RAWFlash: A cloud based RAW image editor

Student Name: Ryan Collins

Supervisor Name: Dr Tom Friedetzky

Submitted as part of the degree of MEng Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

April 24, 2018

Abstract —

Context/Background RAW images are used by photographers to improve the quality of their images, specifically when editing. With JPEG files, the loss of data from compression leads to poorer quality edits, while using RAW results in a higher dynamic range. The editing of these files is traditionally done on a high-spec machine, capable of processing images with large resolutions. In the field, this can be difficult, as these machines tend to be heavy and require power. Furthermore, for amateur photographers, or large scale publications, a thin-client approach to photo editing would require lower specification machines for the office, instead relying on larger render servers. This could, therefore, minimize costs. Furthermore, by creating a system for RAW editing as a service, this system could be utilized within other software, to allow developers to focus on their system, rather than implementing their own method for editing RAW images.

Aims The aim of this project was to test the feasibility of a Cloud-based RAW image editor, both by providing a RAW rendering service, and interfacing with this service through a GUI.

Method The render server was implemented using Java, with a variety of plugins to allow for RAW image parsing (Collins (2017)) and TIFF reading (Kuhr (2018)). An input to the Java render server is received by fetching jobs from a message queue, carrying out the specified rendering instructions, and sending the result via a message queue (and client ID), to the client that sent the job.

A render dispatcher was also implemented, using Node.js with Socket.io, to take the user-requested operations, store the user information in the request (such as their client ID), and send the request to the render server message queue. Additional routines were also written to receive the rendered result, and forward this to the user who requested the job.

In order to interface with our render dispatcher, a web editor was created using HTML, CSS, and JavaScript, to build up the RAW rendering instructions, using a user interface rather than through editing a JSON file. This provides an easier way to interface with the system, but also a modular approach, as this isn't required to be used.

Results Utilizing a sample RAW image, output from various different RAW editors was analyzed and compared to our system, to determine the image quality of our system, when editing. The functionality of our system was also tested using boundary conditions and test cases, to ensure the system was functioning correctly. Furthermore, statistics on the processing times of different RAW editors were analyzed, to analyze the performance of our system for standard human-based RAW image editing.

Conclusions Cloud-based RAW image editing is appropriate for service-based RAW image rendering, where time is not a major constraint. For use as a RAW processor in an existing system, our image editor is ideal, where the latency of rendering an image isn't a major concern. For human-based RAW image editing, our system isn't as suitable due to the time delay, although editing is still possible, but more difficult to achieve.

Keywords — RAW image editing, dcraw, cloud image editing, docker, image processing, distributed systems

I INTRODUCTION

The project itself concerns itself with editing RAW files through a system deployable to the cloud so that it can both be used by people editing photos, and developers looking to work with RAW image formats within their own software.

A RAW Files Explained

RAW isn't a file format in itself, but rather an umbrella term defining many different (mostly proprietary) formats that store the raw camera sensor data. The methods used to store this information is quite often different, making the process of reading RAW image formats more challenging.

When an image is captured, the camera sensor uses charge buildup to represent the amount of light that is incident on the sensor (using a phenomenon known as the photoelectric effect). Color data itself isn't stored at all in the RAW image. Color is achieved by putting a filter over the sensor, such that each individual part of the sensor captures only red, green or blue light, which then allows one to build up a full-color image. The process of creating a full-color image from the camera sensor is called demosaicing. Fraser (2004)

There are several different steps to decoding RAW images, in addition to demosaicing which was explained above. These are:

Color Balance The individual red, green and blue components of the image can be adjusted, perhaps to correct for incorrect sensor capture. This is done by setting multipliers on each individual component, to build up the final image.

Gamma Correction Cameras typically represent Color changes linearly, with a gamma of 1.0. By adjusting gamma, the tonal response of the image can be adjusted, to help produce a desired appearance.

Sharpening/Noise Reduction Image noise is the random fluctuation in image brightness in digital camera images, but more loosely can be defined as anything that somewhat distorts the original captured image (the target).

Sometimes, image noise can creep into the image, particularly if the image itself is shot with large ISO levels (ISO is a sensitivity setting that can be set when taking a photo, to allow the camera sensor to be more sensitive to light. This can be corrected during the editing process.

Also, sometimes edges might need to be enhanced to bring out the detail in the image. This can again be used for stylistic purposes, but this is ultimately the decision of the image editor.

A.1 Comparison to JPEG

JPEG, a common format in the consumer photography market, often has smaller files due to JPEG compression. While this can be good to minimize the amount of memory used, the lossy compression is problematic as adjusting the tone and color information as JPEG heavily compresses colors (so minor adjustments in color will appear as the same color in JPEG). When shooting an image with JPEG, the camera uses a built-in RAW processor to automatically process the RAW data, perhaps using some settings that can be adjusted in the camera and compresses the image down into a JPEG format.

On the other hand, RAW allows one to have far more control over the final image, by adjusting various settings with a far greater freedom and by using the camera's built-in processing.

Fraser (2004)

B Applications of RAW Image Editing

B.1 Image Production/Publishing

In industries where a large amount of RAW image editing is required, a thin-client based approach to RAW image editing might be desirable. Instead of equipping the office with a large number of high specification machines, one could instead use lower specification machines, along with a Cloud-based RAW editing system, along with some servers that can conduct the RAW image rendering. This would reduce overall costs, as office machines won't need to be replaced as often to keep up with the requirements of native editors, as this can be done on the server. Providing the office machine can run a web browser, then it is suitable.

B.2 Content Management Systems for News Websites

When editing certain types of websites (e.g. news) where there may not be time to take a RAW image, edit it natively on a machine which may not be local, a Cloud-based RAW editing system could be integrated into the content management system (CMS), allowing the user to upload the RAW image, carry out basic edits from within the CMS, and publish their news story. While processing power might be harder to come by when traveling (e.g. with only a mobile phone, or low power and lightweight laptop), fast internet access is more prevalent due to 4G mobile networks. As such, uploading a RAW image might be preferable to carrying around a heavier and more powerful laptop computer.

