Video Mash-Up

for Disney Online Studios

Andrew Doppler
Ricardo Lopez
Mike Quinlan
Greg Schoeninger

Introduction

Disney Online provides the content for disney.com/create which includes web apps for artists to create and submit their digital work. Some of their current applications include making a drawing, a picture mashup, a comic strip or a flip book of Disney characters. Once the artists have finished their artwork, they may submit it for their peers to see and/or submit it into other contests.

Disney does not currently have an application for creating video mashups, and had to purchase a third-party solution for a promotion that allowed users to create their own trailer for the Avengers. The client would like to have their own application that can be easily customized for different characters or movies.

This project will also be used to determine if future applications should be written in HTML5 and JavaScript instead of Adobe Flash. HTML5 is an open standard, and is rapidly gaining support in web application and browser development. HTML5 also provides better search engine optimization than Flash, because web crawlers cannot see the content of a Flash object. Additionally, some devices do not support Flash playback at all (most notably the iPhone and iPad). Although Flash has traditionally been the primary choice of web developers, this project aims to show that HTML5 and JavaScript are excellent tools for creating complex client-side applications in modern browsers.

Functional Requirements

Front end must be able to:

- Retrieve and display thumbnails of the videos clips and audio clips.
- Drag and drop the clips to the timeline
- Rearrange and resize the clips on the timeline
- Playback the project in the canvas media player
- Add transitions between the video clips
- Play, pause, and stop the project
- Scrub to a different part of the project
- Save and load projects in their current working state

Back end must include:

- A restful service to serve up the correct asset information (ie. video and audio information) from the server in a JSON format
- A filesystem that holds all of the actual video, audio, and images for the project
- A service to save a project's current state and be able to send that data back to the front end to be loaded up and worked on again
- A way to save a final project and render it into a video

Non-Functional Requirements

The project must:

- Work in a modern web browser such as Google Chrome or Firefox
- Be written in HTML5 and Javascript front end
- Have a Ruby based back end
- Use MongoDB as the database
- Be customizable to different Disney characters and stars

System Architecture

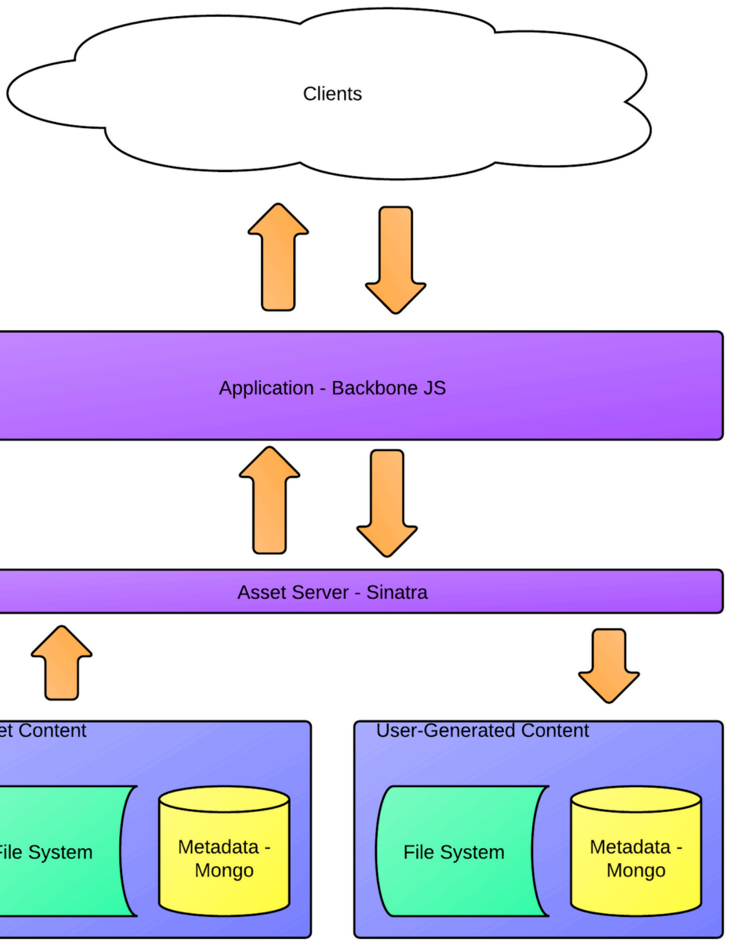

FIGURE I - APPLICATION FRAMEWORK

Sinatra and Mongo

The back end of the video mashup web application uses two different components to serve up the assets. Assets include all of the information about given video clips or audio clips as well as the video and audio clips themselves, along with their associated thumbnails. The two components of the back end are a Sinatra server and two MongoDB databases. Sinatra, a lightweight Ruby web server, creates simple routes that allow web browsers to make AJAX requests to the database.

