Increasing the Modularity of Web Pentimento's Codebase

By Halla M. Moore

S.B, E.E.C.S, M.I.T., 2015

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology
May 2016
[June 2016]

Copyright 2016 Halla M. Moore. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: Signature redacted
Department of Electrical Engineering and Computer Science
May 18, 2016

Certified by: Signature redacted
Frédo Durand, Professor, Thesis Supervisor
May 18, 2016

Accepted by: Signature redacted
Dr. Christopher Terman, Chairman, Masters of Engineering Thesis Committee

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 19 2016
LIBRARIES
ARCHIVES

Increasing the Modularity of Web Pentimento's Codebase

by

Halla M. Moore

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Pentimento is an editor for handwritten video lectures. Its purpose is to allow users to record actions in real-time while providing non-chronological editing capabilities. However, the original Pentimento program only runs on Mac computers. In an attempt to make Pentimento more generally accessible, we are creating a browser-based version of the program. We refer to this version as Web Pentimento. Handwritten video lectures are largely based in online education, so Web Pentimento will bring editing closer to the online playback environment. It also allows for future possibilities, such as real-time collaboration. We treat Web Pentimento as an opportunity to write an application that is more accessible, modular, and extensible than the original Pentimento. The contribution of this thesis involves implementing a more intuitive code architecture, incorporating event managers, and introducing new classes.

Thesis Supervisor: Frédo Durand
Title: Professor

Acknowledgments

I would like to thank my supervisor, Frédo Durand, for introducing me to and letting me be involved with this project. I have definitely enjoyed working on it throughout my time here at MIT. Thanks also to all of my family, friends, and colleagues who have played a part in this. Your support, guidance, and encouragement helped me reach this point, and for that I am extremely grateful.

Contents

1 Introduction **15**

2 Background **17**

2.1 Technical Details 17

2.2 Conceptual Models 18

2.2.1 Audio 18

2.2.2 Visuals 18

2.3 Code Overview 18

2.4 User Interactions 20

2.4.1 Editing Visuals 21

2.4.2 Editing Audio 21

2.4.3 Audio-Visual Synchronization 22

2.5 How It Works 22

2.6 Related Work 22

2.7 Overview 23

3 Architecture **25**

3.1 Hierarchical MVC 25

3.2 Subdomain MVC 26

4 Event Managers **29**

5 Accessors **33**

6 Tools Controller **37**

6.1 Individual Tools 37

6.2 Controller 38

6.3 Event Manager 38

6.4 Actions 40

7 Timeline **43**

7.1 Time Interval Gradations 43

7.2 Playhead 44

7.3 Zooming Functionality 44

8 Playback **47**

9 Selection **49**

9.1 Visuals 50

9.2 Retimer 50

9.3 Track 51

10 Transforms **53**

11 Conclusion **55**

11.1 Future Work 55

A Accessor Functions **57**

A.1 Numbers 57

A.2 Arrays 57

B Visuals Controller **59**

B.1 Tools 59

B.1.1 PenTool 59

B.1.2 SelectTool 59

B.2 Actions 60

List of Figures

2-1 The parent-child hierarchy of Web Pentimento's modules. 19

2-2 The dependencies of the Lecture module's direct children. 19

2-3 Web Pentimento's graphical user interface (GUI) 20

3-1 A singular MVC module 26

3-2 Two layers of MVC modules, following the SMVC pattern 27

4-1 Example of an hierarchical `event_types` object 30

10-1 Application flow with separate functions vs. consolidated functions . 54

C-1 Application flow for starting lecture playback 61

List of Tables

B.1 Operations supported by the Actions' event manager 60

Chapter 1

Introduction

The use of massive open online courses (MOOCs) has grown dramatically in the past few years. There are now over 550 universities offering a total of more than 4,200 courses [7]. This rise in online learning has encouraged instructors to explore different ways of posting course content. In particular, creating handwritten video lectures is a technique that has proven to be very effective. Handwritten video lectures are a style of pre-recorded lectures where the instructor draws on a "virtual whiteboard" while providing oral narration. Students who prefer handwritten video lectures describe the experience as more engaging and personal than other presentations, such as PowerPoint presentations [4].

However, these videos can be a lot of work on the instructor's end. The usual method of creating a handwritten video lecture involves drawing on a sketching program while using a screen capture program to record both the visuals on the screen and any audio over the microphone [9]. This "screencasting" method does not provide any editing capabilities. For instance, if the instructor writes an equation down incorrectly, the only way they can correct it is if they notice it during the recording process and scribble it out. The equation is still incorrect up to that point in the lecture, and having scribbles in a lecture is not aesthetically pleasing. Additionally, speaking is fast and writing is slow. It is difficult to time the narration and the visuals such that the lecture does not have any uncomfortable silences while the instructor finishes drawing the relevant visuals. Finally, if the instructor accidentally skips some of the

lecture content, there is no easy way to seamlessly insert that content into the middle of the existing video. These limitations are extremely prohibitive and require the instructor to record the entire lecture perfectly in one take. Although this process is similar to the that of a live lecture that involves a physical whiteboard, not everyone believes that these pre-recorded virtual lectures have to be constrained by the same rules. This desire for more flexibility led to the development of Pentimento.

Pentimento is an editor that allows users to create handwritten videos while retaining the ability to make retroactive edits. The insertion and deletion operations provided by Pentimento pertain to individual content components, such as a pen stroke or an audio segment, rather than entire video frames. Pentimento also supports layout edits. This enables users to, for example, move or resize a diagram they drew earlier. Other operations that Pentimento has to offer include temporal reordering, audio-visual synchronization, and content replacement[1] . With all of these operations, users have the ability to make many types of changes to their lecture without having to recreate the entire recording.

Unfortunately, Pentimento was written with Apple's Cocoa API and thus is only executable on Mac computers. In an attempt to make this editor more readily accessible, we have started developing a cross-platform, browser-based version of the program. We refer to this version as Web Pentimento. Since most handwritten video lectures exist as part of online courses, having an online version of the editor may be more compelling for first time users. Being online enables us to consider an entirely new feature set as well, including cloud storage and real-time collaboration. Developing Web Pentimento provides us with an opportunity to rewrite the original Pentimento in a way that is accessible, modular, and extensible. Accordingly, the contribution of this thesis involves implementing a more intuitive architecture, creating new classes, and rearranging modules.

[1]Content replacement allows users to replace content while maintaining its temporal relationship to other content.

