for PNaCl instead of NaCl. It is important to note that while the tool supports PNaCl, inline assembly is present in QtBase's source, and inline assembly is not supported by PNaCl . While it is possible to modify the QPA makespecs to permit inline assembly in PNaCl applications, the resulting binaries can not be run in the NaCl sandbox inside of chrome as the inline assembly includes platform specific system calls. Running the nacl-configure tool configured for 64-bit Linux and MacOS gave us the tools required to compile the remaining modules for NaCl.

### A.3.4 Building Qt for NaCl Modules

Before compiling QtDeclarative, it was necessary to apply a patch that provides safe calls to otherwise unsafe functions. The patch also disables multithreading functionality as multithreading is not yet supported by Qt for a NaCl context. After applying this patch, we used the previously built qmake tool to generate a Makefile for QtDeclarative in our shadow build directory. Once that Makefile was generated, simply running make was all that was required to compile QtDeclarative for NaCl. QtQuickControls was compiled in the same way but without the need for a patch.

### A.3.5 Building Sample Application for Qt for NaCl

After successfully building a stable NaCl development environment, we began building example applications that were included in the Qt version 5.6 source code. As a user interface framework, Qt applications try to take over the main thread to make it a rendering thread which is something Native Client does not allow. Because of this, it is necessary to define the Qt application's main function as a specific Qt for NaCl function called `Q_GUL_MAIN`. Skipping this step yields a binary that does not have a main function address, and therefore will not execute. Accordingly, Qt applications that do not have a main function will also not execute in NaCl. We found that in certain Linux configurations of Qt for NaCl, NaCl libraries were not being properly linked. As a result, NaCl libraries had to be manually linked in the project file of some Qt applications.

To compile these applications, we first ran qmake on the application's project file. This generated an LLVM `.pexe` file if the PNaCl platform was specified and generated a NaCl executable if section2-bit or 64-bit platform was specified. In order to finalize the resulting binaries, we used a tool inside of QtBase named `nacldeployqt`. This tool finalized the executable, created a JavaScript loader which starts the Qt environment inside the browser, and created and a skeleton HTML page which embeds the executable to the browser. `Nacldeployqt` can be run with `-run` or `-debug` parameters.

## A.4 Results

During our time working with Qt for NaCl, we retraced Morten Sørvig's workflow and applied quick patches in the compilation process which were reported to our supervisor. By applying these patches, we successfully established a stable Qt for NaCl development environment in 64-bit versions of Linux and MacOS built on Qt version 5.6. Additionally, by using these development environments, we compiled example applications showcasing functionality of both the QQuickControls module and the QDeclarative module. One particular window rastering example also highlighted mouse event and window event functionality. These environments and example applications were positively received by The Qt Company employees.

We found that many otherwise normal Qt applications may not successfully compile in a NaCl environment due to non-standard or conflicting C++ functions linked during the compilation process. In order to determine the offending functions, it is necessary to use a debugger to isolate the function during the browser's `.pexe` to `.nexe` compiling stage. By modifying the source code to use a separate implementation of the function or modifying the bytecode of that function in raw form to jump back into the main loop of the application, these applications can successfully be compiled for NaCl. In addition, some Qt applications simply lack an appropriate main function,

and therefore can not be executed in a NaCl environment. These shortcomings were reported to The Qt Company with positive reception.

## A.5 Future Steps

Running Qt applications in browsers is practical and clearly fits into The Qt Company's aim of keeping Qt a cross-platform API. As of October 2016, Google's NaCl and PPAPI teams were destaffed . Accordingly, we recommend that The Qt Company shifts their course and pursues compiling Qt for WebAssembly instead of NaCl. WebAssembly is a robust in-browser client-side scripting platform built on top of NaCl and ASM.js. Like NaCl, WebAssembly uses LLVM bytecode as an intermediary format between source code and run-time executables and therefore would not require sweeping changes to the existing NaCl QPA . We believe that this shift will be necessary should The Qt Company decide to continue executing their native code in browsers.

Qt Lite is an existing branch of the Qt framework infrastructure that provides the features of Qt in a more lightweight manner . Qt Lite binaries occupy considerably less space than the typically large vanilla Qt binaries. These large binaries cause longer loading times and more data usage when running in a browser environment. We recommend that Qt Lite be configured for web environments so that the shortcomings of large binaries can be addressed.

Finally, Qt announced a platform for serving WebGL commands to the browser from a remote server . The work is currently in a proof of concept stage in which mouse and keyboard interactivity along with rich graphics are supported. Currently, commands are serialized in binary form and then interpreted by a custom front-end written by the end user. The Khronos group has written specifications for a similar, yet standardized, GL command and buffer schema titled Graphics Library Transmission Format (GLTF). Both of these graphics implementations would greatly benefit this project.