Media Addressing Through the Spatial Domain
KML & KMZ Generator

Pelle Bjerkestrand

Abstract

This report documents the product and process outlined in the pre-project report[1]. It will not include the full pre-project report[1], but I recommend reading it first as I will quote and reference it where appropriate.

The implementation chapter (1) will focus on the code and show examples of the scripts themselves, while the documentation chapter (2) will contain a non-technical overview of the process and a user guide.

Alongside this report there should be a set of scripts, a KMZ[2] file and several images. These files are also available at the project site[3].

Chapter 1

Implementation

1.1 Introduction

A goal for this project was to use nothing more than standard Bourne-Again shell[5] scripting. Handling KML[6] can be done without dependencies, but using metadata and processing the images themselves requires dependencies. This will be outlined further under 1.3 and 1.4.

The final implementation exists as a set of Bourne-Again shell[5] scripts:

- append.sh
- endkml.sh
- engage.sh
- gpsfixfile.sh
- gpsfixfolder.sh
- makeimages.sh
- makekml.sh
- sortbyloc.sh
- startkml.sh

These scripts will not be quoted in their entirety, as they are published in full on the project website[3].

1.2 Collecting

Finding old images of Gjøvik was done by searching for "gjøvik" at the The National Library of Norway's photo archive[10] and browsing Mjøsmuseet's photo collection[11]. The images downloaded from these sources had metadata that can at best be categorized as sparse.

Tag	**Value**
File Size	45 kB
File Modification Date/Time	2009:11:27 14:52:42+01:00
File Type	JPEG
MIME Type	image/jpeg
JFIF Version	1.01
Resolution Unit	None
X Resolution	1
Y Resolution	1
Image Width	633
Image Height	480
Encoding Process	Baseline DCT, Huffman coding
Bits Per Sample	8
Color Components	3
Y Cb Cr Sub Sampling	YCbCr4:2:0 (2 2)
Image Size	633x480

As we see from the metadata, it is composed only of technical data relating to the scanned file. At the very least, for an image to be usable in the context of this project, metadata about the location captured in the image is needed.

Capturing new images was done using a Nikon D3000[12] equipped with a GP-1[13] GPS receiver. The GP-1 did unfortunately not work with the D3000, so I resorted to taking pictures with my iPhone[14] and copying the location metadata from those pictures over to the ones taken with the D3000 as well as the old ones acquired from The National Library of Norway's photo archive[10] and Mjøsmuseet's photo collection[11]. How this was handled is documented in 1.3.

Figure 1.1: Trying to figure out why the GP-1 wasn't working

Figure 1.2: Collecting images

1.3 Describing

As Exif metadata needed to be read and written, ExifTool[7] was chosen as a way to do this. As described in 1.2, both the old and new images needed GPS data added. Scripting this seemed to be a good solution since there were many images. Two scripts were written for this purpose:

gpsfixfile.sh

```
exiftool \
-overwrite_original \
-exif:gpslatitude="$(exiftool -gpslatitude -T -n "$1")" \
-exif:gpslongitude="$(exiftool -gpslongitude -T -n "$1")" \
$2
```

gpsfixfolder.sh

```
ORIGINAL_PATH=$(pwd)
cd "$(dirname $0)"
SCRIPT_PATH=$(pwd)
cd "$ORIGINAL_PATH"
cd "$1"
FROM_FOLDER=$(pwd)
cd "$ORIGINAL_PATH"
cd "$2"
TO_FOLDER=$(pwd)

IFS=$'\n'

for image in $(ls -p -1 "$FROM_FOLDER" |grep -v "/")
do
    . "$SCRIPT_PATH"/gpsfixfile.sh \
    "$FROM_FOLDER/$image" \
    "$TO_FOLDER/$image"
done

IFS="$OLD_IFS"
```

As is clear from these scripts, they only work if the source and destination files have the same name. Working with larger data sets, another solution would have been developed, but with the limited number of images I was dealing with, renaming them wasn't a problem. As mentioned in the pre-project report[1] the project notes call for "a group of well described files", so these scripts are included as documentation more than as a part of the product proper.

