Software Design Document

LED Animation

Fall 2013

Team: Automaten

Everett Bloch
Grant Boomer

Last Updated: Dec. 12, 2013

Table of Contents

1.0 Overview

1.1 Purpose

The purpose of this project is to display a YouTube video on a 32x32 LED panel running off of a Raspberry Pi. It is a proof of concept for potential future projects and development.

1.2 Sponsor

The sponsor, and client, of this project is Dr. Robert Rinker, an Associate Professor in the Computer Science Department of the University of Idaho's College of Engineering. Animating an LED panel, using a Raspberry Pi, is a continuation on a project Dr. Rinker sponsored last semester, which developed drivers to displayed text on the LED screen.

1.3 Document Overview

This document describes the implementation and design decisions laid out by team Automaten for development of this project. The following sections and sub sections lay out the design description, architecture, and implementation of this project; A glossary of terms, list of figures, list of functions, and a list of references can be found in the appendix at the end of the document.

2.0 Design Description

The goal for this project is to display a YouTube video on an LED panel using the framework developed by the previous semester's team, RPLD. Team RPLD developed the display drivers for the Raspberry Pi for displaying text on the LED panel. In addition they created a 16-to-26 IDC adaptor for connecting the Raspberry Pi's 26 pin IDC to the LED panel's 16 pin IDC. This framework will make up the interface for displaying the YouTube video on the LED panel.

With this foundation all that needs to be developed is a means to:

- Download a YouTube video
- Decode the video stream, and
- Convert each video frame so it can be marshalled through team RPLD's display drivers to the LED panel

The design approach used for achieving these project goals is to develop each part separately, in three distinct modules. Figure 1 depicts the data flow for this implementation.

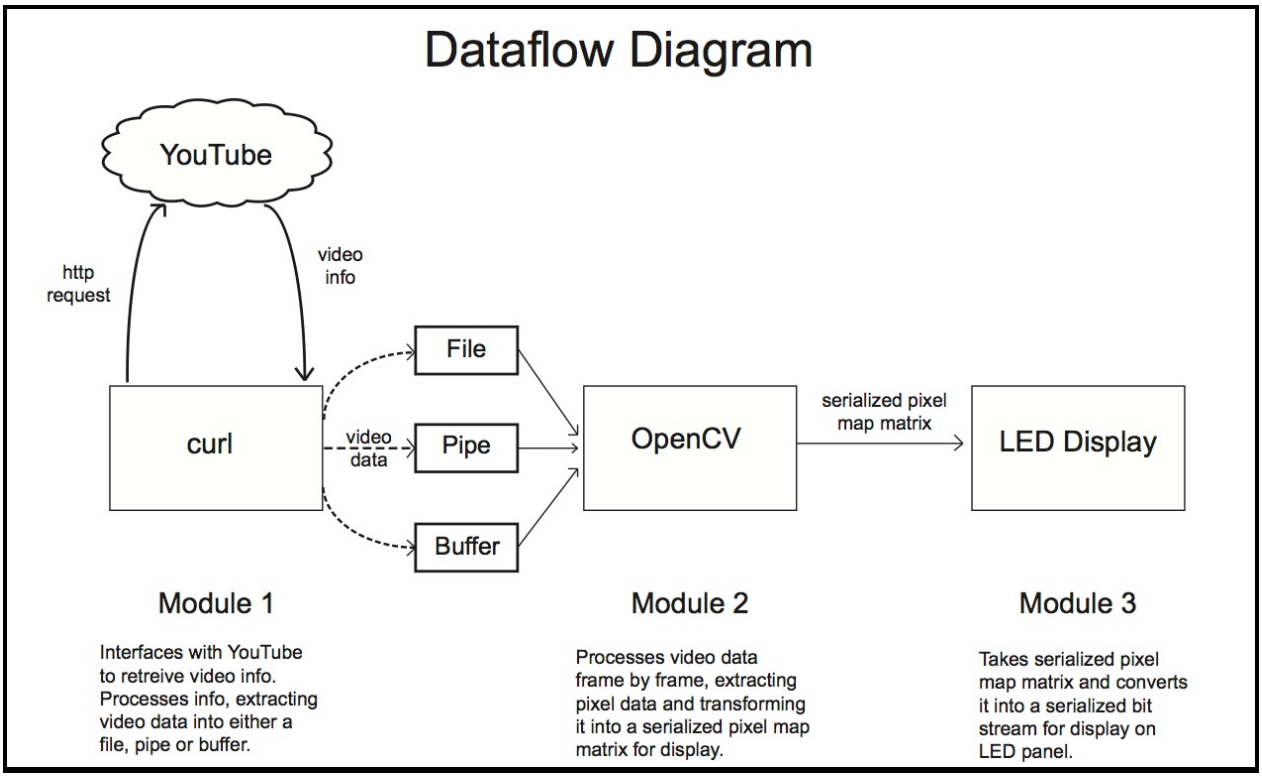

Figure 1 (Dataflow diagram)

As can be seen, Module 1 will download a YouTube video using the curl library (libcurl) to do an http request for the video information. From here the video will be buffered, either in a file, pipe, or data buffer, and sent down the line to the next module. Module 2 will take the video stream buffered by Module 1 frame by frame decode the video stream, extract the pixel data, and transform the native resolution to a serialized pixel map matrix for the next module. The ffmpeg library will piggyback on top of OpenCV, decoding the video stream and extracting each video frame, while OpenCV converts the video resolution to a desirable quality. Module 3 will take the matrix created by Module 2 and display it on the LED panel using team RPLD's display drivers.

