Finding and Characterizing Video Transitions

George Giovanis
ggiovani@sfu.ca

Vincent Law
vclaw@sfu.ca

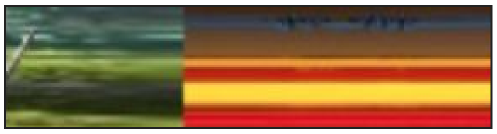
1. Introduction

Video transitions are a post production technique used in video editing to connect one shot to another. There are many types of video transitions such as wipes, cuts, fade in/out, dissolves, etc.

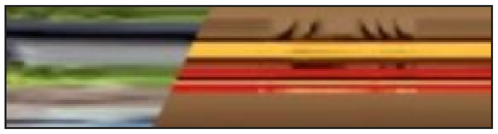
A "Spatio-temporal" image (STI) can be used to find and characterize video transitions in a video file. STIs contain video content for each frame along the ordinate axis, versus time along the abscissa.

1.1 STI by Copy Pixels

A copy of the center from each frame of the video is added to STI. The users have the options of selecting either to copy from center column or center row. In the following figures we see STIs for a video containing a horizontal wipe.



(a)



(b)

Figure 1: (a): STI consisting of the center column of each frame; **(b):** And from center row. It is straightforward to see the general area of the video transition.

1.2 STI by Histogram Differences

The frames of a video are processed sequentially. As the parameter t increments from 0 to the number of frames in the video, t_{max}, the frames F_{t-1} and F_t are resized to 32 x 32 pixels. Next, the frames are read in a column-wise or row-wise order, based on the user's selection. For each vector, a 10 x 10 histogram is constructed using the chromaticity values of the vector's pixels, as per **Equation 1**. Chromaticity values are more characteristic of an image's surfaces than (R,G,B) values, and therefore produce a more useful histogram. We found that 10 bins on each axis produced the clearest STIs. Next, the Histogram Difference is taken as the "histogram intersection" I, as in **Equation 2**, and the resulting value is used to construct a pixel that is darker when the histograms differ.

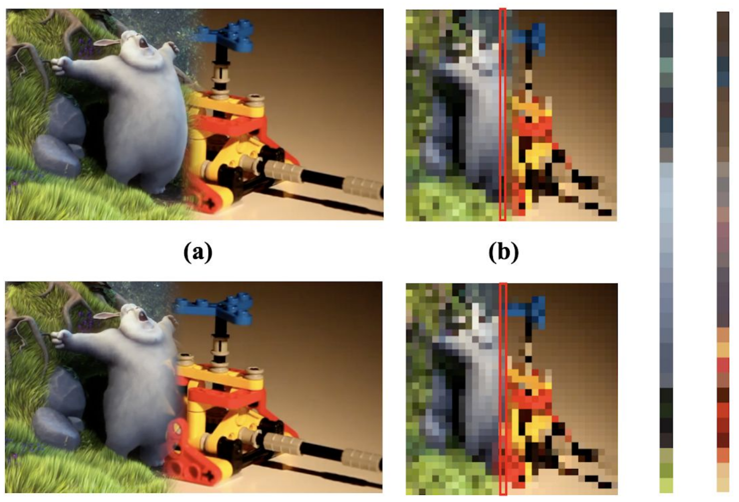
$$\{r, g\} = \{R, G\}/(R + G + B + 1/255)$$

Equation 1: Chromaticity $\{r, g\}$ as a function of (R,G,B) color values.

The resulting STI consists of grayscale pixels and has a size of 32 x Z, with Z equal to t_{max}. A graphical overview of the algorithm is presented in **Figure 2**. An STI made using Histogram Differences is shown in **Figure 3**.

$$\mathcal{I} = \sum_i \sum_j min\left[H_t(i,j)\,,\, H_{t-1}(i,j)\right]$$

Equation 2: Histogram Intersection I as a function of $\{r, g\}$ histograms H_t, H_{t-1} .



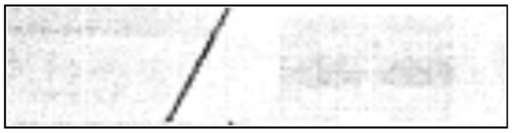
$$\begin{bmatrix} 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&3.125&0&0&0&0&0 \\ 0&0&0&6.25&6.25&0&0&0&0&0 \\ 0&0&0&81.25&0&0&0&0&0&0 \\ 0&0&0&3.125&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \end{bmatrix} H_{t-1}$$

(c)

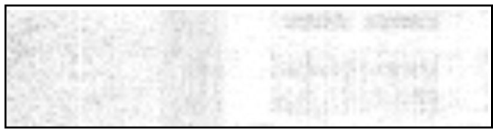
$$\begin{bmatrix} 0&0&3.125&0&0&0&0&0&0&0 \\ 0&3.125&9.375&0&0&0&0&0&0&0 \\ 0&0&6.25&40.625&0&0&0&0&0&0 \\ 0&0&6.25&28.125&0&0&0&0&0&0 \\ 0&0&0&3.125&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \end{bmatrix} H_t$$

(h)

Figure 2: Histogram Differences algorithm. Video frames at times t **(a)** and $t+1$ **(d)** are shown with their 32 x 32 pixel counterparts **(b, e)**. A red box in **(b, e)** shows a pair of columns **(f, g)** being compared. Chromaticity histograms are constructed from each column **(c, h).** I is then calculated as 0.38 with **Equation 2**.