Chapter 2

Background

2.1 Technical Details

Web Pentimento is written in HTML, CSS, and JavaScript[1]. We assume the reader has some familiarity with these languages, especially regarding DOM events. We also assume the reader is familiar with the Observer design pattern and the Model, View, Controller (MVC) architectural pattern. We specify several MVC triads, and generally refer to these groupings as a "module". Another ambiguous term is "class". The most recent JavaScript standard, ES6 [5], specifies that there should be formal class definitions, but these have yet to be widely implemented. To avoid compatibility issues, we continue to not use formal class definitions, but instead use closures to mimic classes. Throughout the rest of this paper, this is what we refer to when we say "class", unless we specify that it is a CSS class instead. Finally, our application has a workspace where the visuals are drawn. We will refer to this workspace as the "canvas".

[1]We use a particular JavaScript library, jQuery, but knowledge of jQuery is not assumed.

2.2 Conceptual Models

Our model for a Lecture involves two types of content: Audio and Visuals.

2.2.1 Audio

The Audio is composed of Tracks, which are further divided into Segments. When an audio recording is made, a new Segment is created with that audio content and added to the active Track.

2.2.2 Visuals

The Visuals are separated into Slides. Each Slide can contain several Visual objects. The main Visual object Web Pentimento supports at this time is a StrokeVisual. StrokeVisuals represent pen strokes made by the user, and this information is stored in the form of Vertices. Each Vertex has an (x, y, t) coordinate, representing a point on the canvas and at what time it appears.

Additionally, there is a distinction between visual time and audio time. The Retimer is in charge of the conversion between these two time scales through the use of Constraints, which connect a specific audio time to a particular visual time.

2.3 Code Overview

Web Pentimento currently follows a layered MVC architecture, which is presented in more detail in Chapter 3. Each parent module has access to the models of its child modules. Most of the modules correspond to the items discussed previously in Section 2.2, but there are a few additional ones:

- **Timer** - keeps track of what time the Lecture is currently at.
- **Timeline** - involves things that are specific to the timeline displayed in the user interface. This is discussed in Chapter 7.

- **Renderer** - draws the Visuals on a given HTML canvas.
- **Thumbnails** - a visualization of the Visuals over the timeline.

The general hierarchy between each of these modules is shown in Figure 2-1. However, some modules need access to sibling models as well. References to other modules are explicitly passed in by the overarching Lecture module. Figure 2-2 shows a dependency map between the Lecture module's children. Most of the modules need access to the Timer to know what point the lecture is currently at. The Visuals and Thumbnails need access to the Renderer, to actually draw the visuals, and the Retimer, to convert the Timer's audio time to visual time. Finally, the Retimer, Thumbnails, and Audio all need access to the Timeline so their views can query the current pixel-to-millisecond ratio, which is necessary to draw elements that indicate durations.

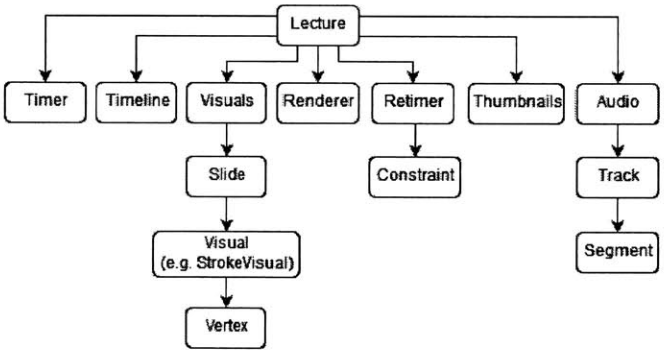


Figure 2-1: The parent-child hierarchy of Web Pentimento's modules.

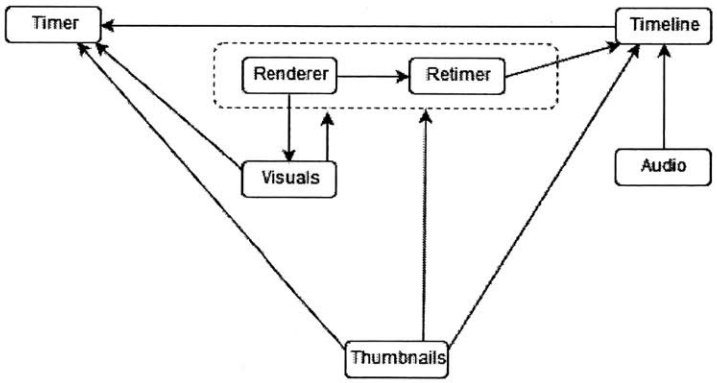


Figure 2-2: The dependencies of the Lecture module's direct children.

2.4 User Interactions

Web Pentimento is presented to users in the form of a graphical user interface (GUI). This interface can be seen in Figure 2-3. To record a video, the user simply presses the record button and starts talking and drawing on the canvas. When they are done, they press the stop button that is now where the record button was. If the user wants to record only one type of content (Visuals or Audio), they can do so by making sure only the corresponding checkbox under the record button is checked. The user can also move the playhead in the timeline to change the insertion point of their next recording.

