Cloud-based RAW image editing

Student Name: Ryan Collins (gcdk35)

Supervisor Name: Dr Tom Friedetzky

Submitted as part of the degree of M.Eng Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

Abstract —

Context/Background

Aims The main aim of this project is to test the feasibility of a Cloud-based RAW image editor.

Method A render server backend will first be implemented as an API, taking in an input as a JSON object, and then processing the image, and then a JavaScript client shall be created to interface with this API.

Proposed Solution A web application that uses dcraw coupled with custom Java code to read RAW images, and allow adjustment of various parameters, with the output being sent back to the user.

Keywords — RAW image editing, dcraw, cloud image editing

I INTRODUCTION

Many photographers use a file format (or rather, a family of very similar file formats) called RAW, which rather than compressing the image and conducting some image manipulation on the camera, store the RAW camera sensor data outputted by the camera sensor, for later processing and editing by a computer. These files can be much larger than the compressed image, but provide a far greater degree of control over the captured image, when compared with a compressed JPEG, along with an increase in quality. A RAW file essentially acts as a digital negative, as the image can be edited constantly without losing any quality between edits. (Verhoeven 2010)

A Project Aim

The aim of this project is to produce a Cloud-based RAW image photo editor, both as an API (allowing potential image rendering headlessly), along with a full RAW image editor interface via a web browser. The image photo editor should be able to read at least DNG and NEF files, and allow contrast, colour and exposure adjustment, along with noise reduction, haze removal, and auto correction options. Furthermore, this system should be multi-user, allowing more than one user to edit RAW photos at a time through the web interface, each with their own individual collection of images.

B Deliverables

II DESIGN

This section presents the proposed solutions of the problems in detail. The design details should all be placed in this section. You may create a number of subsections, each focusing on one issue.

This section should be up to 8 pages in length. The rest of this section shows the formats of subsections as well as some general formatting information. You should also consult the Word template.

A Requirements

B Proposed Extensions

C Architecture

As the system is fairly large, it's important to break it down into individual pieces. These are:
TODO: Add diagram of the system architecture

C.1 Client Interface to Render Server Communication

There are several options for communicating between the user interface and the render server.

Representational State Transfer (REST) A Representational State Transfer system would work by sending a request to the server, requesting an image be rendered. This initial request will be replied to with a job number, which will be quoted in further communications. From here, future requests will be made over a regular interval, requesting the information for the specified job. If this job is finished, the result will be returned, but otherwise further requests will need to be made. This is the process of polling.

The XMLHttpRequest object in JavaScript, used to make AJAX requests, was designed by Microsoft in 1999, and later adopted in the 2000s. This method is definitely the most compatible with browsers, being compatible with Edge, Chrome, Firefox, Internet Explorer 7+, Opera, and Safari. (Mozilla 2017)

However, this method does require making many requests and connections to the server, coupled with code to regularly poll at an interval until done. While this will work, it's not the most optimal solution, and the code produced from this would be more complex (code could be needed to issue jobs, recall jobs, and to ensure that the person who requests the result of a job is actually allowed to see the result of the job).

Web Socket Web Socket allows for two way communication between a browser and a server. It acts in a similar way to traditional TCP socket communication, only it incorporates the origin based security model used within web browsers. By using web socket over REST, opening multiple HTTP connections is not needed, as a single connection is maintained at all times. (Melnikov & Fette 2011)

Socket.io Socket.io is an implementation that relies on both REST and Web Socket. If the browser supports web socket, then it is utilised, but in the event that the user's browser does not support new web socket technologies, then it defaults to using REST, and automatically polls regularly for information. This way, we get the best of both options shown above, in such a way that the code itself remains fairly tidy (as the polling nature of REST is abstracted away from our system).

C.2 Render Server

TODO: ADD DIAGRAM OF THE RENDER SERVER SUB LAYERS (ADAMS, DCRAW) The render server takes instructions given to it (with accordance to our API), and generates the output image based on the RAW image supplied, and the appropriate settings.

One of the first considerations is how to parse RAW files.

dcraw Dcraw is an executable that allows the processing of RAW image files.

Dcraw itself can then be used to convert to other formats, one being the uncompressed TIFF format, giving us a very high quality image that can then be adjusted. Our system isn't merely a wrapper for dcraw in this instance, but extends the features supplied by dcraw. **TODO: Explain what DCRAW is, how it can be used. TODO: DCRAW features, built in. Export as**

libraw LibRaw is a C++ library based on dcraw, that is designed as a library rather than just an executable.

LibRaw would be used when loading RAW images, processing them, and then from this, the image can be processed using custom routines (i.e. converting the libraw format to a matrix representation, and then using that matrix representation to carry out some manipulation).

While LibRaw has many useful features, the documentation is somewhat limited.

ImageMagick ImageMagick is a library that can be used to process any images, not just RAW. While it can read RAW image files, it also has many more features, including many feature that we don't need/want to implement ourselves for the purpose of this project. As such, I believe while ImageMagick is a good option, it's a bit too heavy for our use. **TODO: SOURCE FOR IMAGEMAGICK READING RAW FILES TODO: SOURCE FOR FEATURES OF IMAGE MAGICK**

C.3 Client-side Interface

The page design of the interface shall follow the design in D. The goal of the client side interface is to allow adjustment of the image parameters, and show a preview whenever a parameter is changed.