(a)



(b)

Figure 3: STI generated on a video with a right-to-left horizontal wipe using Histogram Differences by **(a)** column and **(b)** row. The line in **(a)** indicates that the transition was detected, and its absence in **(b)** indicates that no transition was detected. This is typical for horizontal wipes, as entire columns change along the transition edge, but only a few pixels change along each row. NOTE: The black border surrounding each STI above is a visual aid, not a part of the STI.

1.3 STI by Color Nearness (IBM Method with Modifications)

The frames of a video are processed sequentially. Users have the option to generate the STI by comparing (R,G,B) or $\{r, g\}$ values.

$$A_{ij} = (1 - \frac{d_{ij}}{d_{max}}), \quad d_{max} = \max(d_{ij}), \quad d_{ij} = \|x\|_2 \,, x = \begin{cases} \Delta \begin{bmatrix} R & G & B \end{bmatrix}^T, \text{Difference of } (R,G,B) \text{ values at pixel i, j} \\ \Delta \begin{bmatrix} r & g \end{bmatrix}^T, \text{Difference of } \{r, g\} \text{ values at pixel i, j} \end{cases}$$

Equation 3: Nearness matrix A as a function of d_{ij} and d_{max} of each frame pair..

As the parameter t increments from 0 to the number of frames in the video, t_{max} , the frames F_{t-1} and F_t are resized to 64 x 64 pixels. For each pair of frames, a nearness matrix A is constructed as per **Equation 3**. This method for generating A differs from that outlined in the original IBM paper, as *A* is constructed for each pair of frames, rather than being constructed once using histogram values. We found that our method produced much clearer STIs, which is why we made the modification. Next, the frames are read in a column-wise or row-wise order, based on the user's selection. If $\{r,g\}$ was selected, then for both vectors, an 8 x 8 histogram is constructed using the chromaticity values of the vector's pixels, as per **Equation 1**. Otherwise, if (R,G,B) was selected, then for both vectors, a 4 x 4 x 4 histogram is constructed using (R,G,B) values of each vector's pixels.

$$D^2 = \mathbf{z}^{\mathbf{T}} A\mathbf{z}, \mathbf{z} = |flat(H_t) - flat(H_{t-1})|$$

Equation 4: Squared Distance D^2 as a function of **z**

Next, the difference vector **z** is constructed by flattening the resulting histograms and taking the absolute value of their component-wise difference. The Squared Distance D^2, calculated using **Equation 4**, is then used to construct a pixel value for the STI that is light when frame vectors differ.

STIs made using the Color Nearness method are shown in **Figure 4**. As is seen, there is much less variability in the pixels of the $\{r, g\}$ STI as compared to the (R,G,B) STI. For the column-wise (R,G,B) STI it is possible to make out an edge, but there is a lot of noise caused by the large L2 norm of the (R,G,B) values in *A*. It is not possible to make out a clear edge for both row-wise STIs.

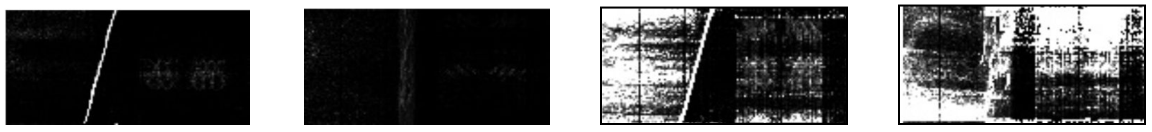

(a) **(b)** **(c)** **(d)**

Figure 4: STIs generated using the Color Nearness method with $\{r, g\}$ values by **(a)** column and **(b)** row, and (R,G,B) values by **(c)** column and **(d)** row.

2. Web Application

The architecture of our application is based on a client-server model. The client component allows end users to interact with and access the features of our application. The server component executes the core features of our application. It handles the client's request and sends back a response to the client. Restful API is used to bridge communications between the client and server components.