These three modules will work in parallel to maximize the video frame rate, while preventing long loading times for the user. The exception to this is if the video is downloaded to a file, instead of streamed through a pipe or data buffer. In this case Module 1 would process until complete, then Module 2 and 3 would operate in parallel to steam the video on the LED panel.

3.0 Architecture Design

The top-down architecture for this project will be composed of four different parts, each containing individual sub-parts to achieve their respective tasks. These four parts interface with different aspects of the program and are distinguished as follows:

- User Interface
- YouTube Interface
- Video Stream Interface, and
- LED Panel Interface

The last three parts are the three modules described in the design description. These will make up the backend for this product. The User Interface will be the front end. Figure 2 depicts a detailed data flow diagram for how these different parts will interact with each other. The black boxes represent parts of each interface and are explained accordingly. The sub-sections that follow Figure 2 describe in detail the architectural design of each interface and their various sub-parts.

Note: Figure 2 has been resized to fit the page, for a pdf of the full diagram visit https://drive.google.com/file/d/0B4pp5JllGzg2cmlPTkRMYnFEd1E/edit?usp=sharing

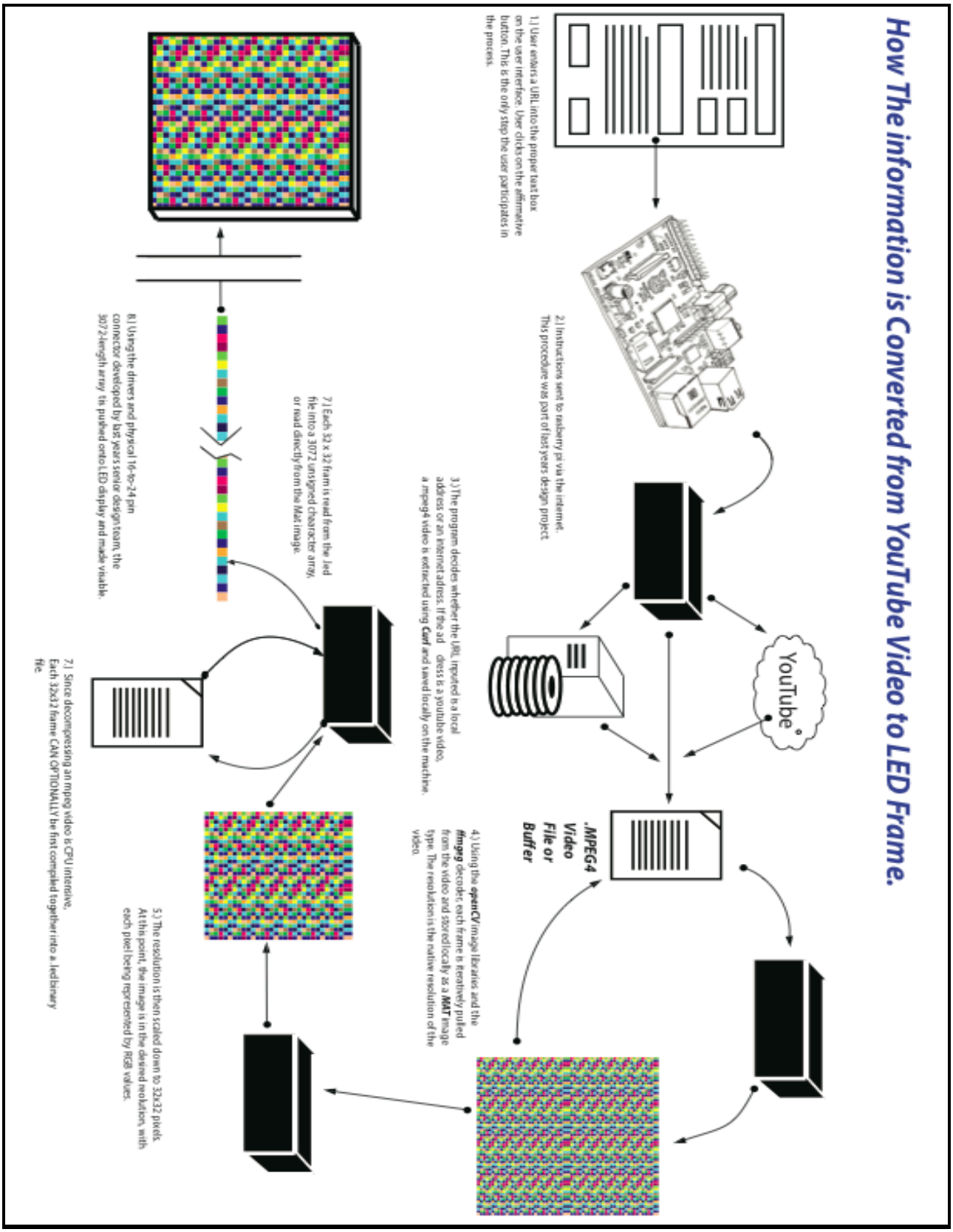

Figure 2 (Detailed Dataflow Diagram)