B.3 RAW Editing as a Service for Developers

Currently, there doesn't exist an easy method of integrating RAW image editing into a bespoke system. As a developer, one would need to implement a RAW processor, or integrate an existing command line application into the pipeline. By creating a RAW editor as a Service, the developer can focus on the more important parts of their application, without needing to spent a large amount of time implementing their own solution to parsing RAW files.

C Problems with Current Implementations

Most current implementations exist as native applications, designed specifically for one or several different platforms. With this, a specific machine shall have the program installed, and shall contain all the features from managing files, to rendering images, to displaying the output to the user as required.

However, larger images can require a larger amount of resources to render properly, and furthermore if one intends on editing RAW images on the go, transporting a larger machine isn't convenient. Furthermore, the RAW editing software might not necessarily be available on the platform that is nearest to the user.

Instead of this approach, this project is taking an approach of having a Cloud system that can render images, which can scale if necessary by adding more servers, so whatever client device is used, raw images can be edited providing the user has an internet connection (which with the recent presence of 3G and 4G, they are likely to have).

Furthermore, due to the proprietary nature of most RAW processing applications, using them as a component in another system becomes very difficult, and by creating a system that can be controlled by other external systems can provide RAW image editing as a service.

D Project Objectives

The project objectives can be expressed as Minimum, Intermediate and Advanced Objectives

Minimum Objectives

- Exposure adjustment
- Noise reduction methods (Gaussian, mean)
- Web Interface interacting with an image processing server
- Non-destructive image adjustment (i.e. no reduction in quality over time)

Intermediate Objectives

- Load DNG RAW files by upload
- White Balance Adjustment
- Gamma Correction
- Modern, user friendly User Experience
- Exporting to other formats

Advanced Objectives

- Addressing potential scalability issues
- Haze removal
- Image Sharpening

II RELATED WORK

Research into a Cloud-based solution for image processing isn't new. Liang u. a. (2009) outlined a prototype system for editing standard images through web services. However, for greater image quality, RAW files are typically used over standard images, and with these files come larger file sizes, no compression, and various extra steps in the image processing pipeline, such as demosaicing. In order to do this, while the basic architecture in Liang u. a.'s work is sound, it isn't designed to be used through a web browser, so the User interface layer will need to be modified, and our services implementation layer will also need to be heavily changed to deal with RAW images, their editing, and the output image.

This section outlines the various changes needed, from the changes needed to parse RAW files, to existing RAW image editors.

A RAW files, and parsing

As explained in the introduction, there are many different types of proprietary RAW file formats, such as Canon's CRW and CR2, and Nikon's NEF among others. Each of these formats stores similar information, but in a different way.

Adobe created and made public their standard RAW file format called DNG. It is defined as "a non-proprietary file format for storing camera raw files that can be used by a wide range of hardware and software vendors" by Adobe in the specification. Adobe Systems (2012) Some cameras use DNG as default, for example Pentax.

B Existing Solutions

Adobe Lightroom Adobe Lightroom is a native and proprietary tool, developed by Adobe, for editing images, both RAW images, and otherwise. RAW processing in Lightroom is done by Adobe Camera RAW (as Lightroom builds on this useful tool). Adobe Camera RAW (2017) Adobe also provide a DNG converter, that converts RAW files from supported cameras, to their DNG standard, for editing. As DNG is an published specification, files in this format are more likely to be supported when compared to the camera proprietary formats. However, the software itself is only available for Windows and MacOS.Adobe DNG Converter (2017) While this isn't necessarily always a problem, for the creation of a portable system this could be problematic, as other operating systems such as Linux are growing in the Enterprise market, representing 20.7 percent of all server revenue, with Windows accounting for 50.2 percent Nagel (2012). Furthermore, Windows deployments on Cloud systems such as Amazon Web Services require payment of licenses to Microsoft, which provide extra costs. The construction of a multi-platform system is more suitable, not relying on single OS applications, that are proprietary to some degree.

RawTherapee RawTherapee is an open source native RAW image editor, which relies on a patched version of dcraw to parse RAW file formats. Their implementation is cross platform, and released under the GNU GPLv3 license. As a native editor, the user interface is provided locally through the QT framework. The use of dcraw is quite interesting, as it follows the open source license of RawTherapee. Raw Therapee README.md (2018) It also provides non-destructive image editing, which is what would like to achieve.

Darktable Darktable is another open source RAW editor. It is also cross platform, running on Linux, Mac, Windows, BSD and Solaris. While some components of dcraw are present in the application (namely the de-noising performed for RAW being ported), the application is based on the rawspeed library. Darktable switched from using dcraw and libraw to using their own RAW library called Rawspeed. Their library loads RAW images for later processing on floating point values, to improve quality. About Darktable (2018)

dcraw DCRAW is the tool used by RawTherapee to process RAW images, and also previously used by Darktable. Unlike the above options, it is a single portable executable that's controllable through the command line. It is portable, using on C standard libraries, and therefore doesn't require any extra build steps to use dependencies. The creator doesn't release dcraw as a library directly, as command line applications cannot use global variables, so an executable is more ideal. The recommendation by the creator is to run it as a separate process, to make it modular. Other users have also used the C file as the basis for their own custom RAW processing engines, such as RawTherapee, among others. Coffin (2017)

libraw Libraw is based on dcraw, but rather than supplying the interface as an executable, it provides the interface as a C++ and C library. It also extends dcraw by fixing bugs and removing some drawbacks such as dealing with EXIF data. Tutubalin (2008).

III SOLUTION

This section presents the solution to the problem, explaining the implementation detail of how the system was constructed. There are many ways of solving the problem, so some other options were also outlined, along with a comparison between them.

A RAW Processor

One key problem is the processing of RAW images. While there are several options out there, LibRaw and Dcraw seem the most popular ones, as they have both been heavily used in production. Libraw is just a library based version of dcraw, for C++. While this might seem like the most optimum solution, the documentation available for libraw is minimal, and integrating it into various C++ build systems is tricky. Furthermore, for our application, using a low level language such as C/C++ directly, for interfacing with the web is not ideal, as potential security issues can arise from interfacing with the metal. Furthermore, linking C++ directly to the web systems might become tricky, and the lack of a dependency management system that works for every dependency means that ensuring the system is portable, and can be built automatically on multiple different systems is not as easy to implement.

