Documenting Changes, Gjøvik city

Christian Hochlin

December 1, 2009

Contents

Chapter 1

Abstract

To create KML-files with a lot of images placed correctly is a tedious task to do manually, and a rather mechanical one too. This projects aim was to take advantage of the mechanical nature of making kml-files with lots of pictures, and automate it, to save time. It was based on a specific task, namely the 150th anniversary of Gjøvik city, which needed old images, and new images from the same angle presented side by side on a map. Bash-scripting were used, with exiftool as the only dependency, and the resulting script makes a KML-file with the paired images at the correct location, with little preparatory work for the user.

Chapter 2

Introduction

This project was made as a part of the Applied Digital Workflow course[1] at Gjøvik University College. It was possible to choose our own project, or pick one of the project suggestions. The project that was chosen focused on the city of Gjøviks 150th anniversary. To celebrate that, a number of old images were supposed to be re-shot, and presented with google maps. To do this by hand would require a lot of time and manual labour. The steps needed to complete the project would be:

- Reshooting old photographs
- Generating a kml-file with the old and new image juxtaposed at the correct location.
- Repeat for each photograph

Repeating this for each image would be tedious and very time-consuming. A lot of time could be saved by making a script to automate parts of the workflow. The question was how much of such a workflow that is possible to automate. The scope of this task was to create a more optimized workflow, with less manual labour needed, using freely available tools, scripted with bash-scripting in a UNIX environment.

The main script should be made with these statements in mind:

- Old images need to be paired with new images with correct geodata.
- The new images have relevant metadata embedded in them.
- The script should be tailored to this specific task

[1] http://www.ansatt.hig.no/kjellr/imt4951/

2.1 Planning

The planning started in week 43, and ended when work on the project began in week 46. See Pre-Project report for the full plan. The final Gantt-diagram ended up like this:

Milestones	Activities	Resources	Dependencies	Risk	Week	Hours
1.Plan					45	**20-25**
1.1	Research how to write	Library/Internet	-	-	44	5-10
1.2	Write plan	-	1.1	-	45	10-15
2.Final Report				Data corruption	49	**30-40**
2.1	Capture images	Camera	Camera available	-	46	5-8
2.2	Plan the script	Internet	-	-	46-47	4-6
2.3	Write script	”	2.2	Bugs	47	10-15
2.4	Test & optimize script	”	2.3	-	47-48	3-8
2.5	Write final report	Library/Internet		-	48-49	9-11

Chapter 3

Process

3.1 Initial segmenting

The project was split in two main parts, with one being the collection of data, and the second the following processing of the collected data. As specified in the Gantt-diagram, the collection of data and the processing part, were not dependent on one another. This means that they were interchangeable, and could be done independent of each other. The collection of data was dependent on a camera borrowed from the school, and the processing was not. That could be done with dummy resources that behaves as the real data, and would, on a later stage, be replaced by the real photographs. So even if the cameras were inaccessible much of the time, that would not slow down the work.

3.2 Deciding on dependencies

The scripts dependencies were important to decide on, to determine what could and could not be done. It was decided to depend on few third-party applications, to make the script as portable as possible. The two dependencies which were considered, was Exiftool and Imagemagick. Exiftool was impossible to do without, but whether or not to use Imagemagick was another question. Carefully weighing the pros and cons of using it had to be done. *Pros:*

- Able to make thumbnails for the maps-balloons.
- which means faster loading of the images in the balloons.

Cons:

- One additional dependency.

- Double the amount of images to upload.

In the end, Imagemagick was not used. The decision to use as few dependencies as possible outweighed the thumbnail-generating. With a little bit of HTML, it was possible to resize the large images and use as thumbnails, even though the loading time increased.

3.3 Re-shooting the images

An employee at Mjøsmuseet[1] helped pinpointing the location of a dozen old images from Gjøvik, and marking them on a map. The cameras arrived later than estimated, so this part was pushed back to week 47. Also, the camera was not able to supply the GPS-device with power, and no way to test what part (cable, connector) was broken due to sickness. This obstacle was overcome by using a GPS-enabled phone to capture photos at the same location. Armed with a map and a camera, the re-shooting was expected to take about an hour and a half. Turns out a lot of new buildings have been built the last century.

Some of the pictures were taken where it now stood large buildings, so some of the photos are only approximately taken from the same spot. Because of that minor inconvenience, it took a lot longer than estimated. The camera used was a Nikon D5000, and an Apple iPhone for capturing the location data. The rest of the time scheduled under "Capture images" in the Gantt-diagram were meant for setting up the camera to work with the GPS-receiver, and logistics associated with the re-shooting.