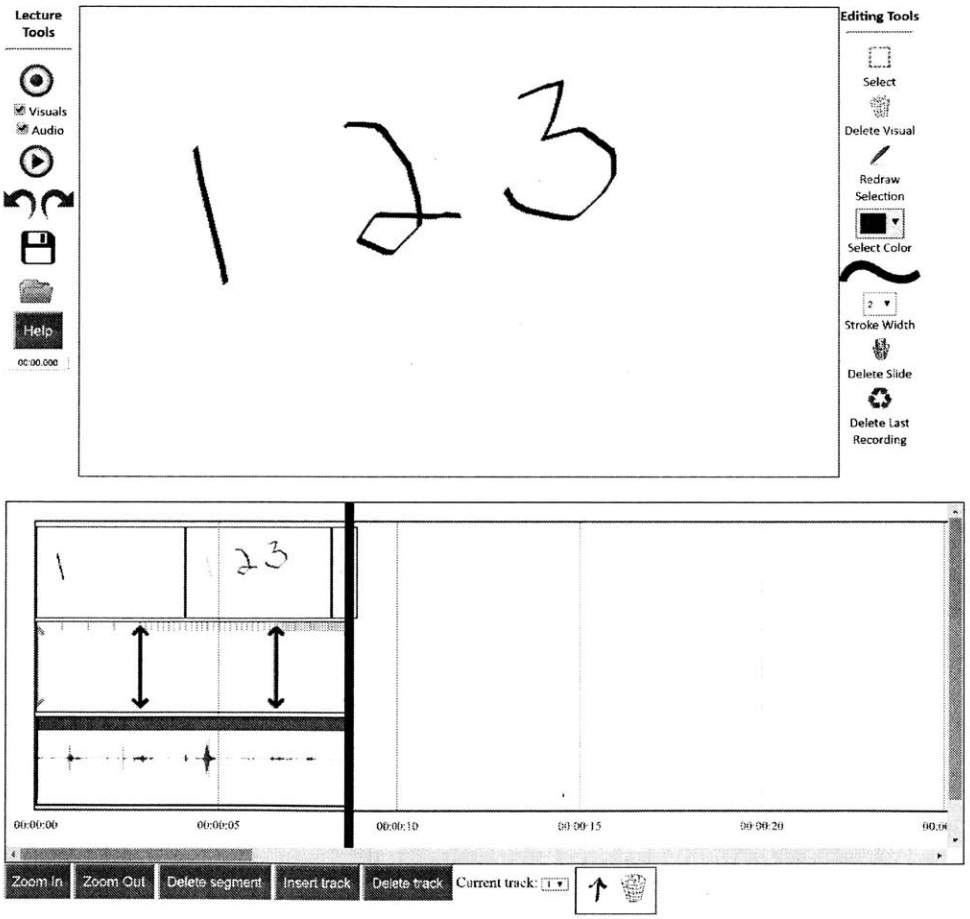


Figure 2-3: Web Pentimento's graphical user interface (GUI)

2.4.1 Editing Visuals

Web Pentimento treats visual edits differently based on whether the user is currently recording or not. If an edit is made while a recording is in process, the change only appears after that point in the lecture. If the edit is made while the user is not recording, the change is reflected throughout the entire lecture. We refer to these modes as "recording mode" and "editing mode", respectively. To edit visuals, the user first has to select the desired visuals with the selection tool. Then, any of the following operations can be performed:

- **Change color** - choose a new color from the drop-down list labeled "Select Color"
- **Change width** - choose a new value from the drop-down list labeled "Stroke Width"
- **Move** - Drag from a point within the selection rectangle to the desired location
- **Resize** - Drag from an edge or corner of the selection rectangle until the visuals have reached the desired dimensions
- **Delete** - click the "Delete Visual" icon

2.4.2 Editing Audio

Audio segments can be moved in time, cropped, and deleted. To move an audio segment, the user can simply click on it and drag it across the timeline. Audio segments cannot be dragged to a place where they overlap with another segment. Cropping a segment is performed by dragging either the left or right edge of the segment inwards. Cropping does not permanently delete the audio content, and users can bring a segment back to its original duration by dragging the edges outward instead. Finally, deleting audio segments is very similar to deleting visuals. The user selects the desired segments and presses the "Delete Segment" button. However, the user does not specifically need to use a selection tool to select segments. Just clicking

and dragging over them will work, or the user can click on an individual segment to make a singular selection.

2.4.3 Audio-Visual Synchronization

To synchronize the visuals with the audio, the user has to add a constraint. The user can add a constraint by clicking on the icon under the timeline that looks like an up arrow, and then clicking somewhere on the portion of the timeline that is in between the visual thumbnails and the audio segments. The user can modify the constraint by dragging the top arrow head to the desired part of the visual thumbnails and the bottom arrow head to the desired part of the audio. The constraint snaps back to a vertical line after the user has modified it. The thumbnails will adjust such that the constrained part of the visuals is now directly over the linked audio content. A user can also delete a specific constraint by clicking on it and then pressing the trash icon that is next to the new constraint icon.

2.5 How It Works

Vectorizing the visuals into (x, y, t) coordinates is what makes all of Pentimento's operations possible. Layout edits, such as scaling or translating, are performed by transforming the visual's x and y properties. Temporal reordering is achieved by mutating the the visual's t property. Linking a visual's t property to an audio time is the key behind both audio-visual synchronization and content replacement. Meanwhile, screencasting only produces bitmap images, which makes it impossible to distinguish and modify the stroke data.

2.6 Related Work

There are not very many programs out there dedicated specifically to handwritten video lectures, but there are programs that solve a subset of the problems that Pentimento aims to solve. Take, for instance, video editors in general. These allow users

to edit video frames, but removing or adding frames can lead to a choppy video flow. There is existing software that attempts to create seamless transitions between the remaining frames[2], but this still deals only with raster images, which means that users can not modify individual pen strokes. In order to provide this capability, some form of vectorization is needed. There is general support for vector animation [8], but this does not incorporate audio content. Animation using Adobe Flash can incorporate audio, but creating handwritten effects in Flash is very contrived. It requires the user to first type out the text, and then erase it little by little, creating frames after each erasure [1]. Lastly, there is software that specifically deals with creating an animation from handwritten input [6], but it also deals in raster images and thus lacks specific retroactive editing capabilities.

2.7 Overview

The remaining content of this paper is organized as follows:

Chapter 3: We propose an MVC variation, called Subdomain MVC (SMVC), that we believe is better suited for our application. SMVC allows for one-way model dependencies between multiple layers of MVC modules.

Chapter 4: We introduce event managers into the controller flow. These event managers help draw a line between the user interaction that indicates a specific operation and how that specific operation is communicated with the models.

Chapter 5: We describe the new Accessor class, which modularly provides getter/setter functions and ensures that the undo manager is notified of any data changes.

Chapter 6: We explain how we cut down on the responsibilities of the Visuals' tool controller by creating a Tool class and introducing another event manager instance.

Chapter 7: We present a new Timeline module, which provides a better conceptual location for logic that involves the time scale.

Chapter 8: We discuss whether the playback functions belong in the models or in the controllers, now that there is a new code architecture.

Chapter 9: We portray the current status of the Selection models.

Chapter 10: We consolidate some of the Transform functions and define a new TransformContainer class.

Chapter 11: We conclude and present future work

Chapter 3

Architecture

Our application lends itself well to a modified MVC architecture. The Lecture is a model, the GUI is a view, and the interaction between the user and the program can be specified by a controller. Our application strays slightly from this architecture in that we have multiple MVC modules. The overarching Lecture module has access to other models, such as the Audio and Visuals, which should each have their own views and controllers. Initially, dedicated views did not exist. There was a Renderer, which drew all of the Visuals on the canvas, but everything else was drawn from controllers. The very first step was to separate out all the drawing logic from the controllers. From there, we had to decide what the best practice for linking each module would be.

3.1 Hierarchical MVC

There is a variation of MVC called Hierarchical MVC (HMVC) [3]. HMVC involves several layers of MVC modules, and the communication between each layer is performed by the controller. Our application was similar to HMVC at first, in that references to controllers were being passed around. However, HMVC only provides references to other controllers because it assumes that they will receive events that are not directed at them. The controllers then have to dispatch those events up to their parent controllers until the events reach their intended controller. With JavaScript,

the controllers can be attached directly to the events they care about. No propagation is necessary. Additionally, our models are very intertwined. For instance, the Lecture model needs access to both the Audio and Visuals durations in order to know how long the lecture is. HMVC is not suited for these model dependencies and, as far as we are aware, neither is any other proposed MVC structure.