The two Mongo databases on the back end serve two different purposes. The first one is accessed in a read-only fashion. This database holds all of the information about the audio and video clips that are loaded when a user first opens the application. This is read-only because users should not be able to modify the information that

Disney provides about the video clips. The second Mongo database is used to store user generated projects. This database allows a given user to save a project and then access it at a later date for further editing. This is write-only so that users cannot access each other's projects. Both of these databases only store information about the assets and not the assets themselves. The actual assets are stored within the file system.

The information that comes back from the database is in JSON format so that it can be easily parsed by the Javascript libraries. This information also does not contain any URLs that reference the necessary files in the file system. It is stored this way because storing information with hard-coded URLs would require a lot of repetition of data which could possibly impair the integrity of the data. Instead, the file system is ordered in a uniform way such that all of the information about the assets being retrieved can be used to derive URLs to their location. The information that is crucial to this mainly revolves around the root URL of the web server, the name of the asset, and its category.

Backbone with Require and Wire

The front end of the web application runs off of a framework called Backbone.js. This is a javascript framework that is based off of the Model View Controller (MVC) paradigm. An MVC framework provides a structured way for the application to access data as well as render it in HTML format so that a web browser can display it to an end user. The model holds all of the information about an object. The object in this particular instance is each audio and video clip that we serve up from the back end. The view takes the information from the model and renders itself in html.

Along with Backbone.js, many other third-party libraries were used in developing this project. When including many libraries, some might be dependent on others and the order in which these libraries load can be crucial to the functionality of the application. For this reason, the application also uses the javascript libraries RequireJS and WireJS to handle this dependency issue. RequireJS allows for the application to namespace the libraries that will be used in the application so that they can be easily referenced later. WireJS allows for the developer to create a file that describes how each library is dependent on other libraries. Using these two libraries ensures that all dependent javascript files will be loaded in the right order every time the application is opened.

Technical Design

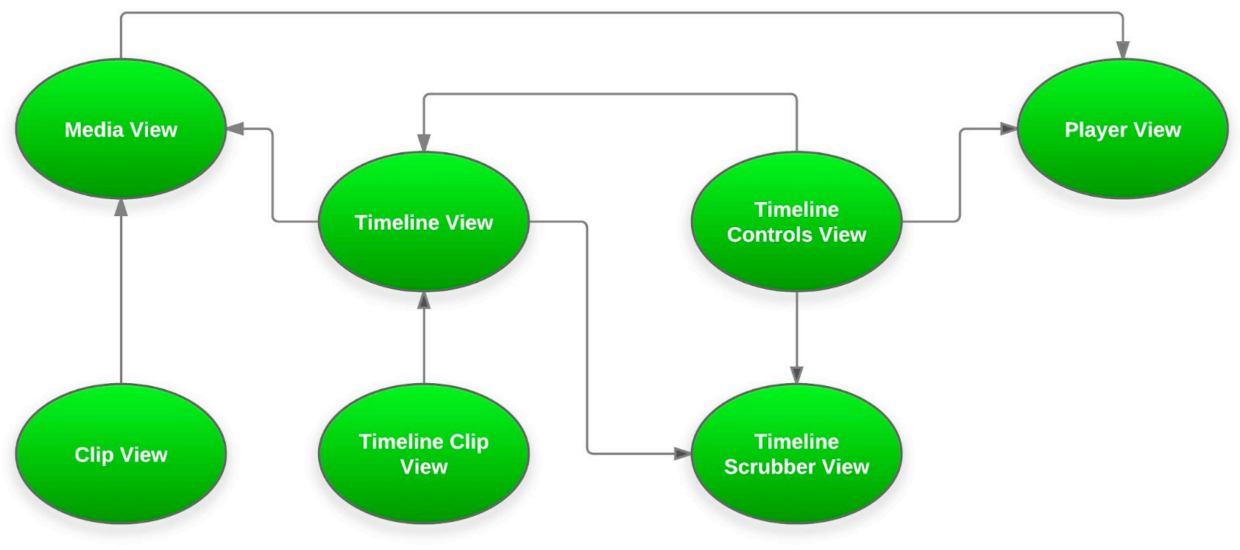

FIGURE II - VIEW DEPENDENCIES

Main App

The main application is comprised of three main components: the Media View, the Player View, and the Timeline View. Each of these components is in turn dependent on smaller subcomponents, including the Clip View, Timeline Controls View, Timeline Clip View, and the Timeline Scrubber View. These dependencies are mapped out in Figure II.