As dcraw is just an executable, a wrapper around the command line executable can be written, to deal with the first stage of RAW rendering. Therefore, a simple library for interfacing with dcraw was created for Java, and open sourced. Collins (2017). This way, rather than manually specifying the arguments for the executable, the images can simply be edited by supplying instruction operations (in the form of Java Objects). The output of the library will be a TIFF file, rather than the default PPM file. This is done as TIFF is a file that can be directly read by Java (with the use of external tools). To read TIFF, the TwelveMonkeys extension Kuhr (2018) was utilized, to read and write TIFF, meaning the uncompressed TIFF can be read from dcraw, and then processed further using custom image processing commands.

B Client-Server Communication Protocol

In order to communicate between the client side interface (where the user provides inputs and instructions for rendering images) to the server side (the entrance of the rendering system, which manages jobs), we need a method for transmitting the correct information in an appropriate manner.

B.1 Communication Methods

Representational State Transfer (REST) The REST method uses URL routes to send/retrieve information, using HTTP methods such as GET, POST, PUT, DELETE.

The solution would be to send a job, and keep sending requests to determine the status of the job, through a method known as polling. When the user sends a request, a *202 Accepted* response should be sent, indicating that the job has been accepted, along with a web resource URL to be used to fetch the outcome of the job. This resource URL would return an estimated time until completion, when queried, while the job was being, or waiting to be processed, or would send the result of the job, if the job was completed. This conforms to the HTTP/1.1 protocol. Farazdagi (2014)

Web Socket Web Socket is a new technology that allows easy two-way communication between client and server, rather like how sockets work within normal network programming. The client would send a message to the server with the required render instructions, and when the job is complete, the server would return the result to the user. Listeners on both sides wait for messages to be received, and process the messages in the appropriate manner. Internet Engineering Task Force (2011) This lends itself quite well to our method, as with the user constantly changing settings and requesting a render (to view the preview), and repeating the process as settings are adjusted, two way communication seems far more appropriate for this application.

However, Web Socket does have downsides. As a fairly new technology, it's not universally supported, and only modern browsers support the protocol, and potentially some firewalls might block ports required, and some hosting providers might not support it either. Therefore, while web-socket is ideal, compatibility might be a problem. As shown in Mozilla Web Socket Developer (2018), Internet Explorer, Opera, and Safari don't have much support aside from early versions of the specification (RFC 6455). Other browsers do have increased support, such as Firefox and Chrome. Depending on the browser, support will vary.

Socket.IO Socket.io itself isn't really a protocol, but an implementation of both REST and Web-socket, designed for event driven applications. Where web-socket is supported, it is used, but where web-socket isn't supported, the system falls back to REST to transmit and receive data through polling. Implementation wise, the system is built in an event driven way, with message handlers being written, and the framework deals with the edge cases with compatibility. MacCaw (2011)

However, a problem arises with this, as this is only fully supported in a limited number of languages, namely JavaScript (both client and server side), and unofficial support is available for Java. Therefore, JavaScript shall be used for the client side (which is really the only option anyway), and Java for the server side, due to the Image library provided.

Furthermore, the added benefit is one of state: in the event the server goes down temporarily, providing the session is still open (i.e. the user hasn't closed their browser window), the system will reconnect when the server comes back online. Providing we implement some stateless system (where all the data needed is provided within the request), then this will help improve the reliability of our system, not requiring users to repeatedly restart their session if some error occurs in the dispatcher.

Unlike REST, and much like web-socket, we have the added complexity of identifying users, so that data is sent back to the correct user. This can easily be done by finding a client unique ID, and sending a message to the connection with that ID. This is an additional piece of information that needs to be included in the request between the dispatcher and the render server (to ensure that when the response is received, the response itself is forwarded to the correct person).

Therefore, this is the solution I've decided to use.

C Rendering Server Structure

The final structure of the render server can be seen in Figure 1.

When designing this section, several different structural methods were considered. Initially, a local cache wasn't implemented, but this was added to improve performance (as discussed in Part E).

Initially, the aim was just to have one single monolithic structure to the image processing, with the same structure dealing with the entire process, from reading in RAW to returning the result. However, the decision was taken to implement these separately, as it would allow for the ability to swap the RAW processor used. This could be useful if for example support is dropped for dcraw, or if a better RAW processing library is created (perhaps one that doesn't require file writes). The DCRAW Manager therefore deals with the Dcraw processing, by interfacing with the custom JDCRAW library that was created