3.1 User Interface

The user interface for this product is a simple command line interface that provides a few different options for the user. The usage syntax for the command line interface is:

```
leda [-x] [-?] [-d file] [-b file] [-l file] <video(URL,ID,or PATH)>
```

`leda` (LED Animation) is the name of the program executable for this product. The default behavior is to stream a video specified by `<video(URL,ID,or PATH)>`, which corresponds a video location, either local or remote. Specifically, the location is either a YouTube video URL or ID, a direct URL to a video file on the network (must be ran with the `-x` flag to designate it isn't a YouTube video), or the path to a local video file on the computer. If the program is ran with the `-?` option, the list of options will be displayed, explaining their use. The remaining flags provide an interface for three alternative options:

- Download video file
- Buffer to object file
- Load from object file

3.1.1 Download video file

`[-d file]`

This flag causes the video to be download to a local file, `file`, instead of streaming it to the LED panel. It is good practice to include a proper video file extension; the behavior of streaming from a file without a file extension, or with an incorrect file extension, is undefined. All videos downloaded using a YouTube URL or ID will be MP4 files, and should have a .mp4 file extension accordingly.

3.1.2 Buffer to object file

`[-b file]`

This flag causes the video to be processed as usual, only instead of displaying the frames on the LED display, they are buffered into a local file, `file`. This option is useful for improving frame rate. The Raspberry Pi isn't powerful enough to provided a convincing frame rate while streaming. So by buffering the frame data in an object file, much of the processing overhead is complete before the video is displayed. The convention for these files is to store them wiht a .led file extention, however any file name will work as expected.

3.1.3 Load from object file

`[-l file]`

This flag causes the video to be loaded from the object file created with the `-b` option. This option provides a much greater frame rate than the streaming option, since most of the processing has been done while buffering the file, all that needs to be done is push the

data through to the LED display.

3.2 YouTube Interface

Note: Any copyrighted material hosted by YouTube is protected by YouTube's copyright agreement and cannot be retrieved and downloaded from YouTube.

The YouTube interface is the first module for this product's backend. The User Interface will send to this module a YouTube video URL, which will be used to retrieve the video stream for processing in the Video Stream Module. To accomplish this task, this module must do the following:

- Complete an HTTP request with YouTube
    - Receive YouTube provided video info
- Extract url encoded stream map from video info
    - Decode the stream map
- Parse desired video streaming URL from map, and
- Send video stream to Video Stream Interface

These four sub-parts are explained in detail below.

3.2.1 HTTP Request

To handle the HTTP requests with YouTube the open source thirdparty library libcurl will be used. libcurl versions 7.9.6 or later are supported for this product, however earlier versions will likely work, but are not being tested. libcurl needs to be preinstalled on the target machine; it is a dependency that will not be included in this products software package. To obtain a free copy of libcurl visit http://curl.haxx.se/download.html and download the appropreate package. If you have `apt-get` installed on your system it can be obtained by installing the libcurl dev package:

```
sudo apt-get install libcurl4-gnutls-dev
```

There are other versions of libcurl in the `apt-cache` however this is the binary that is used for this project. Other versions are not guaranteed to work correctly. When compiling with libcurl the library location must be provided for the linker, an example compilation is as follows:

```
gcc -lcurl libcurl_file.c
```

If libcurl has been installed correctly this should compile without any issues.

In order to get the desired video data from YouTube an HTTP request for a specific video must be issued. This can be achieved by sending an HTTP request to:

```
http://www.youtube.com/get_video_info?video_id=...
```

where the `...` is the YouTube video ID of the desired video. The User Interface can either pass in the entire URL for the YouTube video, or just the video ID. For example, if the following video URL were passed in:

```
http://www.youtube.com/watch?v=ZSt9tm3RoUU
```

it would be the same as passing in `ZSt9tm3RoUU`, and the resulting HTTP request would be sent to:

```
http://www.youtube.com/get_video_info?video_id=ZSt9tm3RoUU
```

The video info that YouTube returns is a URL encoded string containing all of the information for the requested video such as video name, video length, number of likes, number of dislikes, etc. However, if the video requested is copyrighted the returned video info will be a URL encoded string telling us the video can only be displayed on certain sites (YouTube). In the case of a non-copyrighted video, our parameter of interest in the string is the `url_encode_fmt_stream_map` parameter, it contains the data necessary to construct a URL that can stream the video. An example video info string is as follows:

```
supported_without_ads=1&video_verticals=%5B3%2C+435
...
&url_encoded_fmt_stream_map=url%3Dhttp%253A%252F%252
Fr5---sn-mv-a5ml.c.youtube.com%252
...
&video_id=ZSt9tm3RoUU&title= ...
```

This video info string returned by YouTube is around 22,200 characters long; it is guaranteed to be complete, but in no particular order. Everything that we need can be acquired by copying the value of the `url_encode_fmt_stream_map` parameter, which is everything up to the next '`&`' in the string.