A simple script were made that looped through the images from the iPhone, and transferred the location data to the high quality pictures from the Nikon camera. This script is not included here, as it is not a part of the task.

3.4 Writing the script

The first thing that was needed was to decide what part of the workflow to automate, and recognize the limitations associated with the project. Some parts of the project obviously had to be done manually, given the time frame that was available. Pairing the old and the new images had to be done semi-manually, and some solutions to that had to be considered. What is the easiest way for a computer to pair two images? Compare a common property, and pair those that match. The question was what property to compare. Comparing geodata and pair those at the same location would be the most effective, and least laborious method, *if* both images were taken with a modern camera. Unfortunately, hundred year old cameras don't have GPS, so for this purpose, that approach

[1] http://www.mjosmuseet.no/

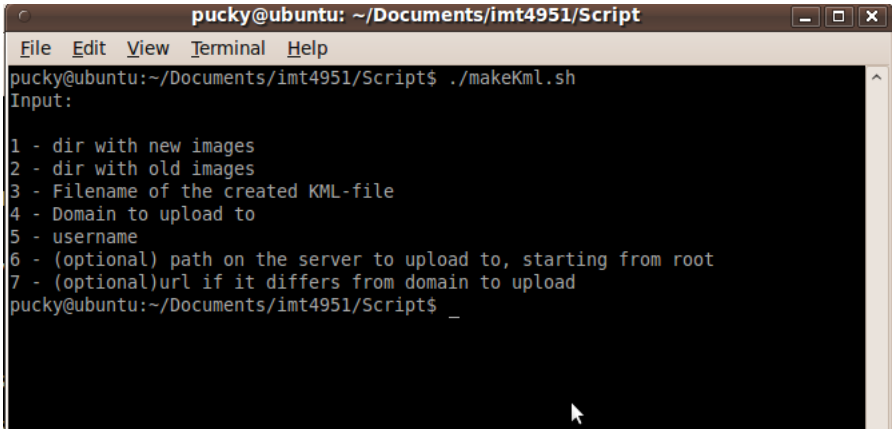
wouldn't work. Finally, comparing filenames came out on top as the best compromise between less initial work, and the scripts accuracy. This means that the user has to rename the old and the new photographs so they have the same name, but are located in different folders.

Using KMZ-files were considered, but because of the decision to not use Imagemagick, the usage of kmz-files would be useless too. Making a kmz-file with all the non-thumbnails would generate a pretty large file, and Google Maps will not read a kmz-file of that size.

The ability for users to specify if they wanted the script to upload the data or not was planned to be supported a long time, but it was decided that that certain functionality wouldn't add any value to the project. KML-files need to be on the internet to work correctly, and making kml-files with local paths would then be pointless, as they would have to be moved eventually.

It was also desirable to only use input arguments, such that the user can write one line, and let the computer/script do the rest. By using "read", the user would be forced to sit and watch the script.

The script takes a number of input arguments. The first two are location (relative from current dir) of the folder containing new and old photos respectively. It then loops through the new images, and processes the corresponding old. This is done in case the number of old and new images differ from each other. All relevant metadata comes from the new images, so it wouldn't make sense to make the kml-file based on the old images. The script copies geodata from the new photos onto the old, so they can be used independently for other purposes, if it is needed. Then the KML-file is generated. If the script is run without input arguments, a reference list with the needed input is shown.



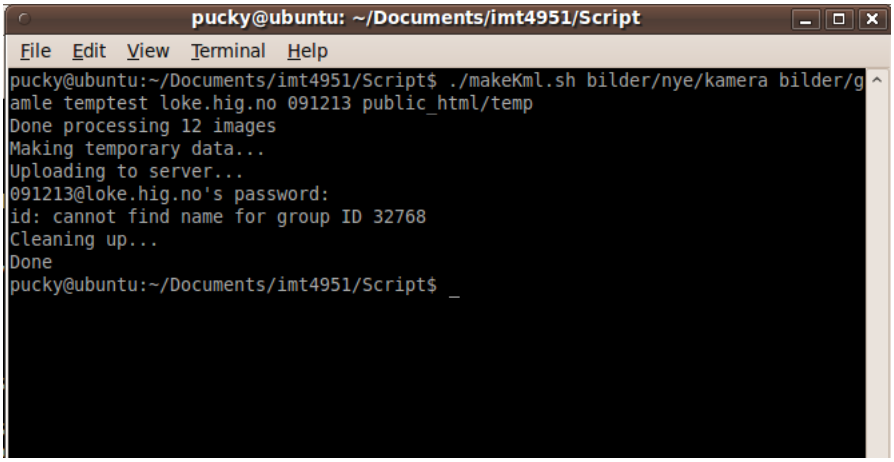
The remaining input arguments deal with uploading to the web. The domain to upload the data to, the username for the domain. The last two are optional. The first should be used when the public url differs from the server-url, and the second when a different location than the root is desirable. The biggest

problem with the script becomes apparent here. The aim was to make a script one could start, and come back to later when it was finished. With the addition of uploading data to the web, this is not as easily obtainable. The scp-function prompts the user for a password in the middle of the script. There is no portable way to get around this. The closest would be to set up RSA, but then the script will be less portable.