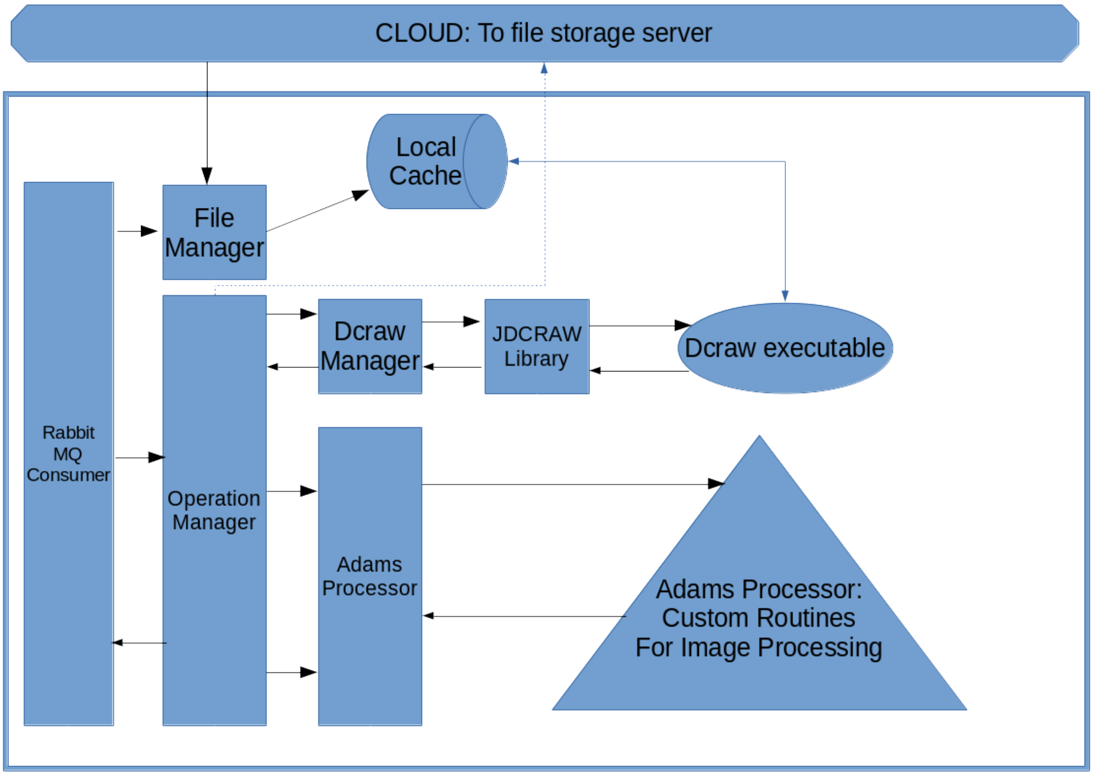

Figure 1: The structure of the render server. The local cache is simply stored on the server local file system. RabbitMQ is a message queue - every single render job is added to the message queue, and the render servers take a job from the message queue to process. There are two queues, one to store jobs to process, others to store the results of jobs. When a job is finished, the result is added to the second queue, and this is then passed to the user.

(but open sourced as a separate module). The creation of this module further abstracts DCRAW from our system.

The entire image editing process however is managed by one class, the Operation Manager. This is responsible for taking the user specified instructions and passing them to all the possible processors (currently only DCRAW and Adams are implemented, but potentially many more processors could easily be added, perhaps using class loading in the future).

More detail about the Adams Processor, which is the name for all the custom image processing implementation can be found in Part D.

D Adams Processor

The Adams Processor, named after the famous photographer Ansel Adams (known for his black and white film landscape photographs), is the name given to the main routines powering our application. This deals with processing the image after generating the RAW image.

D.1 Gamma Correction

The process of gamma correction involves taking the RAW image, which by default has a linear tone curve (gamma of 1.0), and mapping the colors to a different tone curve.

The function f is used as a transformation from the image, to the gamma correction image. f is defined as:

$$f(\gamma, v) = 255 \cdot \left(\frac{v}{255}\right)^{\gamma}$$

In the above definition, v is the brightness value of one cell of the image, where $0 <= v <= 255$. First, we transform the brightness of the individual channel (red, green or blue) to a floating point number, by dividing it by the maximum value for the channel, which is 255. Raising this to the power of the gamma factor applies the transform, and then multiplying it by 255 converts the target floating point representation back to the representation of a standard image (to be stored in a byte per channel).

On the implementation level, the best approach is to use a Lookup Table, with an input of the RAW image components (in our case, Red, Green and Blue), and an output of the adjusted component values in accordance with the gamma correction function. The benefit of using a lookup table, rather than calculating the gamma value for every single possible value, is that one doesn't need to calculate the same piece of data more than once (as bright regions, or dark regions might potentially have the same input values).

D.2 Noise Reduction

Within an image, noise can be introduced by various means, including excessive ISO settings. Therefore are several different types of noise, including Amplifier Noise, Salt-and-pepper noise, Shot noise, Quantisation noise, on-isotropic noise, speckle noise and periodic noise. Hambal u. a. (2017)

Amplifier Noise This noise typically occurs with different sensitivities for different channels (e.g. red, green and blue detectors have different sensitivities). Hambal u. a. (2017) This can be removed by applying a Gaussian Blur. Gonzalez und Woods (2009)

Salt-and-Pepper Noise This noise is random noise, where random pixels might differ greatly from their neighbors. This noise can be caused by dead pixels on the camera sensor. Hambal u. a. (2017)

D.3 Unsharp Image Adjustment

Unsharp Image Adjustment (masking), is one way of sharpening images.

Given an input 2D image $I(x, y)$, applying a Gaussian Blur to this image will yield another image $I_{smoothed}(x, y)$. By doing a simple calculation, we can calculate the value of the mask $g(x, y)$. The parameters of Gaussian kernel size and sigma can be used as additional parameters.

$$g(x, y) = I(x, y) - I_{smoothed}(x, y)$$

By using this, one can then sharpen the image by using the output of this:

$$I_{sharpened} = I(x, y) + \lambda \cdot g(x, y)$$

.

Here, λ is defined as the amount in the Unsharp filter.

Fisher u. a. (2000)

When using the RGB Color space, with Unsharp on each individual channel, the colors can appear warped, so using the HSL Color space instead can reduce this, and also reduce the computation. The HSL Color Space is a way of storing images, storing the Hue, Saturation, and Luminance for each pixel. The hue represents the Color, the saturation represents how prominent the Color is (the purity), and the luminance controls the perceived lightness of the color. This conversion is beneficial because only the luminance channel needs to be modified, rather than all three channels. However, one key problem is that the process can yield values of luminance (L) that exceed the $0 \leq L \leq 100$ range in the color space. Furthermore, if this step isn't carried out, clipping can occur, which yields an overall loss of image data, something that isn't desired in a RAW editor.

This problem can be solved by normalizing the image, by a process called contrast stretching. The max and minimum luminance values are found across the entire modified image, and these are used to scale the luminance values. Using standard Contrast Stretching does work, but at times tonal curves can look weird. By manually applying gamma though, this problem can be solved.

One major problem regarding the Gaussian filter is that the user can potentially choose a large Kernel size, which results in increasingly lengthy periods to process. The alternative is to associate the SIGMA value of the Gaussian with the radius, detailing the standard deviation (spread) of the Gaussian over a fixed kernel. The Mean blur filter remains as before (though this feature is disabled through the web interface, as Gaussian is primarily used).