Filling the *artist*, *datetimeoriginal* and *copyright* tags was done manually with ExifTool[7] since the values for these were not the same for any two images.

1.4 Transforming

1.4.1 Image Processing

Further into the implementation phase, I also wanted the creation of thumbnails and web versions of the original images to be a part of the automatic workflow. This would require another dependency. The existing scripts were not modified so that they could still be used if one already had a set of suitable images. Instead, a new script that generated thumbnails and web versions of images using ImageMagick[8] was created.

From makeimages.sh

```
OLD_IFS="$IFS"
IFS=$'\n'

for image in $(ls -p -1 "$IMAGE_PATH" |grep -v "/")
do
    $(convert "$IMAGE_PATH/$image" \
    -format jpg -quality 75 -resize 290x240\> \
    -gravity center -crop 300x250+0+0\! \
    -background black -flatten "$PARENT_PATH/thumbs/$image")
    wait
done

IFS="$OLD_IFS"

echo -e "\t- web versions"
cd "$IMAGE_PATH"
$(mogrify -format jpg -quality 90 \
    -path "$PARENT_PATH/web" -resize 1280x1280\> *.*)
wait
cd "$SCRIPT_PATH"
```

This script posed some challenges. At first `mogrify` with `-thumbnail` was used to generate thumbnails, but this had to be abandoned due to the fact that `-thumbnail` strips all metadata in order to make the file size smaller. In order to get the look I wanted (fixed size black backdrop), a move from using `mogrify` to `convert` was needed since `mogrify` does not support the full range of options the script would be using.

In addition to the thumbnails I also decided to generate images suitable for web use. These would need to be relatively small both in file size and resolution to fit comfortably in user's connections and on their screens. For these reasons, JPGs with a size restriction of 1280 in both dimensions were deemed suitable and chosen.

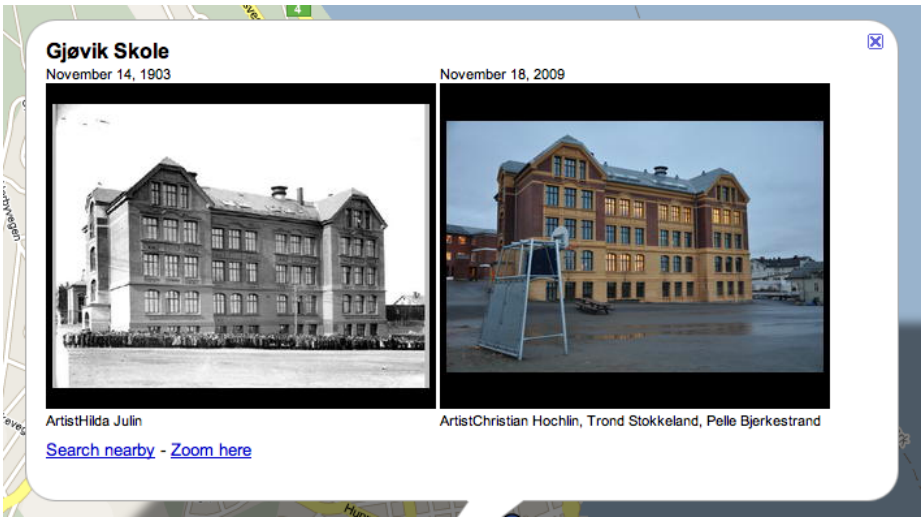

Figure 1.3: Thumbnails with black backdrop

The thumbnails produced end up at 16-61 kilobytes (1.7-6.7 bits per pixel) while the web versions are 200-500 kilobytes (1.5-3.7 bits per pixel). I find this more than acceptable, even for large sets of images, considering that raw image files are usually at 12 or 14 bits per pixel[15].

1.4.2 KML and KMZ Creation

My view on whether to use a KML that references external media or a self contained KMZ is that one should use a self contained KMZ where it is practical. For this project, that means that the KMZ contains the thumbnail images, but not the web versions as that could make the archive quite large. Doing it this way allows the KMZ to be distributed and accessed without the need for dependencies. The thumbnails can even be browsed locally and while offline.