Media View

The Media View simply holds a collection of Video and Audio Clip Views. This view represents the library of different clips that are available for use in the video mashup. Clips can be used zero or more times in a project.

Clip View

The Clip View stores and displays information about each audio or video clip. This information includes the name, category, and thumbnail. Each Clip View can either be dropped onto the Player View to preview the clip, or it can be dragged into the corresponding audio or video timeline to add to the mashup.

Player View

The Player View contains an HTML5 canvas element that is used to play back the videos. Video data is loaded from a hidden video tag, then painted onto the canvas using KineticJS. This view also contains a volume slider for adjusting the audio level during playback.

Timeline Controls View

This view is responsible for playing back the project. When a user clicks the play button, all video clips in the video timeline are sequenced using VideoTrain.js, then played back along with any audio clips.The Timeline Controls View is also responsible for saving and loading the current state of the timeline. When the save button is pressed, all the Timeline Clip models are serialized into a JSON object and sent to the web server. The project can be restored by pulling the most recent version of a project from the web server, then parsing the JSON object to reconstruct all clips in the timeline.

Timeline Scrubber View

This view renders the scrubber, which provides a visual indication of the current playback position in the project. It can also be used to manually scrub through the timeline and change the playback position.

Timeline View

This view contains the video and audio timeline. Clips can be added to the timeline, reordered, and deleted. The timeline view automatically resizes itself to fit the total duration of all clips in the timeline.

Timeline Clip View

The Timeline Clip View adds more information to the Clip Views. The width of each timeline clip is directly proportional to its duration. Audio clips display a waveform generated from the audio. Video clips display a thumbnail preview of the beginning and end frames of the clip. The user can resize the clip, which changes both the displayed preview images and the saved start and end time. Video clips also include an edit button to configure the clip’s transition. If the user adds a transition to a clip, the clip overlaps on the previous clip and displays a semi-transparent overlay to indicate an active transition. Both types of clips can be removed by either dragging them out of the timeline, or by clicking their individual delete button.

Design & Implementation Decisions

This project uses MongoDB to store information about existing assets and saved projects. The flexibility of a NoSQL database was especially helpful for saving and loading projects, since the content of a video mashup is fairly open-ended. The client uses MongoDB for other projects, so this project should be easy to integrate with their existing structure.

MongoDB has the capability to store files in the database using GridFS. Videos were initially stored within the database to provide better scalability, but this caused problems when playing back the project in the browser. GridFS returns chunks of files, but the sequencing and scrubbing features relied on loading the entire video file at once. Loading video files from a static file system solved the video playback issues.

The client also requested Sinatra for a web application framework and Thin for the web server. Sinatra is a lightweight web application framework that is very easy to setup and configure. Thin is a fast web server that can handle high concurrency. Using Sinatra on Thin provides a fast and flexible web application.

Instead of creating a new system for organizing all the client-side Javascript, this project used Backbone.js (a Javascript MVC framework). Using Backbone models eliminated the need to store that data in the DOM, which allowed the code to be much more flexible. Underscore.js, Backbone's dependent library, greatly simplified binding Javascript functions to objects and events.

Playing back videos in a sequence had initially been delegated to an extension of the Popcorn.js library, Popcorn.Sequence.js. Although it appeared promising at the start, further experimentation revealed it was lacking some key features required for this project. These issues resulted in the creation of a custom library to handle video sequencing with transitions, dubbed VideoTrain.

Playing back multiple videos with transitions also presented some obstacles. The initial approach was to layer HTML5 video elements on top of each other, then play and remove each video in order. In order to provide transitions, the videos had to be decoded from the video tag, then drawn on to an HTML5 canvas. Creating custom transitions by extracting subsets of pixel data from two videos was quite complex and fragile. This was solved by using a third-party library called KineticJS which was specifically designed to provide a simple interface for creating powerful animations on the canvas. Although it was not specifically designed for video playback, the library is flexible enough to allow drawing pixel data from the videos onto the KineticJS objects. KineticJS also handles dynamic frame rates based on the performance of the user's browser, which makes for very smooth playback of videos and transitions.

Results

This project has been tested and is working well in current version of both Google Chrome and Firefox. It will probably work in Internet Explorer, but may require increased coding to reach full functionality. It will not function in older browsers that do not support HTML5. The only remaining issue is that the scrubber may not always move smoothly after pausing and scrubbing back over a transition. This is a minor aesthetics detail, and does not affect the playback of the actual video.

All functional and non-fuctional requirement have been met except for rendering the project into a single video file. This requirement was given a very low priority by the client, and could not be completed due to time constraints. However, the saved project contains all necessary data to reproduce a project, and should be easy to integrate with the client's existing rendering process.