To display the image, an HTML5 Canvas will provide a large amount of control to how we can display the image, allowing for features such as zooming, and drawing. This can't be achieved using a standard HTML image.

D User Interface

A sidebar should be used as the main interface for adjusting image parameters. This sidebar should be able to be hidden, showing the image fully underneath. When the sidebar is in the expanded state, the preview image should be displayed fully on screen.

The user interface design is shown in Figure **??**

Within the sidebar, clicking on a navigation item will display a new submenu. If the item is a parameter adjustment, updating the value in the menu will also update the value that is sent to the render server to generate a preview.

TODO ADD INTERFACE DRAWINGS.

E Implementation Information

Each individual module of the system requires different technologies in order to produce an overall system.

Client-side Interface For this, HTML5, CSS3 and JavaScript (with jQuery to provide extra functionality) are ideal, as HTML5 can be used to create the user interface, with CSS3 providing styling and some basic animation to improve the UX. Using JavaScript with jQuery, it allows us to keep track of the state of the editor, and transmit and receive information between client and render server with help of some library. jQuery is useful for interfacing with the DOM (the webpage itself, and it's components), allowing us to specify events, and functions to run on particular events, along with template loading and various other functionalities.

As this is mostly static content, and doesn't directly require any web service to be performed for the main editor, the static content can be hosted using a web server like Apache or NGINX, only serving static content, without any dynamic scripting needed.

NGINX is what I'd recommend in this situation because unlike Apache, NGINX isn't configured out of the box for dynamic content (e.g. PHP/Python/other external processor). As we don't need any dynamic content, as we are just serving static files, NGINX is ideal for our workload (as less overhead is needed for features that won't be used such as dynamic content rendered on the server side). (Ellingwood 2015)

Render Server While lower level programming languages might be useful for image processing, the requirement of linking this with web technologies means that a language such as C or C++ is not as ideal.

Java is a better choice, as there are libraries available to provide web services (REST), along with Web Socket implementations, along with Socket IO. Furthermore, Java contains some libraries within the language that allow for image manipulation, most notably BufferedImage related features such as ConvolveOp, image resizing, and various other standard features build in. This is useful, as rather than starting completely from scratch, the custom image processing routines can build on the built in Java implementations, and create more advanced algorithms using them.

While Java doesn't support loading TIFF into BufferedReader using ImageIO directly, an extension exists online, in the form of TwelveMonkeys. No RAW image loading library exists within Java, but an executable such as dcraw or rawspeed can be used to generate a file that can be read within Java, while retaining the quality (e.g. uncompressed TIFF). This can be done using ProcessBuilder, and ideally writing a library to do this.

As this system will likely use a large amount of external dependencies, a dependency management tool like Apache Maven should be used, to both manage dependencies from an external source (e.g. GitHub, Maven Repositories), and also to build the system. **TODO: reference libraries here.**

Account Management Server As this system needs to be used by multiple people, we need to have some record of users, and their associated images.

Rather than rendering each page dynamically, we can instead use JavaScript to make AJAX requests, fetching the data needed, and this way, we don't need to render whole HTML web pages. In terms of backend technology, it doesn't matter too much what is used, as it won't be used too much, but in my case I'll be using Node.JS with an Express server, simply because it's easy to deploy and install dependencies ("npm install" can be run to install dependencies in one command), and it can also connect to a database to provide services such as login, and records of individual files.

Distributed System Management As this system consists of several smaller systems, it's important that these are managed. For each smaller component, a Docker container can be created, containing the configuration between each one. From here, Docker Compose can then be used to manage the entire system, bringing everything online, opening ports, and building everything automatically.

This requires us to create a Dockerfile per system (one for the render server, one for the client front-end). This Dockerfile specifies the image to build from, along with instructions that need to be run, and files that should be shared, to make that container work. Rather than customising an individual image, this customisation can be done by building on existing images, saving time without needing to start again from scratch. (Kent 2014)

TODO: some source of docker

F Evaluation

III REFERENCES

References

Ellingwood, J. (2015), 'Apache vs nginx: Practical considerations'. Accessed: 2018-01-10.
URL: *https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations*

Kent, A. (2014), 'Docker, distributed systems, and why it matters to magneto (part 1)'. Accessed: 2018-01-10.
URL: *https://alankent.me/2014/08/28/docker-distributed-systems-and-why-it-matters-to-magento-part-1/*

Melnikov, A. & Fette, I. (2011), 'The WebSocket Protocol', RFC 6455.
URL: *https://rfc-editor.org/rfc/rfc6455.txt*

Mozilla (2017), ‘Xmlhttprequest’. Accessed: 2018-01-09.
URL: *https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest*

Verhoeven, G. J. J. (2010), ‘It’s all about the format unleashing the power of raw aerial photography’, *International Journal of Remote Sensing* **31**(8), 2009–2042.
URL: *https://doi.org/10.1080/01431160902929271*