EC5 Video Encoder

EC 504 | Professor Kulis | Spring 2017

Hari Saran

Vinay Khemlani

Ao Li

Nate Reddi

I. Documentation

A. Problem

Without the implementation of compression algorithms, the storage of videos and sequentials would require enormous amounts of space. To solve the problems associated with the transmission and storage of files of this nature, image/video compression techniques have been developed. Some methods allow one to recreate the original file completely (lossless), while others simply eliminate "unnecessary" bits of information or decrease the quality of the file, tailoring the file so that it requires fewer bytes (lossy). For our final project, our team has chosen to create a Video Encoder (listed as Project 2 in the Project Suggestions file) to convert multiple JPEG images into a single encoded video. More specifically, the goal is to encode over 100 images into a single video within 5 minutes and be able to play that video back on demand. After coming up with multiple ideas to efficiently encode the video, our team has decided to produce a lossy algorithm that reduces the image quality of the JPEG files making the video, but focuses on minimizing the time required for the encoding. More specifically, we implement an algorithm that utilizes the power of Huffman coding by reducing the number of different values that need to be encoded, reducing the size of the variable-length codes. We then concatenate all the pixel information that we need for the images into one file that we given the “ec5” file extension. To utilize the application, we will also be providing a command-line interface to easily allow the user to encode and playback their file. Finally, Python was used to implement the applications for the project.

B. Relevant References and Background Materials

- Understanding JPEG Encoding (https://blogs.msdn.microsoft.com/devdev/2006/04/12/how-does-jpeg-actually-work/)
    - The above blog post discusses the standard encoding of the JPEG file type and the procedure that is required for converting a group of pixels into the file type. This was required to understand how the size of the file was determined from the initial pixel set and how we could modify the image itself to result in a smaller file size.
- Huffman Coding (https://www.cs.duke.edu/csed/poop/huff/info/)
    - This article helped to further reinforce the Huffman coding algorithm and how it converts an alphabet with corresponding frequencies to an encoding file.
- Numpy Array Manipulation Routines (https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html)
    - This set of documentation for the Numpy library allowed us to understand how to quickly and efficiently manipulate arrays of RGB pixel values based on the transformations we needed.

C. High-level Algorithm Description

Our encoding algorithm exploits the Huffman/Entropy coding portion of the JPEG encoding algorithm. In its standard form, JPEG encoding converts a series of RGB values that

each from 0 to 255 to YC_BC_R values, then performs various transform on these values to efficiently store them into a file, while also exploiting the human visual system to remove any differences in luminosity and color that we may not be able to perceive. Part of that transform is the previously mentioned Huffman coding algorithm that produces variable codes to represent each of the values that needs encoding. The length of the codes depends on the corresponding frequency of its value - more frequently used values correspond to smaller codes to minimize the space they require. However, the more values that need encoding, the longer the codes become as no one code can be a prefix of another.

Therefore, to reduce the length of the codes and size of the file, our algorithm reduces the range of RGB values from 0 to 255 to 0 to 127 upon encoding. This corresponds to less values needed to encode the converted YC_BC_R values and thus a smaller size. Then upon decoding, the RGB values are multiplied by two to scale them back up to a range of 0 to 254. This does result in a loss of color quality because RGB values that have an odd parity, are decoded as the even number one less than them. However, given the speed at which the images are being displayed, we believe that this change is negligible given the sensitivity of the human eye. Below is a worst case test image where all RGB values are odd before encoding (left) and are "incorrect" upon decoding (right).

To ensure that the size is reduced, we attempted to implement our own Huffman coding algorithm to re-encode the resulting values. However, we ran into troubles converting these codes into valid Huffman Tables that could be used to decode the values. Therefore, we used the optimize flag in the Pillow imaging library to select optimal encoding settings. There are limitations to this algorithm, though. For example, if all the entire image is only color, our algorithm only removes any associated metadata. If there is no metadata, the size of the encoded file is equal to the sum of the sizes of the encoded files.

The final part of our algorithm is the concatenation of our JPEG files. We do so simply by concatenating the bytes of each of the newly encoded JPEG files into one file. Then to decode, we use the standard JFIF encoding markers to detect the start and end of each file, buffer those bytes and convert them into Image classes and later Numpy arrays which we can manipulate.

D. List of Features Implemented

- Developed an Android client for viewer.

  Our Android client was developed using Android Studio and supports Android all android platforms starting with Jelly Bean. The application allows the user to access their file storage and select a valid ec6-encoded file. We then follow the same process of decoding the individual files by identifying the necessary markers within the file and converting them to Bitmap objects. To scale up the images, we utilize the ColorMatrix class to quickly apply a transformation to the pixel array of the image. Finally, to display the images, we utilize the ImageView to display one Bitmap at a time, allow the user to identify the speed at which they want to show the images, and iterate through the images accordingly.