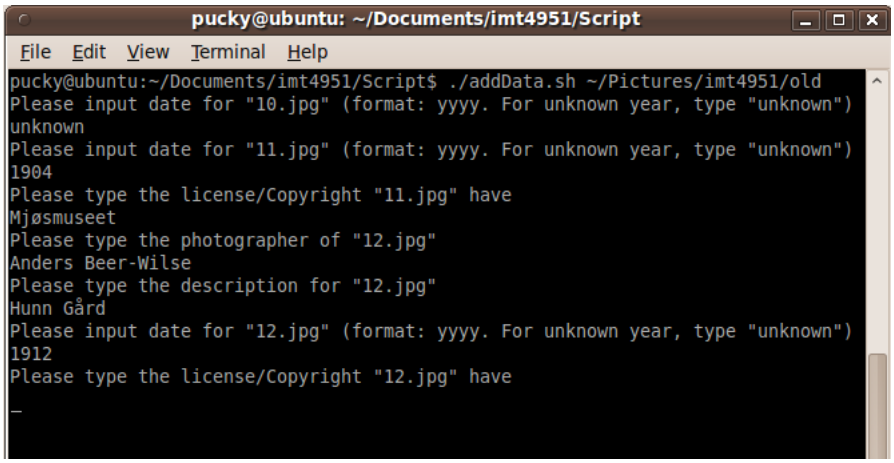
The script has a simple append-functionality, so if a KML-file with the same name already exists, it will append the new entries to it, else it will make a blank file. This was added to make it easier and faster to keep it updated. One needs only to process a pair of images once, and additional entries, made at a later date, will be appended to the same KML-file. The names on the new images cannot be the same as the names on the already existing images. Furthermore, if images are appended to a document, the original entries will lose its formatting.

When uploading to a server, absolute paths were needed. Google maps does not work properly with relative paths in KML-files. This would not be a problem if every server had the same url for http-requests and ssh-requests. Unfortunately, not every server does, so the solution was to accept an additional input argument for those cases where the http-url differs. A related problem was figuring out a way to upload the images and the KML-file to the server in one go. If more than one scp-procedure was used, the user would have been prompted for his password multiple times, which is unacceptable. Several attempts to work around this were made. When more files or folders were specified, the resulting folder structure on the server, would be unpredictable, and would in some cases produce a lot of superfluous folders. The final script makes a temporary folder where it places all the files in a predictable folder structure, and uploads the contents of that folder to the web server.

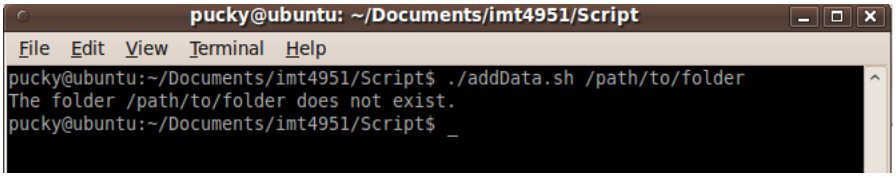
The output from the script looks like this: (Please note that the "id: cannot find ..."-line is produced by the server, and not by the script)



Additionally, a second script was made. This can be used to ease the adding of correct metadata to the images. It checks if the fields, specified by the exif-specification[2], for Author, Date and Time, Copyright and description is set. If they aren't, the user will be prompted to fill them out, with a reference to the current file. This is specifically tailored to updating metadata on old images, so it is only possible to add year as the date, as it is rather rare to know the exact date a very old image was taken. Unknown is also a possible input for year (as shown in the image below). This will set the year to 0000. The main script will recognize this, and write unkown date in the KML-file.



Both scripts also check if the folders exist, and outputs an error if they don't.