3.2 Subdomain MVC

We decided to try our own variation of MVC, which we call Subdomain MVC (SMVC). SMVC ties multiple MVC modules together such that parent modules have unrestricted access to the models in their child modules. The child controllers and views are not accessible. To understand a case where this design might be acceptable, let's consider the single module MVC structure in Figure 3-1.

In this scenario, the model is a container that stores instances of other classes. These instances are in the subdomain of the model, in that the model has complete access to each of them. These classes are, in a sense, models themselves, but they do not currently have a controller or view. Yet, there are parts of the controller and view that deal directly with each child instance. This logic can be separated out into a different view and controller. In particular, this new view and controller would belong to the subdomain model, as in Figure 3-2.

This is the basic structure of SMVC. It allows us to create layered MVC modules, while still retaining the appropriate dependencies between models. The communica-

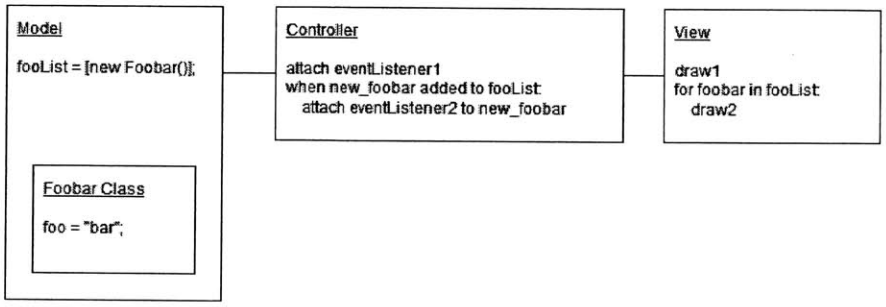


Figure 3-1: A singular MVC module

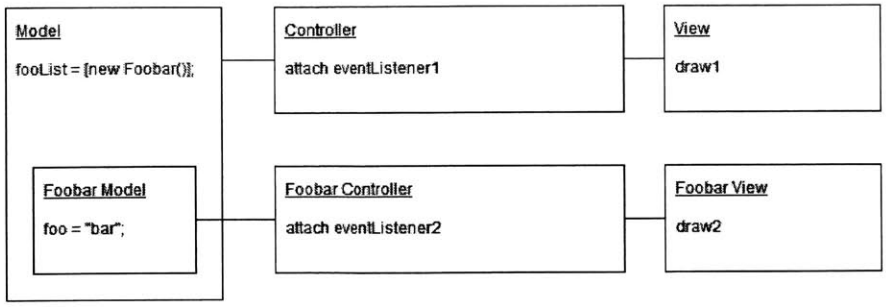


Figure 3-2: Two layers of MVC modules, following the SMVC pattern

tion link across models is only one-way, so parent models can access and manipulate their subdomain models, but subdomains do not know anything about their parents.

After refactoring our code base to have this SMVC architecture, the application flow is much more intuitive and we are no longer using controllers as indirect references to models.

Chapter 4

Event Managers

At this point, the controllers were in charge of connecting user events to how the data in the model should be manipulated. This is consistent with most MVC formats, but there are really two separate jobs being done here. The first is interpreting the user event to figure out what operation is being performed, and the second is knowing how to communicate that operation with the model. For example, let's say that the user inserts a new slide. The controller has to know a) that a click event on the "Add Slide" button means the user wants to add a slide and b) that it needs to pass that along to the model using the `addSlide` method. We decided to split these up into two separate entities, so that changing one does not interfere with the other. We left the controller in charge of communicating with the model, and created separate event managers to translate user events into specific operations.

Each of our event managers cater to unique user events and operations, but they all need the same basic Observer pattern functionality. To provide this functionality in a modular way, we created an EventManager superclass. This EventManager provides the following methods and properties:

- **initializeEvents(event_types)** Initializes the listeners data structure with each event in `event_types`. `event_types` is an object with event keys as keys and a string identifier for the corresponding events as values. `event_types` can be a hierarchical object (as in Figure 4-1), as long as the final values are unique

```
{
    drag:  {
       start:  "drag start",
       update:  "drag update",
       stop:  "drag stop"
    },
    select:  {
       start:  "select start",
       update:  "select update",
       stop:  "select stop"
    }
}
```

Figure 4-1: Example of an hierarchical `event_types` object

string identifiers. This method also sets `EVENT_TYPES` equal to `event_types`.

- **EVENT_TYPES** an object that matches the `event_types` object passed into the `initializeEvents` function. This is made accessible in order to get the correct string identifier when calling methods that require `event_type` as a parameter. `EVENT_TYPES` must not be externally modified for the EventManager to behave correctly.

- **addEventListener(event_type, listener)** Attaches the given `listener` function to the specified `event_type`. `event_type` must be a string that was provided within the `event_types` object when `initializeEvents` was invoked.

- **removeEventListener(event_type, listener)** Removes the given `listener` function from the listeners that are attached to the specified `event_type`. `event_type` must be a string that was provided within the `event_types` object when `initializeEvents` was invoked.

- **fireEvent(event_type, event_args)** Calls each of the listeners that are attached to `event_type` with the `event_args` object as a parameter. The data in `event_args` may vary. `event_type` must be a string that was provided within the `event_types` object when `initializeEvents` was invoked.

- **disableEvents()** Disables the event manager. When an event manager is

disabled, it will not notify listeners of any events.

- **enableEvents()** Re-enables the event manager. The event manager will resume notifying listeners when events happen.

Each of our individual event managers are then a subclass of EventManager, with their own unique event types. The main advantage of the event managers is that the logic for converting user events into specific operations is isolated. If we add more ways for users to perform the same operations (e.g. keyboard shortcuts) or if we just completely change how a user performs an operation in the first place, we do not have to change how the controller interacts with the model as long as the same data is provided in the event arguments. As such, when implementing any future event manager/controller logic, we should be mindful in choosing where the line between the event manager and the controller lies. It should be implemented such that we reasonably expect that the event arguments will be able to stay the same in the face of future updates.

One other point about the event managers that is worth noting is their dependence on their corresponding views. The view is in charge of drawing, e.g., a segment's drag handle, but the event manager needs to know what that element is in order to listen to its events. The current solution for this is to have the view make any relevant element IDs or classes (of the CSS nature) accessible through their namespace (e.g. `SegmentView.CLASSES.drag`). We implemented it this way simply because it was an easy solution, and it was not entirely obvious what the "right" solution would be. That being said, this mechanism could probably be improved in the future.