3.2.2 URL Encoded Stream Map

Once the `url_encode_fmt_stream_map` parameter value has been extracted from the video info string it can be decoded. libcurl provides a URL encoded string decoder function called `unescape`, a few iterations of this are required to fully decode the stream map because it has been encoded several times.

This stream map consists of several different comma separated parameter lists. These parameter lists contain the pieces needed to construct the streaming URL for the video. Individually the parameter lists are ordered from highest quality to lowest quality. Additionally there are four different video formats that are provided:

- WebM
- MP4
- X-FLV, and
- 3GPP

The format that we are interested in is MP4. Since the goal for this project is to display a video on a 32x32 LED panel, the last MP4 format in the stream map is our desired parameter list, the lowest quality MP4 video stream.

3.2.3 Parse URL Stream

Each URL stream within the stream map contains several different parameters needed for the video stream URL. These parameters, along with their values, need to be parsed and pieced together to form the desired URL. The order of parameters for a correct URL is not specific, any order will work with the exception that the `url` parameter value comes first. Additionally no parameters can be duplicated, or the URL will be incorrect. The parameter order we have chosen is as follows:

```
url - cp - ip - upn - sparams - sver - source -
expire - itag - ipbits - id - key - ratebypass -
fexp - burst - pcm2fr – hightc - algorithm - clen -
dur - factor - gir - lmt - mt - mv - ms -
fallback_host - type - quality - signature
```

Some things to note when constructing this parameter sequence is: `signature` is the `sig` parameter in the stream map, `signature` is required to have a correct URL; `url` is strictly the value, not the parameter as well like the other name/value pairs; `url` has one of the above parameters included with its value before the `?`, so that parameter must be excluded in the query string; and `itag` has two parameters in each stream map, they are exactly the same but only one is required in the URL.

Be aware: These above values are specified by YouTube, not all are required but specifications might change. While it isn't necessary to include all of these name/value pairs in the URL it is uncertain as to what YouTube will require in the future. During the development of this project a few parameters were added to this list which required an

amendment of the code to support the new additions. As of yet all of these changes have been a value specified in the `sparams` parameter value. If problems arise when attempting to utilize the video stream URL, this parameter will likely contain a value that is not included in the code. To amend this, simply add the new parameter name to the enum defined at the top of the `parseURL()` routine and add the new parameter to be parsed under the `// parse url param` commented section. Follow the same format that is used for parsing the other parameters.

All of the above parameters, besides `url` are `&` separated name / value pairs. An example video stream URL is as follows:

```
http://r5---sn-mv-a5ml.googlevideo.com/videoplayback?ip=75.87.253.
252&upn=f3bT3Q8XCZE&sparams=id,ip,ipbits,itag,ratebypass,source,up
n,expire&sver=3&source=youtube&expire=1382945715&itag=22&ipbits=0&
id=652b7db66dd1a145&key=yt5&ratebypass=yes&fexp=935028,937600,9072
36,916611,924616,924610,907231&mt=1382925342&mv=m&ms=au&fallback_h
ost=tc.v16.cache4.c.youtube.com&type=video/mp4;+codecs="avc1.64001
F,+mp4a.40.2"&quality=hd720&signature=EF24C6F5B7FC5B2AA7BACAAA96A5
C7A6C04B0A83.ACF5BC22019A71C1A70B6E2843D72371529834FF
```

The streaming URL obtained is temporary, it will only be valid for a few hours after it is created. It is also worth noting that this URL will be redirected to a valid instance of the video stream. When accessing the URL be certain that the HTTP request for retrieving the video stream handles redirects, otherwise nothing will be returned.

3.2.4 Buffer Video Data

The final piece of the YouTube Interface module is the video buffering for the Video Stream Interface. There are three methods of buffering we are considering:

- File
- Pipe, and
- Dynamic Buffer

Each has their advantages and disadvantages.

Using curl, we could save the video stream to a file. The advantage here would be long term use and quick lookup for already downloaded videos; the disadvantage is the concurrency between the YouTube Interface and Video Stream Interface would be compromised since the entire file would have to be downloaded before processing.

Secondly, we could set up an unnamed pipe. The advantage is it allows for OS managed

concurrency between the two interfaces; the disadvantage is it uses the filesystem for I/O, diminishing performance.

Finally, we could implement a thread safe dynamic buffer. The advantage would be maximized performance; the disadvantage would be implementation time and correctness. Alternatively, the YouTube Interface could simply send over the streaming URL to the Video Stream Interface and have the decoder handle the buffering, but this depends on a third party library and may not appropriately manage the data.

We will offer two options: download then stream so the video can be saved, i.e buffer via a file, or just stream, via a pipe or dynamic buffer. These options will be specified in the User Interface, with concurrent streaming as the default.