One problem encountered in implementing this was lack of HSL support built into Java. Instead, an implementation of an HSL colour converter, was used, as documented in Jianzhao (2015)

D.4 White Balance Adjustment

Each color image is made up of a variety of individual images, which are placed together to produce the color images. These are the Red, Green and Blue masks.

When one adjusts colours, our aim is to mix these three images together with a variety of different amounts, so that the overall image produced blends them together to create white (rather than an orange). The white balance control here can be adjusted further to tint the entire image.

This is important in photography, as light sources don't always emit a uniform white light, but rather have some colour tint to it. As a result, the image that is captured will have colours appearing incorrectly (whites will appear orange in some cases, or other shades). This effect can be removed in editing, but carrying out White Balance adjustment, also known as colour adjustment.

In order to do this, we can simply scale the components of the internal camera RGB model, which can be done by adjusting the coefficients of these channels, which create a final colour image. In Viggiano (2004), Viggiano found that less distortion of colours was produced when using the camera RGB representation rather than monitor based RGB, as we are using the RAW format, we can store the camera representation, therefore producing less distorted colours as a result.

E Improving performance of fetching RAW files

One of the biggest sources of delay in our system is the necessity to obtain RAW files. The user provides a URL pointing to the RAW file in their render request, and then the system downloads the file from that URL to use as the RAW file. Doing this for the same image constantly is very time consuming, and as the user will likely be making many edits on the same RAW file, many downloads will be needed.

Therefore, implementing some form of caching is needed, to ensure that this download step only occurs once, and then afterwards the cached RAW file will be used. Caching works in this situation, as when rendering, the RAW file will not change at all (it's read only from our point of view), meaning that we obtain the maximum quality through edits.

In order to do this, we need to save the image to the render server local disk, without any filename collisions (where two different RAW URLs are cached to the same filename on disk).

The solution to this is to use a Universally Unique Identifier (UUID).

Our method uses the RAW file URL as the basis for the UUID generation, generating a Type 3 UUID based on the full path specified. When downloading a file, the system first generates this UUID filename, and checks the local file system. If the file is present, then it is already cached, and therefore the cached version is used. If the file isn't present, then it isn't cached, and a file is downloaded, and written to a filename based on the UUID (in this case, the format will be UUID followed by the extension). Type 3 UUID generation is based on the MD5 hashing algorithm, by generating a 128 bit UUID. The hexadecimal representation (ignoring the dashes), is used as the replacement filename.

For a large number of RAW files being processed, using Type 5 UUIDs might be more suitable (as the algorithm used is SHA1 with a larger entropy), but for a prototype use with few image URLs, this is more suitable.

One other concern is the possibility of deleting files. While this is a grave concern typically, the current system doesn't have a time decay of caching, simply because RAW files might be worked on over a period of days, and furthermore as the system is stateless, all files stored on the local system shall be deleted regularly (configured inside Docker). Restarting render servers regularly is ideal, as this way we prevent memory leaks from the Java BufferedImage implementation which have been minimized in our implementation but not completely eradicated.

F Web Editor Functionality

The web editor itself provides an interface to the RAW Editing Service. Rather than manually adjusting the settings and sending requests to the service, the parameters shall be controllable through a

Graphical User Interface, with the image result also being displayed on the screen.

Furthermore, one must be able to undo and redo particular tasks. This is because the RAW editing process typically requires refining parameter values, testing, and then opting to use the previously set value. Having an undo/redo system would make this easier.

F.1 Undo/Redo Functionality

Due to our implementation, coupling that with the nature of our system, no current undo/redo queue library seemed to be quite an ideal fit for our system. Therefore, it's necessary to build our own.

The system should, on modification of any parameters, store the change in some data structure, so that it can be reverted.

This entire system relies on using a stack. When a change is made (one change shall be made at a time), both the name of the parameter edited shall be stored, along with the old and new values. This shall be stored together in an object, and pushed to the stack. To undo, one simply needs to pop the stack, and set the specified parameter (given by the parameter name) value to the old value specified.

In order to enable redo functionality, we create an empty stack, called the redo stack. When the undo function is called, the object popped from the stack is also pushed to the redo stack. If the redo queue is not empty, and a normal change in parameter has occurred, then the redo queue is emptied (one can no longer redo).

There are also a few edge cases. If someone presses the undo button when there are no actions to undo, then nothing can happen, as there is nothing to undo. The same is true for the redo button.

G Storage of RAW Images

The web service was constructed in a manner that allows RAW images to be fetched from any valid URL. Instead of limiting the user to using a particular storage backend such as Openstack Swift, or Google Drive, any location accessable by a URL can be used.

Initially, a Django-based system was partially implemented to allow the user to upload an image, and choose it to be edited, but this was abandoned as it wasn't entirely relevant to the overall goal of the project.

H Managing a big system

The architecture of our system is shown in Figure 2. As our system design calls for a large number of individual components, managing all of these together can be quite complex.

One option is to write a script that deals with the setup of each individual component: web server, web services, render dispatcher, render servers (however many we need), and message queue. However, if a component of the system stops working, the error will bring down the system entirely, and writing a script won't necessarily allow us to deploy the system across several different machines.

Our solution to this problem is to use Docker. Each component in the system shall be given its own individual container (essentially a virtual machine), where it is sand-boxed to ensure that systems don't interfere with one another, and each container is networked together. Rather than manually setting up the networking, one can use Docker Compose to automatically deal with the links between different containers, to ensure that private links between containers are set up, but these links can't be accessed from outside the system. This method is preferable, as then one cannot access the message queue/other sensitive components directly. That way, only by sending the proper request, will the system respond with an output.

With a micro-service architecture, and some element of load balancing is recommended, along with some method of deploying services together so that dependencies are met. These lessons learned from Balalaie u. a. (2016), were implemented by implementing our message queue backed render server, to load balance requests, and to use Docker Compose as Balalaie u. a.'s work also did.