[2]http://www.exif.org/Exif2-2.PDF

Chapter 4

Conclusion

4.1 The finished script

The final product contains two scripts. The first was made to make it easier to add metadata to images. This is not completely automatic, but can save the user a lot of time, compared to manually checking if the metadata exists, and add it if not. The second, and most important script is the one which pair images, and make the kml-file. Both of the scripts are only dependent on exiftool, and are therefore highly portable to any machine running a UNIX-based Command Line Interface. Some Linux-distributions are even packaged with Exiftool. This script assumes that the new pictures have correct geodata embedded in them, and that there is one matching old image for each new one. The input arguments needed are:

1 Location of folder with new images

2 Location of folder with old images

3 Name of the generated kml-file

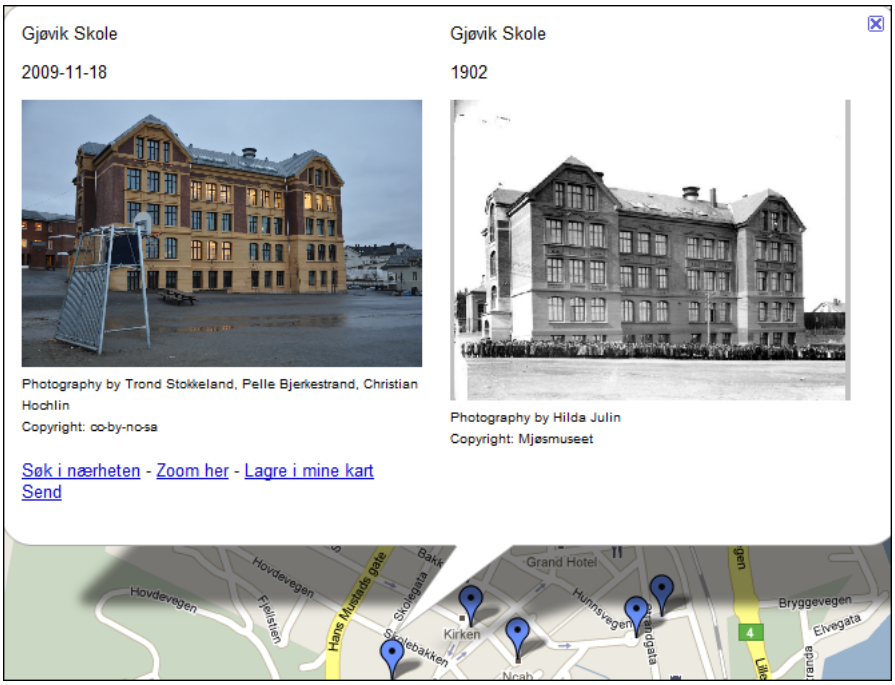
4 Server to upload data to

5 Your username on that server

6 (optional) The url to the server, if it differs from the uploading-address

7 (optional) The folder to upload to on the server

The KML-file produced have marker balloons that look like this:



There are a couple of drawbacks to the finished script. Because the KML-file loses its formatting when appended to, the KML will be harder to read for humans, and therefore harder to update manually (e.g. correct small mistakes). The files that are paired need to have the same name *and* extension (this is case sensitive too, unfortunately) as each other.

The actual work was kept surprisingly in line with the initial plan. The capturing of images that was supposed to happen in week 46, had to be pushed back due to cameras being unavailable, but that was taken into account, and did not slow down the progress. A feature mentioned in the pre-project had to be cut though, the ability to use geodata from a separate device. This was not done because of time constraints, but due to lack of test data.

4.2 Possible extensions

This script was created for a very specific cause – The anniversary of Gjøvik city – and thus, a useful modification for this script would be to make it able to work in a more general way. The script could be split into two. One part would be covering the specific functionality for this project, pairing images where one doesn't have any relevant metadata, and transfer metadata to it. The second would be a more general script, that paired images based on the geodata contained in the metadata. Furthermore, a cron job could be set up to watch a folder for new images, and append them to the kml-file automatically.

As mentioned in the pre-project, choosing between geodata extracted from the files themselves, and from an external device would be a natural extension to this project. Unfortunately, there was no access to external gps-data, and to integrate that kind of functionality would require access to some dummy data.

4.3 Automation

Is automation of workflows useful? The answer to that would be a big resounding yes. Assuming the images have well-formed metadata, the script uses a couple of seconds to extract all the metadata, and generate its part of the kml-file from two images. A normal human would maybe use more than a minute for each pair of images. With a hundred images to be paired, the script uses just over 3 minutes, and the human would use more than one and a half hour.

The challenge is to recognize which tasks are suited for automation. Some tasks would maybe require more work up front to prepare the data than was saved on automating it in the first place. Purely mechanical tasks like this, on the other hand, are well suited for a script, and works well with little work up front for a user.