Chapter 5

Accessors

One functionality that is essential in any modern editor is the ability to undo. The basic idea behind the implementation of an undo manager is that whenever an operation is performed, its inverse operation is stored on a stack. When a user undoes something, the latest inverse operation is popped off the stack and executed. This requires us to register with the undo manager every time data is changed. To ensure that this registration occurs, all of our state variables are encapsulated within getter and setter functions. The setter functions are in charge of registering their inverse operations with the undo manager. At the beginning of this thesis, the getter/setter pairs looked something like this:

```
var foo = 'bar';
var getFoo = function() { return foo; };
var setFoo = function(new_val) {
    var old_val = foo;
    undoManager.add(function() { setFoo(old_val); });
    foo = new_val;
};
```

Note that explicitly storing the old value of `foo` in a separate value is intentional and necessary. Simply calling `undoManager.add(function() { setFoo(foo); });` would use the most recent value of `foo`, regardless of whether or not this line appears before `foo = new_val`. This is because `setFoo(foo)` is not evaluated until the undo

manager is actually performing an undo, so it will use whatever value is stored under `foo` at that time. To remedy this, we simply created a scoped variable that would not be changed.

However, there were still two other problems. The first was that the original variable is still accessible within its scope. This means we had to be diligent about referencing it. While it seems like it would be okay to reference the original variable just to get its value, it makes it easy to forget that the setter function has to be called for mutations. The second issue was that these lines existed for every single variable that stored some state about the lecture. This not only made our code long and hard to read, but also meant we would have to go through and change every single one if we decided to change the way we registered inverse operations with the undo manager.

To address these issues, we implemented an Accessor class. The Accessor constructor takes in an initial value and returns an object that provides the getter and setter functions for that value. The variable that stores the current value is now a private member of the Accessor class, so it cannot be directly referenced anywhere else. The implementation of the getter and setter functions is also now exclusively within the Accessor, so only that one file needs to be modified if there are any changes to the getter and setter functions. Plus, creating an encapsulated variable is now a simple one-liner: `var foo = Accessor('bar');`

So far, we have only considered the case where `foo` is a primitive data-type. Our code depends heavily on arrays as well, and the getter/setter implementation for an array is a more complex than that for a primitive data-type. Here is an example of what it might have looked like pre-Accessor:

```
var foo = [1, 2, 3];
var getFoo = function() { return foo.slice(0); };
var insertFoobar = function(foobar, index) {
   undoManager.add(function() { removeFoobar(foobar); });
   foo.splice(index, 0, foobar);
};
```

```
var removeFoobar = function(foobar) {
    var index = foo.indexOf(foobar);
    if (index < 0) {
       throw Error("foobar does not exist in foo");
    }
    undoManager.add(function() { insertFoobar(foobar, index); });
    foo.splice(index, 1);
};
```

Note that the getter function has to account for aliasing, and the setter functions have some extra logic. To get this different behavior for arrays, the Accessor constructor uses the data-type of the initial value and to determine which functions it should make available. Appendix A lists the functions provided for different data-types. One might notice that the getter function does not return a deep copy of the array, and so any nested arrays or objects would still be mutable. However, all the items in these arrays already have the Accessor functionality, so mutating them via what was returned from a higher level getter function would still require using a lower level setter function. Thus, the change would still be registered with the undo manager.

One last functionality the Accessor object has is event firing. In order for the controller to tell the view to update, it has to know that data in the model has changed. With the data now in Accessor form, the place that most clearly knows when data has changed is within the Accessor class itself. By making Accessor a subclass of EventManager, the controller can add view updates as a listener to the relevant Accessor instance. Since the Accessor class only ever fires the same generic event, it is unnecessary and cumbersome for its listeners to look up and pass in the event type's string identifier. As such, the Accessor class provides wrapper functions around the standard EventManager functions to automatically pass in the event type.

Using Accessors to provide the getter/setter and event functionality has dramatically improved the readability of our code. It also makes it much more likely that registering with the undo manager will not be overlooked when future state variables are added.

Chapter 6

Tools Controller

Another aspect of our code that could benefit from some modularity was the Visuals' tools controller. This was a single file that did everything from dealing with canvas events to figuring out which tools should stay active to implementing the exact behavior of each tool. If a developer wanted to add a new tool, they would have to implement its behavior and then change several parts of the file to distinguish between different tools. We decided the tools controller would be more modular and easier to read/modify if we split this file up into a) individual tool subclasses that are concerned only with what their own behavior is, b) a controller that creates the tool instances and knows which one is active, and c) an event manager that tells the controller which stage of operation a tool is in.

6.1 Individual Tools

To ensure that the controller does not need different implementations for different tools, each tool should be invoked in the same manner. We defined a Tool class to have three functions: `start(event)`, `update(event)`, and `stop(event)`. These functions pertain to the different stages of operation for a tool. `start(event)` is called when the user has just pressed the tool down on the canvas, `update(event)` is called as the user drags the tool across the canvas, and `stop(event)` is called when the user lifts the tool off the canvas. The `event` object should be the original DOM

event, with the following properties appended:

vis_t - the visual time of the event

coords - an array of all the coordinates from the event.

The initial implementations of these functions in the Tool class simply throw an error, stating that they must be overridden by a child class. The Tool class thus serves as a template for child classes to fill in. A description of each child Tool can be found in Appendix B.1.

6.2 Controller

With this Tool interface in place, the controller implementation becomes pretty straightforward. First, it needs a way to update the active tool. This is currently done by listening for click events from all elements with the "tool" CSS class. Upon receiving these events, the controller can figure out what tool was selected by using the element's ID to look up the corresponding tool instance. Once it has the active tool, the controller only needs to call the tool's functions based on what operation is communicated by the event manager.

6.3 Event Manager

As was briefly mentioned before, the different stages of operation for a Tool correspond to different events from the canvas. Since there are already specific events that translate directly from the canvas to each tool operation, it seems that a dedicated event manager is not strictly necessary. However, delving deeper reveals that the event mapping is not quite that simple, and a separate event manager is beneficial.

Mouse and touch events. We expect most instructors will use a stylus with touch input to create their lectures, but some instructors may use a mouse instead. As such, this event manager has to listen for both mouse and touch events. Mouse and touch events have to be handled differently in

that a mouse event has a single coordinate, while a touch event has multiple coordinates. Moreover, these coordinates are stored under different property names. The single mouse event coordinate is identified by its `pageX` and `pageY` properties, but the touch event coordinates are listed under its `originalEvent.changedTouches property`. Neither the controller nor the tools should have to differentiate between the two, so the event manager needs to convert the data from both of these events into the same format. The event manager currently puts any coordinates, whether singular or not, into an array under the `coords` property of its event object. This ensures the tools can always iterate through the coordinates, even if there is just one.