3.3 Video Stream Interface

In order to download, buffer, process, and resize video data, a third party library suit OpenCV is used. OpenCV is a project dependency and will not be provided with the project's software package. It is important to build OpenCV with the ffmpeg library used for decoding video data. To build OpenCV, with all required depenances, on a Linux machine follow the instructions laid out at http://www.ozbotz.org/opencv-installation/. To building on OS X follow instruction liad out at http://tech.enekochan.com/2012/07/27/install-opencv-2-4-2-with-ffmpeg-support-in-mac-os-x-10-8/, be sure to add the install path to the `PATH` environment variable if it isn't already, and install pkg-config as well; exporting `PKG_CONFIG_PATH=/usr/local/lib/pkgconfig` environment variable for opencv.pc, this will allow pkg-config to output OpenCV's library paths. This path should be the directory containing opencv.pc. To compile with OpenCV it will look something like this:

```
gcc `pkg-config opencv --cflags --libs` opencv_file.c
```

For intermediate object files compilation would look like this:

```
gcc `pkg-config opencv --cflags` -c opencv_file.c
gcc `pkg-config opencv --libs` opencv_file.o …
```

The Video Stream Interface is responsible for taking the stream URL provided by the YouTube Interface, or any valid video path, downloading the compressed video data, decoding the video frame by frame, resizing each frame to 32x32 resolution, and converting each resized frame to a bitstream that the LED Panel Interface can use to display the video. To achieve this, the Video Stream Interface is divided into five sub sections:

- Download/Buffer
- Decode
- Resize
- Convert
- Intermediate file (optional)

3.3.1 Download/Buffer

OpenCV provides an API for streaming videos from a file or URL, the VideoCapture class provides a seamless interface for this process. Once a VideoCapture object has been initialized, a video from a file or URL can be downloaded and buffered. With the stream opened locally, the remaining sub parts need only pull from the local information.

3.3.2 Decode

When OpenCV has been build with ffmpeg as a dependency, the buffered video stream

can be decoded and processed frame by frame. ffmpeg allows for the decoding of all of the popular video encoding formats such as mp4, avi, flv, and more. Once the decoding scheme has been identified, the compressed video data can be extracted from the buffered information one frame at a time.

3.3.3 Resize

Once a frame has been captured, it needs to be resized to a 32x32 resolution. This is done using areal interpolation, where the average of several pixels is taken to estimate the resulting resized pixel data. This process is iterated until the desired resolution has been achieved. Areal Interpolation is very fast and effective for shrinking an image, but not very effective for enlarging an image. Video data will very rarely be smaller than 32x32 resolution, so this method of resizing will be desirable for any reasonable scenario.

3.3.4 Convert

With the video frame the correct 32x32 resolution the pixel data can be extracted from the OpenCV data structure that contains the pixel data in a matrix. This information needs to be extracted from the matrix and put into a 1x3072 `uint8_t` array for transfer to the LED Panel Interface. The OpenCV data structure a class called Mat, each frame is stored in a Mat, where the pixel data is stored as BGR values. The LED Panel Interface is expecting RGB values, so for each pixel the data needs to be extracted in reverse order.

3.3.5 Intermediate file (optional)

An optional feature of this interface is to buffer the converted `uint8_t` array in an object file. This option simply stores each converted frame into the object file instead of sending them on to the LED Panel Interface. By handling the CPU intensive video processing (detailed above) prior to displaying the video, the display throughput can be significantly improved.

Do note that this buffering process can take some time, depending on video length and quality. On average computation time will take about half as long as streaming the video to the LED display. The process can be stopped at any time, allowing for a partial video to be displayed through the LED Panel Interface. See section 3.1 for more details on usage.

3.4 LED Panel Interface

The following classes are used to display the video from a `uint8_t` array, created from the Video Stream Interface， to the LED panel. There are four parts to the LED Panel Interface. LEDDriver and VideoDisplay are designed to run on threads created, managed,

and linked by VideoController. With the exception of VideoReader, each of these is an extension of the project written by RPLD. For more information, see referenced documentation.

1. **VideoReader** - Responsible for filling a uint8_t buffer with the contents of an abstract frame.
2. **VideoDisplay** - Responsible for managing the VideoReader in a thread. When started, upon timeout this thread will invoke VideoReader for the next frame, then emit it to the LEDDriver.
3. **LEDDriver** - Responsible for writing information from the buffer to the LED display. Upon timeout, this LEDDriver will write two rows to the LED display (the *nth* and the *n + 16th*), then increment which rows it is writing to.
4. **VideoController** - Responsible for starting and stopping the Display and Driver threads. Establishes the connection, via Qt *connect*, between LEDDriver and VideoDisplay.

Note: Due to hardware limitations, the current implementation does not use this model. The above model is favorable since it provides a way to separate screen refresh rate and frame rate. However, the coarse timer and CPU requirements of opencv caused obtrusively slow frame rate and the slow interrupts caused distracting noise. The current implementation is as follows:

- The display thread is not ran, and the driver thread does not run on interrupts. Instead, each row is refreshed as quickly as possible, then a “for” wait allows each row to be seen by the user. This reduced noise.
- Frame rate is now dependent on refresh rate, which is handled by a counter in the LEDDriver. Since each native video has a different number of frames per second, this leads to a variable frame rate between videos.