I Unit Testing Issues with Image Processing

For one to compare the output of the system with other RAW editors is not necessarily as easy as carrying out a bitwise-and of the files, and finding the percentage differences. While this might work if

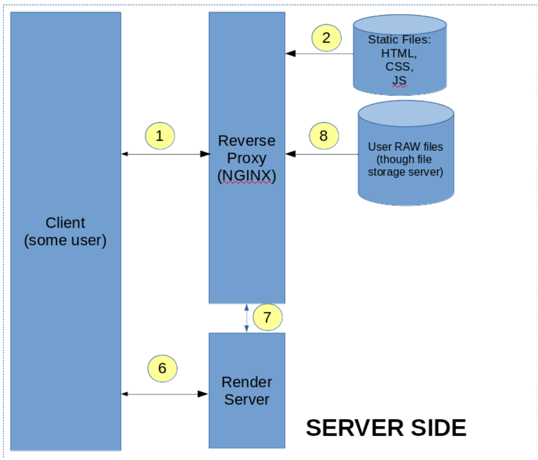

Figure 2: The architectural structure of the system. (1) File download of the client-side user interface. (2) Static files served through the web server. (3, 4, 5) These were part of a system to allow users to upload and manage their collection of images. This was not implemented due to time constraints, and it was not directly related to the overall goal. (6) Socket.io communication between the client and the render service, to generate previews and output images. (7) RAW Images are fetched from the web server. (8) Test images can be served through the web server too.

all systems use the same algorithms to do the same tasks, due to the proprietary nature of RAW image editing, different implementations of demosaicing exist, yielding slightly different images, along with potential different ways of adopting equalization, gamma correction and various other colour adjustments (for example, RGB gain was used to adjust the colours, while Lightroom instead uses chromacity). It'll be difficult to use the same settings for every single system. Instead, a better solution would be to use a test image, and compare the adjustments by eye between different applications, and use base cases or other expected behavior to test the output of the system.

I.1 Base cases include:

Gamma Correction Setting $\gamma < 1.0$ will yield an image that appears whiter than expected. Setting $\gamma = 1$ should yield the input image. Setting $\gamma > 1.0$ will yield an image that is darker in tone.

Colour Balance Setting red gain to 1, and all others to zero will yield an image with only the red channel. Doing the same for green and blue should reveal the same. This will ensure the algorithm is working correctly.

Exposure Exposure zero should yield a completely black image. Using a very high exposure should yield a completely, or nearly white image.

IV RESULTS

This section outlines the results, to test the system against the deliverables and to compare it to other systems that are already in production. The section aims to show that our solution works, and provides results to determine how well it works.

A Functionality Testing

The functionality of the system was tested using a mixture of boundary checks, and eye based tests. Parameter values were specified with the expected outcome, and the output of the system was tested against this.

Figure 3: Test of Exposure Adjustment (a) Exposure Level Small: 0.0001 (b) Exposure normal: 1.0 (c) Exposure level high: 5000

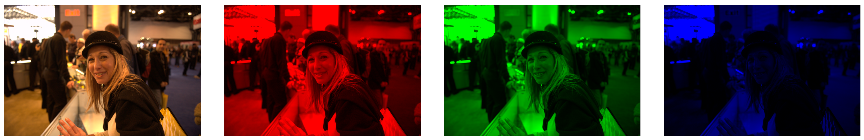

Figure 4: Test of Colour Balance (a) No Adjustment (b) Red only (green and blue zero) (c) Green only (red and blue zero) (d) Blue only (red and green zero)

A.1 Exposure Checks

Various different exposures were checked. With an exposure close to zero, a black picture should be yielded. With an exposure of 1.0, the image should appear at a normal brightness. With a very high exposure, the image yielded should be almost white, with some slightly darker patches (which should have some tint).

When testing, an exposure value of zero yielded an inverted colour. However, an exposure of 0.0001 was used, and this gave the expected outcome.

The overall outcome of these checks can be seen in Figure 3.

A.2 Colour Balance Checks

Colour balance is controlled with red, green and blue gains. If we set only one of these gains to 1.0, and the other two to zero, we should have a version of the image, but only with the specified colour, present: rather like a grayscale image but in one specific colour rather than black to white. The results of this check can be found in Figure 4. These results show that the colour balance algorithms are working correctly.

A.3 Unsharp

Enter a fairly high Unsharp radius (the sigma of the Gaussian), and amount. This should then yield an image with enhanced edges in comparison to the normal image without Unsharp masking. As a reference point, a specific part of the image should be used to compare the sharpness, to determine the functionality of the implementation.

Reducing the unsharp amount to zero should cause the output image to be the same as the input image.

The output is shown in Figure 5. As one can see, the unsharp output makes the hat edge sharper. This means that the routines are working correctly.

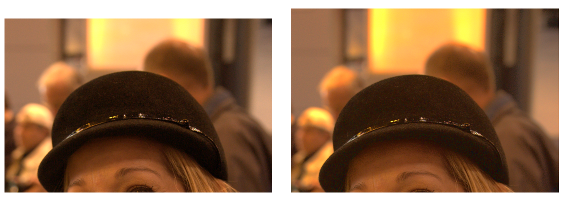

Figure 5: Test of Unsharp output. (a) Without Unsharp applied (b) With Unsharp applied

Figure 6: Test of Gamma Correction output. (a) $\gamma = 0.3$ (b) $\gamma = 1.0$ (c) $\gamma = 1.7$

Processor	Intel i7-7700k, 4.2GHz
RAM	16GB at 3000
HDD	1TB Seagate Barracuda, spinning at 7200RPM
Operating System	Debian 9

Table 1: The specification of the test system.

A.4 Gamma Correction

Different gamma factor values were entered into the system, in accordance with the expected cases. Gamma factor is defined as γ.

For $\gamma < 1$, the output image should appear with a whiter tone.

For $\gamma > 1$, the output image was expected to appear with a darker tone.

For $\gamma = 1$, the output image was expected to be the same as the input image.

The outcome of this test can be seen in Figure 6. The implementation yields this expected outcome.

B Comparison With Other Editors

In order to evaluate the performance of the system, two methods were used: comparing the overall output of the image, and measuring the responsiveness in seconds. For all of these tests, a set of different image manipulations were carried out on various different RAW editors, along with our system. These were all executed on the same hardware, with the same operating system.