Coordinate transformations. The coordinates provided by the DOM events are relative to the upper left corner of the entire web page. However, the tools need to know what the coordinates are relative to the upper left corner of the canvas. The same coordinate transform should not be reimplemented in each individual tool, and the tools should not have to know where the upper left corner of the canvas is anyway. Instead, the event manager performs the coordinate transform before sending the coordinates to the controller.

Dragging versus moving. The DOM's mousemove event is fired when a user is dragging the mouse cursor, but it is also fired when the user is moving the mouse without dragging. The event manager is in charge of distinguishing between the two. There are two ways that the event manager could do this. One is to just have a boolean value that is set to true on mousedown and set to false on mouseup. Mousemove events that are fired when the boolean is true indicate that the user is dragging the mouse. The other way to accomplish this is to dynamically bind event listeners. Only the mousedown handler is attached at first. When it is executed, it attaches the handlers for mousemove and mouseup. When the mouseup handler is executed, it detaches the mousemove and mouseup handlers. This ensures that the mousemove handler only receives events while the user is dragging the mouse. Our event manager currently

implements the latter option, but it is unclear which option is better.

6.4 Actions

As currently defined, tools are mechanisms that are persistently active and are used by dragging the cursor across the canvas. However, the original tools controller also dealt with operations that do not match these specifications. Examples of these operations include adding a slide and changing the color of a StrokeVisual. We decided to refer to these operations as "actions" instead of "tools". Each action is a singular operation without any persisting state.

Actions can further be divided into "general actions" and "property changes". Property changes require further input from the user, while general actions do not. Thus, the add slide operation would be a general action, but the color change would be a property change because it needs to receive a color input from the user. Appendix B.2 contains a comprehensive table of the currently implemented actions.

The Actions have an event manager that is separate from the Tools event manager. This event manager returns event types that are specific for each general action, to allow the controller to determine which action is being called. To do so, the event manager listens for all clicks on elements with the "visuals-action" CSS class. It then determines which event type to fire based on the ID of the target element. Property changes are currently implemented as drop-down menus, so the event manager listens for change events on elements with the "property-change" CSS class. The event manager always fires a "property_change" event for these, but it puts the property name and the new value inside the event object. It determines the property name and new value from the ID and value of the target element, respectively.

The unique event types for each general action seemed necessary at first because, unlike how the tools part of the controller knows that the "active tool" is being targeted, the actions part of the controller does not know which action is being targeted. This also followed naturally from the fact that each action's behavior is currently implemented directly in the controller. Having the actions implemented in

the controller did not seem like a terrible idea for the time being, because there are not very many actions and each implementation is only a few lines long. However, the communication between the controller and the event manager could be simplified by incorporating an Action class that is similar to the Tool class. The controller could keep a structure connecting IDs to instances of Actions, and then the event manager could always fire the same event type, "general_action", with an ID in the event object. The controller could then just use that ID to look up the Action instance and execute its operation. This would also be beneficial if the number of Actions increases, or if any particular action has an implementation that is more that a few lines long.

Chapter 7

Timeline

As mentioned in Section 2.2, the Retimer connects audio times to visual times. This is visualized on top of a Timeline. Visualizations for the Audio and Visuals are also shown on top of the Timeline, to more clearly show the content at each point in time. The Audio is visualized with Segment waveforms and the Visuals are displayed in the form of Thumbnails.

At the start of this thesis, the Audio module was responsible for making sure all of these elements were drawn on the timeline. The Audio automatically drew the timeline and its own Segments, and also any "plugins" that were registered with it. The Thumbnails and the Retimer had to register themselves as plugins in order to be drawn in the timeline. It is unclear why this logic was placed in the audio module. Perhaps it was because the Timeline is displayed in audio time. Nonetheless, we decided it would make more sense if we introduced a separate Timeline module. Now, this module is in charge of the time interval gradations, the playhead element, and the zooming functionality.

7.1 Time Interval Gradations

The most essential part of a timeline is displaying the time intervals. The Timeline view inserts gradation lines at intervals of five seconds, in audio time. It also updates a pre-existing overlay element to match the size of the gradations con-

tainer element. This overlay is the element that the audio, thumbnails, and retimer views can populate with their own specific views. They access this overlay in the same manner that the event managers access view selectors. In this instance, it is `TimelineView.IDS.overlay`. The overlay is pre-existing and pre-populated with wrappers for the other views to ensure that they appear in the proper order of: Thumbnails, Retimer, Audio. This order is necessary because the Retimer draws Constraints with the visual time on top and the audio time on bottom. For users to be able to intuitively constrain a certain part of the visuals with a certain part of the audio, those edges should align with their associated content. Having the view container elements pre-existing in the HTML is another solution that may or may not be the "right" implementation, but it currently seems that the Timeline should not have any notion of what is being drawn over it, and this solution incorporates that.

7.2 Playhead

The Playhead is essentially a time cursor. It shows the Timer's current time on the Timeline. During recording, this scrolls along automatically. During editing, the user can move it to edit the visuals at a particular point in time. This almost seems like it would belong to the Timer, except for some of the view intricacies. From a user interface perspective, the playhead should stay visible even if the timeline is too long to be displayed all at once. As the current time progresses to the point that the playhead would disappear off the page, the Timeline needs to update its view to show the portion of the timeline that the playhead is currently traversing. Thus, the Timeline is in charge of the Playhead.

7.3 Zooming Functionality

The zooming functionality will allow users to change the scale of the time intervals in the Timeline. This is not fully implemented yet, but the Timeline is set up for

it. Namely, zooming will be possible because of the Timeline's `px_per_ms` property. This indicates how many pixels there are per millisecond of audio time. The Timeline provides `pixelsToAudioTime(pixels)` and `audioTimeToPixels(audio_time)` functions that use `px_per_ms` to help convert between the two metrics. Views that need to draw elements relative to the scale of the Timeline make use of these functions. To implement zooming, we only need to add a user interaction that changes `px_per_ms`, and attach some event listeners to `px_per_ms` so that the relevant views update when it changes.

Chapter 8

Playback

In addition to recording and editing, Pentimento also needs to be able to play the Lecture. At the start of this thesis, the Lecture controller was in charge of the initial starting or stopping of the playback, and other controllers had their own `startPlayback()` and `stopPlayback()` functions that could be called as necessary. However, now that there is no longer external access to the controllers, it is questionable whether those playback functions still belong in the controllers or not.