The act of making a KMZ is as simple as making a KML and archiving it and its dependencies in a standard ZIP file with a KMZ extension.

From startkml.sh

```
echo -e "<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<kml xmlns=\"http://www.opengis.net/kml/2.2\"
    xmlns:gx=\"http://www.google.com/kml/ext/2.2\">" >> "$1"
```

From endkml.sh

```
echo "</kml>" >> "$1"
```

What's interesting is what happens between `startkml.sh` and `endkml.sh`.

From makekml.sh

```
echo -e "\t- sorting images by location"
. "$SCRIPT_PATH"/sortbyloc.sh "$IMAGE_PATH" &
wait

echo -e "\t- adding images to \"$1.kml\""
(
    IFS=$'\n'
    for dir in $IMAGE_PATH/*
    do
        if [ -d "$dir" ]
        then
            . "$SCRIPT_PATH"/append.sh \
            "$dir" "$PARENT_PATH/$1".kml "$3" &
            wait
        fi
    wait
    done
)
```

First, `sortbyloc.sh` is run so that all images are sorted into folders based on location. Second, these folders are passed to `append.sh` which appends each image in the folder to the KML file. Being over 100 lines of code, I won't quote `append.sh` here, but what it does is:

- Extract location data, in the form of latitude and longitude, from the folder name
- Figure out the relative path from the KML to the images
- Check for the tags `ImageDescription`, `XPTitle` and `DocumentName` so a title can be made
- Extract other specified tags

It then injects this data as valid HTML and KML into the KML file.

Figure 1.4: HTML output structure

Further, the KML and its dependencies is archived in a standard ZIP file with a KMZ extension.

From makekml.sh

```
cd "$PARENT_PATH"

echo -e "\t- making \"$1.kmz\""
zip -q "$1".kmz "$1".kml
wait

echo -e "\t- adding images to \"$1.kmz\""
zip -q -g -r "$1".kmz "$IMAGE_PATH_R"
wait
```

When the operation is complete there will be a KML file, a KMZ archive, a folder of web images and a folder of thumbnails located next to the specified input folder, ready for distribution and use.

1.5 Presenting

Presenting the generated KMZ[2] can be done by referencing its location in the Google Maps[9] search field. The file has to be reachable over standard HTTP for this to work, but can be located anywhere.

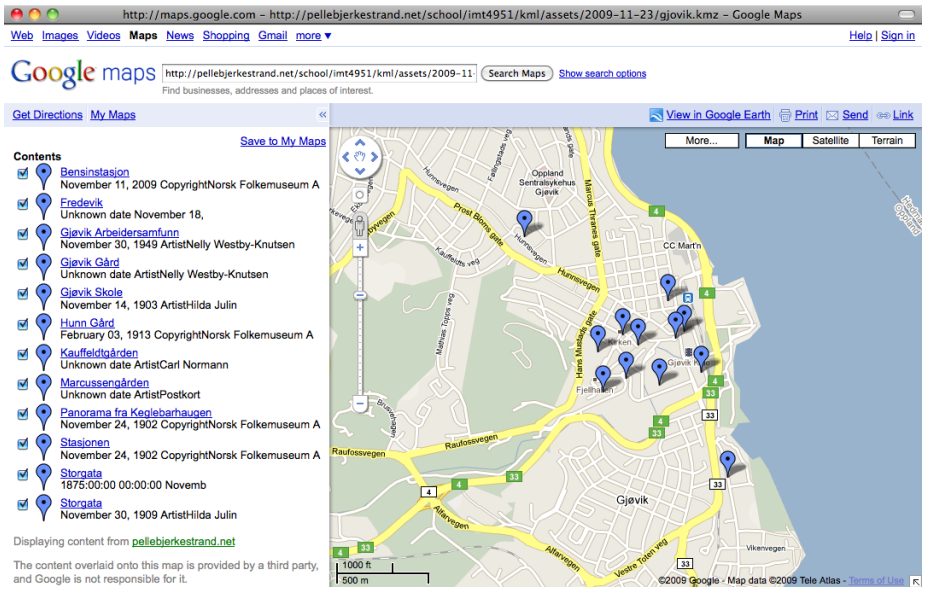

Figure 1.5: Presenting a KMZ[2] in Google Maps[9]

1.6 Automating

Here I will provide an overview of the automated workflow as it exists in the scripts. This will be done in a fashion as to, as clearly as possible, explain which scripts do what jobs.