4.0 Implementation

This section describes the files and functions created for this project that are available for download in in the project's code repository

https://github.com/bloche/LED-Animation

as well as in hard copy form supplied to the lead instructor and mentor of this project, Bruce Bolden at the University of Idaho.

The following descriptions will be on a file by file basis. Header files will contain data structures and procedures that are global to the file suit. Any code files included are to detail their implementation for easing future development, however not all code files will be provided (code files are .c and .cpp files).

4.1 `yt-surl.h`

This file declares the primary functions and data structures used in Module 2, the YouTube interface.

`VIDEO_FORMAT`

This is a typdef enum type that enumerates the four video formats that YouTube provides for their video stream. The enum values and their respective representative formats are as follows:

`VF_WEBM`	– WebM video format, convenient for browser streaming.
`VF_MP4`	– MPEG4 video format, supported by almost all video players.
`VF_XFLV`	– FLV video format, Adobe Flash video encoding.
`VF_3GPP`	– 3GPP video format, convenient for mobile video streaming.

As of this writing these are the only available video formats provided by YouTube.

`VIDEO_QUALITY`

This is a typedef enum type that enumerates possible video qualities to be selected from the list of YouTube video streams. For simplicity, only the highest and lowest quality videos are available through this data type.

`VQ_LOW`	– The lowest quality video stream of provided streams.
`VQ_HIGH`	– The highest quality video stream of provided streams.

char* getStreamURL(char* video_url,
VIDEO_FORMAT format,
VIDEO_QUALITY quality)

This is the major function of the YouTube interface. The parameters are defined as follows:

`char *video_url`	– The desired YouTube video URL to be streamed, or optionally, the video ID of the YouTube video.
`VIDEO_FORMAT format`	– The desired format of the video stream.
`VIDEO_QUALITY quality`	– The desired quality of the video stream.

On success the appropriate video stream URL will be returned as a `char*`, this has been dynamically allocated and should be freed accordingly.

On failure either a `NULL` value will be returned, indicating that an uncorrectable problem occurred while processing the URL, or an error string will be returned, indicating the error. To test whether the URL or an error string was returned, compare the string length to 150. If the string has less than 150 characters it is an error, otherwise a URL was returned.

char* strFormat(VIDEO_FORMAT format)

This function takes a video format enum value and returns a string corresponding to the name of the video format.

4.2 yt-surl.c

This file contains the definitions of all of the procedures used in the YouTube Interface Module. The data structures and functions listed below are local to this file. These data structures and functions are included for potential future development,

struct MemoryStruct

This struct is used to carry data to and from the curl instance via the callback function. It contains two data fields:

`char *memory`	– The string returned from the curl request.
`size_t size`	– The size of the string pointed to by `memory`.

The string returned in `memory` is dynamically allocated and should be freed when it is no longer needed.

```
static size_t curl_capture_cb(void *contents,
                              size_t size,
                              size_t nmemb,
                              void *cb_data)
```

This is the callback function for the curl instance, it is called whenever data is received from the curl request. This callback function stores the received data in `MemoryStruct`. The parameters are as follows:

`void *contents` – The data returned from the curl request.
`size_t size` – The number of blocks of data returned.
`size_t nmemb` – The number of bits per block of data.
`void *cb_data` – The structre to store data, `MemoryStruct` in this case.

The return value is the size of the data received from the curl request.

`static char* getVideoId(char *video_url)`

This function takes a YouTube video URL and strips out the video ID. Optionally the video ID can be provided, in which the ID will simply be copied. The parameter is as follows:

`char *video_url` – A YouTube video URL, or YouTube video ID.

On success the the video ID is returned. This is a dynamically allocated string and should be freed when it is no longer needed.

On failure a `NULL` will be returned indicating that the video ID could not be identified.

`static char* decodeURL(CURL *curl, char *url)`

This function decodes a URL encoded string. The parameters are as follows:

`CURL *curl` – An open curl instance.
`char *url` – A URL encoded string.

On success a non URL encoded string will be returned. This string is dynamically allocated and should be freed when it is no longer needed.

On failure a `NULL` will be returned indicating a problem occurred while decoding the the URL encoded string.

static char* parseURL(char *stream_map,
VIDEO_FORMAT format,
VIDEO_QUALITY quality)

This function parses a non URL encoded stream map, extracting the desired video format and quality from the video streams, and pieces it back together in a valid URL. The parameters are as follows:

`char *stream_map`	– The stream map string to be parsed
`VIDEO_FORMAT format`	– The desired video format
`VIDEO_QUALITY quality`	– The desired video quality

On success a string is returned containing the desired streaming URL. This string is dynamically allocated and should be freed when it is no longer needed.