One could argue that playing the Lecture's content is an action performed by the user, and so it does belong in the controllers. However, this operation is not modifying the data. It is just retrieving the data in an ordered fashion, which makes it a getter function. So, although there should be a controller that initially connects the user interaction to the playback functions, maybe the the playback logic itself should belong in the models. Neither of these seemed particularly disadvantaged, so we made a decision based on the implementation costs of each option. As the code is currently structured, the models do not have access to anything inside the controllers. This means that if the playback functions were in the controllers, they would need to be executed via events. The tricky parts are cases where the execution of playback functions have to be staggered. For instance, the playback functions for each Segment should not all be executed at once. The second segment should only be played after the first segment has finished, and so on and so forth. Implementing this effect with events would be complicated. On the other hand, if the playback functions are in

the models, the models could simply call the next child's playback functions once the previous child has finished. With these considerations, we decided to implement the latter option.

There should still be a controller that recognizes when the user is trying to start or stop a playback. This logic was initially in the Lecture controller, but we put it into a separate controller so that the Lecture controller does not have too many responsibilities. This Playback controller is also in charge of starting or stopping the Timer. The Lecture controller is still in charge of connecting events from the Playback controller to the playback functions that start the entire playback chain. Appendix C shows the current logic flow for playing the lecture.

Chapter 9

Selection

In order to determine which items a user is modifying, we need to keep track of the current selection. This selection should also support multiple items, so the user can perform operations on a group of items. Like with the playback functions, it is not entirely clear whether the selection belongs in the model or the controller. Knowing which item an operation is performed on seems like the controller's domain, but the selections should also be rendered in the view and be able to be restored upon undo/redo actions. With these considerations, it seems like the selection is persistent data that should be stored in the model. Having the selection in the model may also make it easier to add future features involving selection, such as being able to record the act of selecting.

Adding functionality to store and modify a selection takes up a lot of space though, so rather than just putting it all directly in the model, we put it in its own model and instantiate it from the corresponding parent model. The general idea is that the selection model provides access to the selected items and a function to update which items are selected. Part of this update involves letting the items in the new selection know that they are selected, and telling the items in the old selection that they are no longer selected. This is necessary so that the views can render an item as selected or not. The controller usually tells the parent model which item it should be mutating, but the controller can also tell the parent model to perform the operation on every item in the selection by passing in `null` instead. The modules that currently have

selections implemented are the Visuals, Retimer, and Track modules

9.1 Visuals

This selection, the VisualsSelection, is instantiated in the Visuals model. We considered putting this in the Slide model, since the Visual objects belong in a Slide, but then there would be a separate selection for each slide. This is unnecessary because there can only be one populated selection at any given time. The function for updating this selection takes the starting and end coordinates of the selection rectangle, as well as the visual time at which the selection happened. It then iterates through all of the Visual objects that are visible at that time and selects those that are within the selection rectangle. It determines which Visual objects are within the rectangle through the use of a helper function in the Visual class, `isWithinBox(time, top, right, bottom, left)`. A StrokeVisual considers itself to be inside the box if it at least half of its vertices are within it. The selected Visual objects are notified that they are selected and their views render them with a different color than non-selected Visual objects. The Visuals selection also stores the current coordinates of the selection rectangle, so that the Visuals view can render the boundaries of the rectangle.

9.2 Retimer

The RetimerSelection stores which Constraints are selected. The update function takes a starting and ending x position, since the only degree of freedom the user has in this selection is along the x-axis. It updates the selection based on which Constraints fall into the corresponding audio time range. Similarly to the VisualsSelection, the Constraints are rendered differently if they are told they are selected, and the Retimer view draws a visualization of the starting and end x positions.

9.3 Track

The selection model for Tracks is actually called a SegmentSelection, because the items stored in the selection are Segments. This is currently inconsistent with the naming of the RetimerSelection, which stores selected Constraints. The update logic for this selection is also currently inside the Track controller, instead of within an update function inside the selection model. Additionally, the update logic is overly complicated at the moment and would probably behave better if its implementation was more like that of the RetimerSelection's update function. Overall, this selection model needs to be improved upon.

Chapter 10

Transforms

Pentimento supports various operations on Visual objects that involve a transformation. This could be a property transform (e.g. changing the color of a visual) or a spatial transform (e.g. translating a visual to the left). As briefly mentioned in Section 2.4.1, how these transformations are applied depends on whether the user is recording or editing. It also depends on which type of transformation it is. For instance, this is how each scenario for a StrokeVisual is treated:

Recording - The transform will be applied at a specific time in the lecture

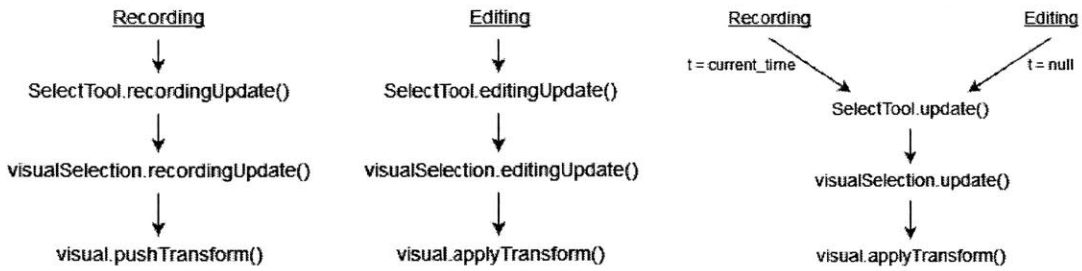
Property Transform - The transform will be pushed onto an array of all the property transforms that have been performed on that Visual

Spatial Transform - The total transformation matrix at that time will be calculated, and then that will be pushed to an array of total transformation matrices for that Visual.

Editing - The Visual's base identity will be changed.

Property Transform - The Visual's initial properties will be modified to reflect the changes.

Spatial Transform - The transformation will be applied to the initial coordinates of every vertex in the Visual.



(a) Separate functions for recording and editing

(b) Consolidated functions

Figure 10-1: Application flow with separate functions vs. consolidated functions

There used to be separate functions for each of these four scenarios. Now, they are consolidated into two functions, `applyPropertyTransform( property_name, property_value, t)` and `applySpatialTransform(matrix, t)`. `t` is an optional time parameter. If it is null, it indicates the user is not recording. Otherwise, the user is currently recording. This is beneficial because it eliminates the need to provide analogous separate functions at all levels in the chain of command. See Figure 10-1.