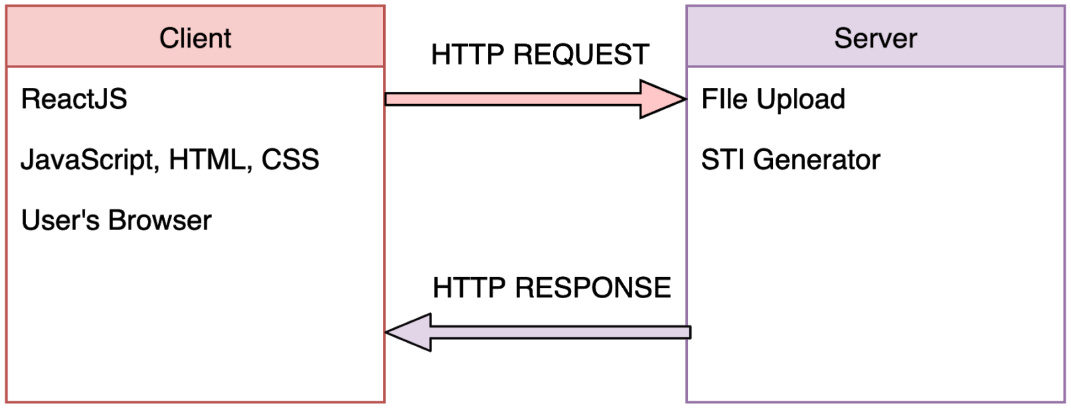


Figure 5: Web application architecture

2.1 Client Component

Our client component is written in JavaScript, HTML, and CSS, using ReactJS as the framework to build the user interfaces. The user's browser is used to display the client component.

There are several ReactJS components in our client that allow the end users to interact and access features such as file uploading, selecting videos, displaying video and STI. The ReactJS components were designed to enhance efficiency and ease of use for these functionalities:

- The Drag-and-Drop File Upload component handles file selection by either dragging the file from desktop to browser or opening a file selection dialog.
- The Select component collects user input from a list of options.
- The STI generate button allows users to execute the computation of STI and display in real-time.

The library Axios was used to make HTTP requests to the server.

2.2 Server

We used the web framework Flask to build our web server. There are two components to the server, routing and STI generator.

The routing comprise of HTTP endpoints to cover the following use cases:

- File uploading by sending HTTP POST request
- Retrieve video file by sending HTTP GET request
- Streaming STI data by sending HTTP GET request

STI Generation is done using Python, and uses the library OpenCV and NumPy to process videos and generate STI.

3. Additional Features

Below is an overview of additional features that we implemented for our project.

3.1 Frontend Features

- Elegant and intuitive user interface
- Video player with play/pause, volume, and download controls
- Video player extension to allow users to move video within the browser
- Adjustable parameters for generating an STI by row or by column
- Adjustable parameters for generating an STI with (R,G,B) or $\{r,g\}$ values
- Threshold slider for thresholding an STIs grayscale values within [0, 255]
- Thresholding on/off switch

3.2 Backend Features

- Flask server to handle user requests
- Ability to simultaneously have video playback and STI generation
- Real-time STI generation - users can watch the STI being made
- STI caching - STI of the same type is not remade
- STI generation by copying rows/columns
- STI generation by color nearness (IBM method with Modifications)

4. Application Demo

4.1 Web Demo

A live version of the application can be demod by following this link: https://infinite-harbor-16318.herokuapp.com/. Please note that the app is running on a free hosting platform, and as a result, responses can be delayed. Please allow 5-10 seconds between uploading a video and attempting to generate an STI.

4.2 Video Demo

A recorded video demo can be found by following this link:
https://drive.google.com/open?id=1SwFahp3h07T4XYxERAQt6xF3bRz3vE6x

5. Reflection

The architecture chosen to construct the application allows users to seamlessly generate STIs in an intuitive manner. Since the application runs through the browser, users are not required to install software on their computers, and are not limited by their computer hardware.

ReactJS is a widely-used framework for generating user interfaces. By choosing to use it, we were able to easily generate components for uploading a video from the user's computer to the server, playing back the video in the user's browser, displaying the STI being generated in real-time, and buttons and selectors to allow a user to choose the parameters used in generating the STI. Additionally, ReactJS supports CSS styling, which we used to produce an elegant-looking UI.

The only limitation we encountered with ReactJS is the lack of native video playback support offered by web browsers. Currently, most browsers only support playback of .mp4 files. Thus, users can only playback uploaded .mp4 videos. Importantly, this limitation does not prevent users from generating STIs with videos of other file types. A future extension to our code would be to utilize the Python backend to convert files to .mp4 so that they can be played in the frontend.

The use of Flask and Python for the backend enabled us to efficiently write code for handling server requests and for generating STIs. The OpenCV and NumPy libraries are, in our opinion, a match made in heaven for processing videos using matrix manipulations. We encountered no limitations when using these libraries.

Overall, this project was a good learning experience. It was not too hard, but it was also challenging enough to be a fun project. It gave us an opportunity to learn new frameworks and implement powerful video analysis techniques. This was a very valuable experience for both of us, as we are both interested in full-stack multimedia application development.