In order to evaluate the performance of the system, both in the output image, along with the overall system performance, responsiveness (measured in delay in rendering an image), a set of different image manipulations were carried out on various different RAW editors and our system. These were all run on exactly the same hardware, with the same Operating System.

The system specification used can be seen in Table 1.

Adobe Lightroom, RawTherapee, and Darktable were used to compare the performance of our system. For the comparison results, a RAW sample was obtained from the D700 camera.

B.1 Test 1: Exposure Adjustment

In order to compare exposure, various different exposures were used. Some by decreasing the exposure level, others by increasing the exposure level, and leaving the exposure level at 1.0

B.2 Test 2: Sharpening

While these are defined by different terms between software programs, we can still test the overall outcome. Rather than deciding the quality of sharpening by how pronounced the edge is, it was instead done by comparing a portion of the image (the brim of the hat), sharpened between different applications. The results are shown in Figure 7.

In terms of quality, while RawTherapee and Darktable perform slightly better, RAWFlash does sharpen the image, though leaving some noise. Each system however did enhance the brim of the hat, sharpening the line and making it less blurry. Our system enhanced the edge enough so that it could be seen, but not so much as to make it too pronounced.

Figure 7: Test of unsharp sharpening. The brim of the hat is used as the edge to compare. (a) RAWFlash, our system (b) RawTherapee (c) DarkTable

Figure 8: Test of Gamma Correction output. (a) RAWFlash, our system (b) RawTherapee (c) DarkTable

B.3 Test 3: Colour Adjustment

Colour adjustment was carried out using the colour channels, and adjusting the different levels of red, green and blue that existed within an image. Between all three systems that were tested, the parameter values remained fairly constant (some use percentages, while others use decimals, but they all represent the same transform).

Each system appeared to act in a similar fashion. Some difference in colour was found, but this is likely due to the different way applications process colour spaces.

B.4 Test 4: Render Time

Render time was an important metric to measure and compare. Settings are often refined over time, and therefore quick render times for preview speed up the workflow. The time was measured between sending an image to render, and the rendered image appearing on the screen. The results can be seen in Table 2. Our system performed poorly compared with native editors, perhaps due to the lack of multi-threading, coupled with the network latency.

V EVALUATION

This section evaluates how well our system works, how well it solves the problem of RAW image processing as a service, and how it can be applied.

A Application as a RAW Image Editor

The system worked well at producing images, taking the RAW images and adjusting exposure, colour, and adding sharpening, and exporting the final image. The overall images produced were as expected.

Application	Time Taken To Render (seconds)
RawTherapee	0.3
DarkTable	0.3
RAWFlash	7.3

Table 2: The time taken between triggering a render and the image displaying on the screen measured in seconds.

However, the delay in producing the desired result (as shown in Table 2) could mean that the editing process is slower than other editors on the market. Therefore, for RAW image editing for standard consumers, this probably isn't as viable as a native RAW editor.

B Application to Web Content Management Systems

When applying our system to web content management systems, or for other applications that require a headless method of rendering RAW images, it functions well. With the CMS workflow, of uploading a set of RAW images, and having the system process them in batch our system would work better, as this batch processing will often be done in the background, producing the output that can then be stored, ready for use.

While this wouldn't be used in every case, for news websites this could be a potentially good tool, allowing RAW images to quickly be uploaded and edited anywhere, without a heavy amount of equipment.

C Performance Issues and Improvements

C.1 Problems with testing

While testing, it was found that the existing native applications differ in how parameters for different functions are specified. Therefore, specifying exactly the same parameters between different editors, to yield a direct comparison, is not possible.

In particular, Gamma Correction control wasn't able to be accurately compared with the other two editors, as both RawTherapee and Darktable require the gamma curve to be specified through a user interface widget, not allowing a gamma factor to be entered. Therefore, for gamma correction, we were unable to compare all three systems.

C.2 Problems with the Java Image library

As a basis for the Adams Processor image processing, the Java Image library was utilised. ImageIO was also used to load an image into a BufferedImage object, and various BufferedImage operations were then carried out on the image, in accordance with the instructions specified by the user.

In order to speed up the implementation, the built-in Java routines for convolution, and other common image adjustments were planned to be used. However, there were many problems overall with Java's Image implementation. These are:

High Memory Usage/Memory Leaks When processing images, high memory usage was often encountered. For example, while processing an image from a Nikon D7100, 6GB of RAM was used to process the image alone (not including any other components of the system).

When researching into this further, it was found that instances were not automatically being deallocated, causing the memory leak. As a fix, garbage collection was called manually after the response had been sent, which improved performance. Furthermore, smaller RAW images (from the D700, rather than the D7100) were used, as the test system was unable to cope with both acting as the render server, web server, and the client.

In order to fix this, a new Image representation is needed, that is able to deallocate memory properly, to help avoid the un-necessary bloat of BufferedImage.

Non-functioning Operations While implementing Gamma Correction, the plan was to utilize Java's *LUTOp*, to create a lookup table (LUT) that is based on the gamma value, and apply it to the image. When implementing this however, the image itself would not output any applied lookup, as the original image would be output. The LUT method generated the correct lookup table relation, and this was passed to the correct routine, but for some reason it was never applied to the image. Looking through documentation and examples didn't assist with this, as examples that were supposed to work didn't.

Eventually, this implementation approach was abandoned, and replaced with a custom approach of iterating through the image and applying the LUT manually. While slower (not relying on Java's faster implementation of yielding RGB values), it solved the problem and only appeared marginally slower.

Ideally, another image library could have been used, or a custom image library would require implementing these, but give more control in how they are implemented (e.g. allow multi-threading).

Lack of precision in Convolution Java's built-in convolution operator requires a kernel to be specified as a single precision floating point array. While this is fine for some uses, when using large kernel sizes with Gaussian Blur, a difficulty arises in representing the Gaussian coefficients within this precision. Therefore, a custom implementation was instead implemented, by creating a new *AccurateKernel* class which used double precision floating point numbers, and linking into Java's existing convolution operations.