On failure `NULL` will be returned, indicating a problem occured while parsing the the stream map. A message will be printed to `stderr` indicating the problem.

static char* laststr(char *str, char *substr)

This function finds the last instance of a substring in a string. The parameters are as follows:

`char *str`	– The string to be searched through.
`char *substr`	– The substring we want to find.

On success a pointer to the last instance of the substring will be returned.

On failure `NULL` will be returned, indicating the substring was not found in the provided string.

4.3 Video Display Classes

The following classes are used to display a video data in the form of a `uint8_t` array to the LED panel. Each of the classes fall under one of three roles:

- Model
- View
- Controller

Figure 3 shows a UML diagram depicting the general structure of, and interaction between, each class:

Note: Figure 3 has been resized to fit the page, for a png of the full diagram visit https://drive.google.com/file/d/0B4pp5JllGzg2OEJVOE1ZaU1fTDg/edit?usp=sharing

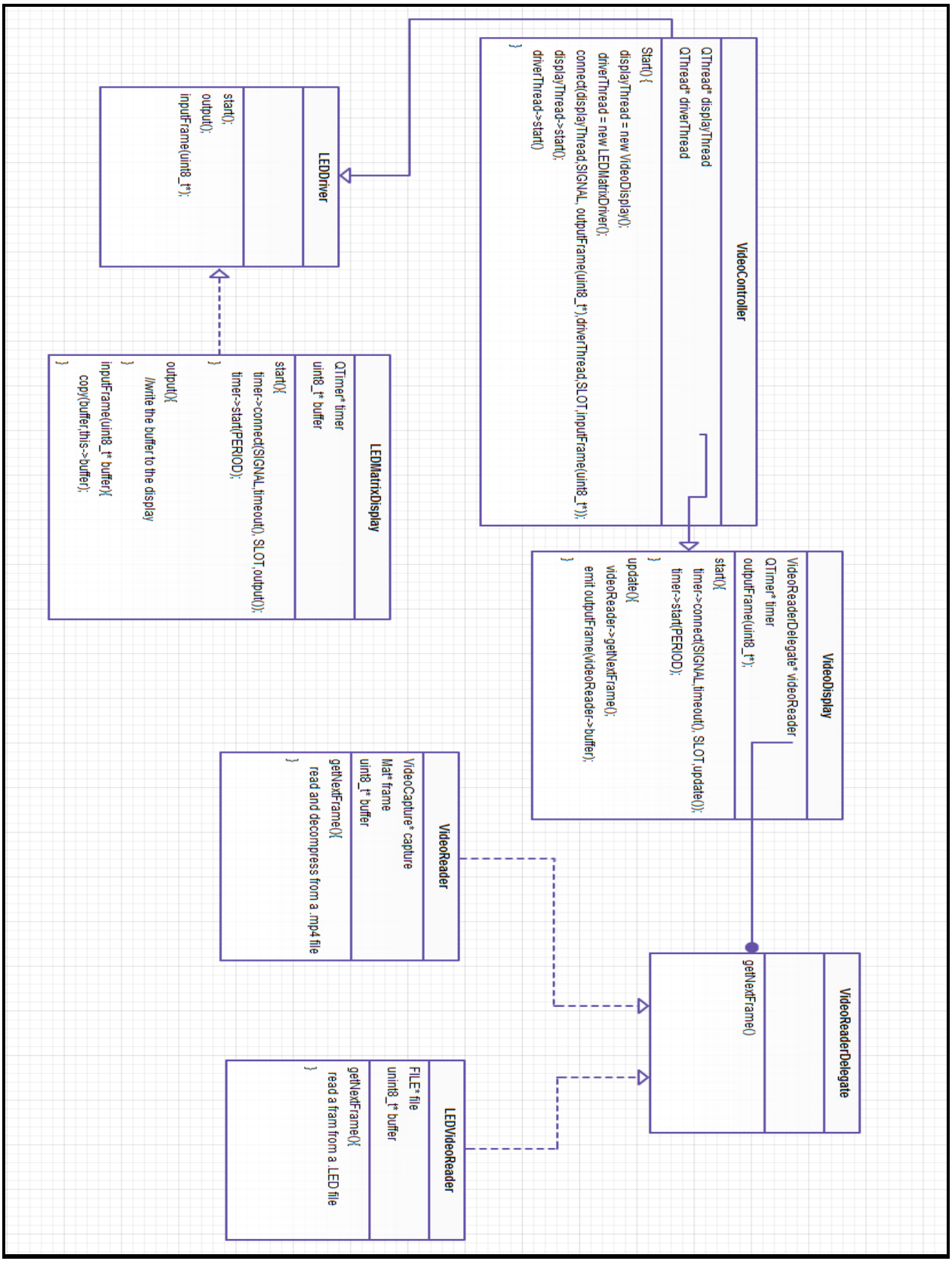

Figure 3 (Video Display Classes UML Diagram)

4.3.1 Model Classes

`VideoReaderDelegate(char* filename, uint8_t* buffer)`

This abstract interface classes sets the contract for reading frames from a file format and loading them into the passed buffer. The contract includes the following methods:

`void getNextFrame()`

reads the next frame from the file and places the contents into the passed buffer. The format should be in the 3 color unint8_t array format that is required by the View.