There are also several operations that are needed for both the array of spatial transforms and the array of property transforms, so we thought it would be prudent to create a TransformContainer class. These operations have the same implementation, regardless of the type of transform. The TransformContainer is itself an Accessor with an array value that holds all the transforms, so it supports all of the Accessor methods for arrays. Additionally, the Transform Container implements these methods:

getIndexByTime(time) Returns the array index of the transform that is active at time `t`.

insertTransformByTime(transform) Inserts `transform` into the array based on its `t` property. The array is ordered, so this matters. Transforms should not just be pushed onto the array.

getTransformAtTime(time) Returns the transform that is active at time `t`.

removeTransformAtTime(time) Removes the active transform at time `t`.

Chapter 11

Conclusion

As part of this thesis, Web Pentimento's codebase underwent some major changes. We implemented a different architecture pattern, introduced new classes, and rearranged modules. These changes provide a more modular approach, so that hopefully the code will be easier to build upon as we move forward. Although these changes were a step in the right direction, there are still some current code practices that have room for improvement and other features that need to be implemented.

11.1 Future Work

- The event managers should be implemented in such a way that we expect the data in the event object to stay the same as Web Pentimento evolves. Some of the current event managers follow this ideal, but we should double check that they all do. On a related note, the tools controller currently adds data to the event object before passing it on to the active tool. Perhaps this data should be added directly in the event manager.
- Implement a way to pass the view's IDs and classes to the event managers without publicly exposing them.
- Create a modular Action class
- Add keyboard shortcuts to change tools

- Implement the Timeline's zooming functionality
- Find a way to render the visuals in a way that more closely follows the MVC format. (E.g. Visual objects should be drawn by their views, rather than the Renderer).
- Implement the scaling of visuals. This should be done in the SelectTool class, similarly to how translating visuals has been implemented.
- Update the Track selection to be more consistent with the selection for the Visuals and Retimer.

Appendix A

Accessor Functions

The Accessor class provides a default getter/setter for any value. However, more specific setter functions may be desired, depending on the data-type of the value. Web Pentimento uses Numbers and Arrays frequently, so the Accessor class provides some extra functions for those data types. These functions are listed in the following sections.

A.1 Numbers

get() Returns the current value.

set(new_value) Updates the current value to equal `new_value`.

increment(operand) Adds `operand` to the current value.

A.2 Arrays

get() Returns a copy of the current value.

set(new_value) Updates the current value to equal `new_value`.

iterator() Returns an object that has a `hasNext()` function, which returns true if there are any more items to iterate over, and a `next()` function, which returns

the next item in the stored array.

indexOf(item) Returns the index of `item` within the stored array, or -1 if it does not exist.

valueAt(index) Returns the item that is at that `index` within the stored array.

getLength() Returns the length of the stored array.

push(item) Adds `item` to the end of the stored array.

insert(item, index) Inserts `item` at the location specified by `index` within the stored array.

removeValue(item) Removes `item` from the stored array, if it exists.

removeIndex(index) Removes the item at `index` within the stored array.

Appendix B

Visuals Controller

B.1 Tools

Currently, Web Pentimento only supports the Pen and Select tool. The event parameter is a DOM event with the following additional properties:

vis_t - the visual time of the event

coords - an array of all the coordinates from the event, as objects with x and y values.

B.1.1 PenTool

start(event) Creates a new StrokeVisual and add it to the Visuals model.

update(event) Adds new vertices to the current StrokeVisual

stop(event) Removes the tool's reference to the current StrokeVisual, so that vertices are no longer added to it.

B.1.2 SelectTool

start(event) Stores the first and last coordinate of the selection rectangle and determine which mode should be activated: none (selecting), translating, or scaling.

update(event) Depends on the mode, as follows:

None - Changes the last coordinate of the selection rectangle

Translating - Calculates the shift and move the selected visuals by that amount

Scaling - Not currently implemented

stop(event) Resets the mode to None.

B.2 Actions

Table B.1: Operations supported by the Actions' event manager

Operation	Action Type	Event Type	Event Object
Add Slide	General Action	"add_slide"	{}
Delete Slide	General Action	"delete_slide"	{}
Delete Visuals	General Action	"delete_visuals"	{}
Change Visual Color	Property Change	"property_change"	{ name: 'color', value: new_color }
Change Visual Width	Property Change	"property_change"	{ name: 'width', value: new_width }

Appendix C

Playback Application Flow

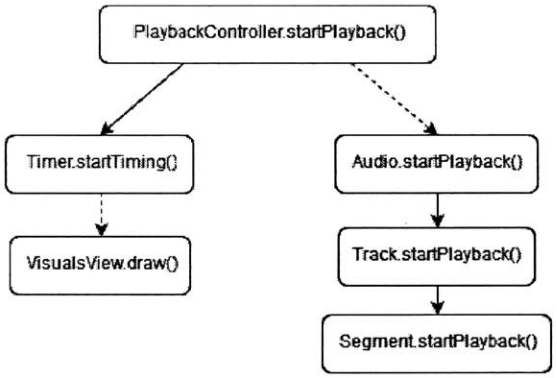


Figure C-1: Application flow for starting lecture playback

When the user starts a playback, the Playback controller first starts the Timer. The Visuals view is a listener (as indicating by the dashed line in Figure C-1) to the Timer's current time, so the Visuals automatically playback from there. The Playback then notifies its listeners that a playback is starting. Since the Visuals are already updated from the Timer, only the Audio's `startPlayback()` function needs to be attached as a listener. The Audio's `startPlayback()` function then calls each Track's `startPlayback()` function, and these in turn call each Segment's `startPlayback()` function. However, the Segments have to be staggered so that they do not play directly on top of each other. The Track makes use of timeouts to achieve this.

Bibliography

[1] Adriatic11. Handwriting flash cs3 tutorial. YouTube, June 2008.

[2] Floraine Berthouzoz, Wilmot Li, and Maneesh Agrawala. Tools for placing cuts and transitions in interview video. SIGGRAPH, August 2012.

[3] Jason Cai, Ranjit Kapila, and Gaurav Pal. Hmvc: The layered pattern for developing strong client tiers. JavaWorld, July 2000.

[4] Andrew Cross, Mydhili Bayyapunedi, Edward Cutrell, Anant Agarwal, and William Thies. Typerighting: Combining the benefits of handwriting and typeface in online educational videos. CHI, 2013.

[5] Ecma-262: Ecmascript 2015 language specification. ECMA International, 2015.

[6] J. Loviscach. A real-time production tool for animated hand sketches. CVMP, 2011.

[7] Dhawal Shah. Moocs in 2015: Breaking down the numbers. EdSurge, December 2015.

[8] Svg 1.1: Animation. W3C, August 2011.

[9] Robert Talbert. How i make screencasts: The whiteboard screencasts. The Chronicle of Higher Education, June 2011.