- Developed a Graphical User Interface that provides an intuitive method for entering and/or reorganizing files, progress bar for encoding, and controls for viewing the encoding.

  Our Graphical User Interface utilizes the Tkinter library to allow customization of options for the user. Upon encoding, it allows the user to add/remove files from the encoding process and allows them to reorder the images. In addition, it gives them a second opportunity to name the file, and then shows them a progress bar during the encoding process. This GUI is optional and can be accessed by adding the "*-g*" flag to the encoding command.

  A GUI is also available and required for the viewing of the file. This interface allows the user to choose the effects they would like to apply to the images. The default is to show all images in the full color. However, the user can choose either of the image color effects (see below), the distortion effect, or a masking effect. Each interface simply maintains a list of images and iterates through it on functions like swapping/adding and while applying transformations.

- Provided real-time video effects during playback.

  Our interface allows for three times of video effects. The first are comprised of two color transformation. The first is a greyscale transformation. This is computed by calculating a weight average of the RGB values to transform them into a scalar quantity between 0 and 255. This is done by applying a dot product to each pixel with a constant, predefined weight vector. The other color transformation is a color muting transformation. This is done by again maintaining either the R, G, or B value of each pixel in an image and then turning the other two channels to 0, simply by iterating through the list of images. For the purposes of this assignment, we alternate between keeping only the red, then green, then blue channels un-muted for consecutive images.

  Our masking effect is done by generating arrays of 1's and 0's in blocks. We do so by dividing the image into square blocks of a random size and then randomly selecting blocks to be on. We perform two stages of masking so that the second mask only adds blocks to the first mask, instead of turning any off. These masks are randomly generated once per viewing and the same masks are

applied to each image. Any block of pixels that contain a 1 in the mask are left untouched in the original image. On the other hand, if the block contains a 0, we turn those pixels to white. The third stage of the mask shows the entire picture for the user to see and is displayed for twice as long as each of the masks before it. For example, the first and second masks will be applied for 25ms and the final, full image will appear for 50ms for the user to be able to identify it.

Our final effect is the distortion effect. We perform the distortion by defining a sine wave that based on a desired amplitude and wavelength. The width of the image represents the domain of the sine function and thus we iterate through each column of the image and apply the sine function to the index to determine a "shift" amount for the column. We then use this number to shift each pixel either up or down, depending on the sign of the number. If a pixel is shifted so that it either leaves the top or bottom of the image, it is rolled around to the added to the opposite side. This produces a distortion with a visible sine wave within the image.

II. Code

A folder containing all necessary code, plus test images is provided along with this write-up. In order to correctly use the command-line interface, the "encode" and "view" files must be made executable. For Linux/Mac machines, open a Terminal instance in the directory contain the folders and perform the following command for both files: `chmod -x filename`. The files then need to be moved to a folder defined with the $PATH variable of the operating system. The current directory could either be added to the PATH or, the more recommended way, is to print out the directories contained in the path (on all operating systems, in a terminal, command prompt window: `echo $PATH`) and copy-paste the files into any of those folders. For example, one directory in the path on Linux machines should be `/usr/lib/bin/`. Once the files have been moved, install the necessary libraries, as instructed below, and run the program using the commands defined in the project description.

III. Supporting Files

The following libraries are required for the use of the encoder. We recommend installing "pip" to easily install the libraries (https://pypi.python.org/pypi/pip).

- Pillow - a fork of the Python Imaging Library (if you already have PIL, you should be able to skip this step) (https://python-pillow.org)

  ```
  pip install pillow
  ```

  - Note: This library requires some dependencies. If not already installed, install the libjreg and zlib libraries first (on Linux: `sudo apt-get install libjpeg-dev` and `sudo apt-get install zlib1g-dev`)
- Tkinter - should come installed with python but if not, on Linux systems:

  ```
  sudo apt-get install python-tk
  ```
- Numpy (http://www.numpy.org)

  ```
  pip install numpy
  ```

As described in the project description, the project can be run through the command line. We implement the exact commands required. Therefore, given the directory submitted for the project, we can run the following command to encode test images attached with the project:

```
encode testPics/test1/* --output output.ec5
```

This will encode 100 images of a small size (each test folder contains 100 images of increasing size) into a file entitled output.ec5 using the GUI. When the encoding button is pressed, it also outputs statistics onto the terminal screen regarding encoding size and time.

To test our application, we focused on varying input file count, image dimensions, orientation and operating system. The program is built to run on any version of Python >= 2.7. Furthermore, there are 5 folders containing test images in the provided directory. Each folder contains images of a common video resolution. The application encodes all 100 files in each folder in well under a minute, across operating systems.

The following naming and dating of each team member acknowledges that each team member has agreed to their individual contributions:

Vinay Khemlani 5/5/17
Ao Li 5/5/17
Nate Reddi 5/5/17
Sai Sri Hari Saran 5/5/17