Lack of support for multi-threading Initially, the plan was to apply multi-threading to convolution operations, but this didn't end up getting implemented due to the overhead of BufferedImage. As BufferedImage allows for multiple representations of images, it does not allow access directly to a primitive data structure storing the image, or some way of yielding one directly. The naive way by passing a BufferedImage object as a pointer didn't work either, as the overhead caused the system to run much slower than without parallellization.

While multi-threading itself isn't always wanted on image processing (with web servers, each request is assigned a process, and therefore if each process generated several threads it could cause problems), for our architecture, multi-threaded processing is suitable and encouraged, as there would be a performance increase.

In the future, building a custom image data structure would be ideal, as this would allow for multi-threaded processing of images. This would also massively improve the time taken to render the image, reducing the preview time down from 7.3 seconds to a figure more in line with RawTherapee or Darktable.

D Comparisons to Objectives

Overall, all but the Haze Removal objectives were met. Haze Removal involved many blur operations, which are very slow in our current framework, and therefore the time taken to render an image with haze removal would be far too expensive.

The output of the sharpening process was clear, but could have been improved further to yield sharper images.

VI CONCLUSIONS

Overall, the project showed that a Cloud-based approach of editing RAW images does work, and can work well for rendering RAW images in the current architecture. While the current implementation has some performance issues, these can be easily rectified, and the approach of editing RAW images as a service is very much possible. Using this approach directly in a browser would also work, although the delays in rendering the image on the server side make RAW editing slightly more work when compared with using native editors when using the standard RAW workflow.

References

[About Darktable 2018] Darktable (Veranst.): *About Darktable*. URL: `https://www.darktable.org/about/`. April 2018. – Accessed: 20-04-2018

[Adobe Camera RAW 2017] Adobe Systems Incorporated (Veranst.): *Camera RAW*. URL: `https://helpx.adobe.com/camera-raw/using/supported-cameras.html`. May 2017. – Accessed: 19-04-2018

[Adobe DNG Converter 2017] Adobe Systems Incorporated (Veranst.): *Adobe Digital Negative Converter*. URL: `https://helpx.adobe.com/photoshop/using/adobe-dng-converter.html`. October 2017. – Accessed: 19-04-2018

[Adobe Systems 2012] Adobe Systems (Veranst.): *Digital Negative (DNG) Specification*. June 2012. – Version 1.4.0.0

[Balalaie u. a. 2016] BALALAIE, A. ; HEYDARNOORI, A. ; JAMSHIDI, P.: Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. In: *IEEE Software* 33 (2016), May, Nr. 3, S. 42–52. – ISSN 0740-7459

[Coffin 2017] COFFIN, Dave: *Decoding raw digital photos in Linux*. URL: `https://www.cybercom.net/~dcoffin/dcraw/`. May 2017. – Accessed: 20-04-2018

[Collins 2017] COLLINS, Ryan: *JDCRAW: A DCRAW Wrapper for Java*. URL: `https://github.com/OhmGeek/JDCraw`. June 2017. – Accessed: 19-03-2018

[Farazdagi 2014] FARAZDAGI, Victor: *REST and long-running jobs*. URL: `https://farazdagi.com/2014/rest-and-long-running-jobs/`. October 2014. – Accessed: 19-04-2018

[Fisher u. a. 2000] FISHER, Robert B. ; PERKINS, Simon ; WALKER, Ashley ; WOLFART, Erik: *Hypermedia Image Processing Reference*. John Wiley and Sons LTD, 2000

[Fraser 2004] FRASER, Bruce: *White paper: Understanding Digital RAW Capture*. September 2004. – Accessed 10th April 2018

[Gonzalez und Woods 2009] GONZALEZ, Rafael C. ; WOODS, Richard E.: *Digital Image Processing*. 2. Prentice Hall, 2009. – ISBN 9781259081439

[Hambal u. a. 2017] HAMBAL, Abdalla ; PEI, Zhijun ; ISHABAILU, Faustini L.: Image Noise Reduction and Filtering Techniques. In: *International Journal of Science and Research* 6 (2017), March, Nr. 2, S. 201–213

[Internet Engineering Task Force 2011] Internet Engineering Task Force (Veranst.): *The WebSocket Protocol*. December 2011

[Jianzhao 2015] JIANZHAO, Jeff: *HSLColor.java*. URL: `https://github.com/jeffjianzhao/DAViewer/blob/master/src/daviewer/HSLColor.java`. January 2015

[Kuhr 2018] KUHR, Harald: *TwelveMonkeys ImageIO: Additional plug-ins and extensions for Java's ImageIO*. URL: `https://github.com/haraldk/TwelveMonkeys`. April 2018. – Accessed: 19-04-2018

[Liang u. a. 2009] LIANG, Y. ; XIAO, Y. ; HUANG, J.: An Efficient Image Processing Method Based on Web Services for Mobile Devices. In: *2009 2nd International Congress on Image and Signal Processing*, Oct 2009, S. 1–5

[MacCaw 2011] MACCAW, Alex: *JavaScript Web Applications: JQuery Developers' Guide to Moving State to the Client*. 1. O'Reilly, 2011. – ISBN 9781449303518

[Mozilla Web Socket Developer 2018] Mozilla (Veranst.): *Web Sockets*. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`. March 2018. – Accessed: 20-04-2018

[Nagel 2012] NAGEL, David: Linux Leads Server Growth. In: *The Journal* (2012), May. – Accessed: 20-04-2018

[Raw Therapee README.md 2018] RawTherapee Team (Veranst.): *Raw Therapee Readme*. URL: `https://github.com/Beep6581/RawTherapee/blob/dev/README.md`. February 2018. – Accessed: 20-04-2018

[Tutubalin 2008] TUTUBALIN, Alex: *LibRaw Project Goals and Objectives*. URL: `https://www.libraw.org/about`. April 2008. – Accessed: 20-04-2018

[Viggiano 2004] VIGGIANO, Stephen: Comparison of the accuracy of different white balancing options as quantified by their color constancy. In: *SPIE: The International Society for Optical Engineering* 6 (2004), March, Nr. 2, S. 201–213