`void setBuffer(uint8_t* buffer)`

Sets the buffer that the reader writes to to the passed parameter.

`VideoDisplay(char* filename)`

This concrete class provides the passes for a thread that communicates with the model, via Qt QThreads. For muti-thread work, this class implements the following methods:

`void start()`

start the thread, creating a new timer. The default interrupt is once every 33 milliseconds.

`void stop()`

stops the threads. If the display thread is killed, main is signaled to stop.

`void outputFrame(uint8t* buffer, int size)`

This method is used to connect the `DisplayThread` (represented by this class) and the View thread. A reference to the buffer is passed as well as its size, and the View's method `inputFrame` is called.

4.3.2 View Classes

See also SDD for previous years project produced by team RPLD. RPLD's documents can be found can be found here: http://seniordesign.engr.uidaho.edu/2012-2013/raspberry_pi/documentarchive.html.

`LEDMatrixDriver()`

Class that writes the `uint8_t*` buffer to the LED display. With the exception of the following method, this class remains as per teams RPLD's implementation.

`void output()`

In addition to the original functionality of writing two rows of the buffer to the LED display,

this function also manages a 16-bit mask `modulationMask`. On instantiation, the `modulationMask` has a value of 1. Once each row has been written to once, the value of `modulationMask` is shifted left once. Once the value reaches 32,768, or the one is in the leftmost place, the value is then set back to 0, and the whole process begins again.

4.3.3 Controller Classes

`VideoController(char* filename)`

This class is responsible for creating and starting both the model and view threads. As such, this class keeps a private reference to `QThread* displayThread` (model) and `QThread* driverThread` (view). to connect the two, the following method is used (inherited from `QObject`):

```
connect(this->display, SIGNAL(outputFrame(uint8_t*,ulong)),
this->driver, SLOT(inputFrame(uint8_t*, ulong)));
```

The last responsability of this thread is to exit the program once the display thread kills itself, either through an error or by reaching the end of the file.

Appendix A: Glossary

curl: a free, open source computer software project providing a library and command-line tool for transferring data using various protocols, including HTTP, FTP, TELNET, SMTP, and much more (see libcurl).

ffmpeg: a free, open source, complete, cross-platform command-line tool and library suit for recording, converting and streaming audio and video.

IDC: (Insulation-displacement connector) a serial port connector that allows for serialized I/O between the Raspberry Pi and the LED panel.

LED: (Light Emitting Diode) an electronic device that lights up when electricity passes through it.

libcurl: the curl library; a free client-side URL transfer library that is use in this project for interfacing with YouTube.

MP4: (MPEG-4 Part 14) a digital multimedia format most commonly used to store video and audio.

Interpolation: a method of constructing new data points within the range of a discrete set of known data points.

OpenCV: (Open Source Computer Vision Library) a free, opensource, cross platform library that focuses on real-time image processing.

Raspberry Pi: a credit-card-sized single-board computer developed in the UK by the Raspberry Pi Foundation with the intention of promoting the teaching of basic computer science in schools.

RPLD: (Raspberry Pi LED Display) the team from the previous semester that worked on this project. Specifically they created display drivers for displaying text on the LED panel.

SRS: (Software Requirements Specification) the document that describes the software requirements for this project.

SDD: (Software Design Document) the document that describes the design and implementation decisions for this project.

WebM: an audio-video container format designed to provide royalty-free, open video compression for use with HTML5 video.

X-FLV: a container file format used to deliver video over the Internet using Adobe Flash Player versions 6–11.

3GPP: a multimedia container format defined by the Third Generation Partnership Project (3GPP) for 3G multimedia services.

Pulse-Width Modulation: The process of controlling the intensity of color though variation of the length an led is on or off in a given period.

Appendix B: List of Figures

Appendix C: List of Functions

```
VideoReaderDelegate(char* filename, uint8_t* buffer)
     void getNextFrame()
     void setBuffer(uint8_t* buffer)
```
Page 22

```
VideoDisplay(char* filename)
     void start()
     void stop()
     void outputFrame(uint8t* buffer, int size)
```
Page 22

```
LEDMatrixDriver()
     void output()
```
Page 22

```
VideoController(char* filename)
     connect(this->display,
            SIGNAL(outputFrame(uint8_t*,ulong)),
            this->driver, SLOT(inputFrame(uint8_t*, ulong)));
```
Page 23

Appendix D: References

http://curl.haxx.se/
http://en.wikipedia.org/wiki/Flash_Video
http://en.wikipedia.org/wiki/Insulation-displacement_connector
http://en.wikipedia.org/wiki/MPEG-4_Part_14
http://en.wikipedia.org/wiki/Raspberry_Pi
http://en.wikipedia.org/wiki/WebM
http://en.wikipedia.org/wiki/3GP_and_3G2
http://opencv.org/
http://seniordesign.engr.uidaho.edu/2012-2013/raspberry_pi/index.html
http://www.ffmpeg.org/