To facilitate launching the script from any folder and specifying both absolute and relative paths, `engage.sh` includes code that makes, and saves in constants, all absolute paths needed. `makeimages.sh` and `makekml.sh` both include variations of this code so that they can be run independently of `engage.sh` and each other.

From engage.sh, makeimages.sh and makekml.sh

```
ORIGINAL_PATH=$(pwd)
cd "$(dirname $0)"
SCRIPT_PATH=$(pwd)
cd "$ORIGINAL_PATH"
cd "$2"
IMAGE_PATH=$(pwd)
cd ..
PARENT_PATH=$(pwd)
cd "$SCRIPT_PATH"
```

1.6.1 engage.sh and makeimages.sh

The `engage.sh` script is a launcher that starts the workflow. It takes two or three arguments: desired KML and KMZ file name, source image folder and optionally a web URL where the linked to images are located. After this it calls `makeimages.sh`. What `makeimages.sh` does and how it does it is shown in 1.4.1.

1.6.2 sortbyloc.sh

Further, `sortbyloc.sh` is called with the `IMAGE_PATH` as the only argument.

From sortbyloc.sh

```
(
    IFS=$'\n'

    for file in $(ls -p -1 "$1" | grep -v "/")
    do
        location="$(exiftool -gpslongitude -T -n "$1/$file") \
                $(exiftool -gpslatitude -T -n "$1/$file")"

        if [ ! -d "$1/$location" ]
        then
            $(mkdir "$1/$location")
            wait
        fi

        $(mv "$1/$file" "$1/$location/$file")
        wait

    done
)
```

This loops through all files in the folder given as an argument and checks to see if there exists a folder that is named the GPS location of the file being processed. If a folder does not exists, one is created and the file is moved into this folder. If the folder exists, the file is just moved. `append.sh` parses these folder names to determine the location of the images inside. This is done to reduce calls to ExifTool[7].

1.6.3 makekml.sh

After `sortbyloc.sh`, `engage.sh` calls `makekml.sh` with either two or three arguments depending on whether an external URL was specified or not. The name in `engage.sh`'s first argument is passed as the first argument to `makekml.sh`, the newly created thumbs folder's location as the second argument and optionally the external URL as the third. What `makekml.sh` does and how it does it is shown in 1.4.2.

In order to properly reference files in a KMZ, `makekml.sh` needs to know the path of the thumbnails relative to the KML. The solution to this is the following bit of code.

From makekml.sh

```
OLD_IFS="$IFS"
IFS="/"
ARR=( $IMAGE_PATH )
IFS=$OLD_IFS
IMAGE_PATH_R="$ARR[$#ARR[@]-1]"

cd "$PARENT_PATH"
```

This makes an array out of `IMAGE_PATH` with `/` set as the delimiter, making each folder name an entry in the array. It then saves the last entry of the array (the array's length minus one) as the new variable `IMAGE_PATH_R` which can now be used to reference images from the KML inside the KMZ. The script is already running in `PARENT_PATH` before this piece of code executes and keeps running in it afterwards as to be able to correctly add the thumbnail images, using `IMAGE_PATH_R`, when making the KMZ.

From makekml.sh

```
echo -e "\t- adding images to \"$1.kmz\""
zip -q -g -r "$1".kmz "$IMAGE_PATH_R"
wait
```

1.6.4 gpsfixfolder.sh and gpsfixfile.sh

These two scripts were used to facilitate the transfer of GPS coordinates from one image to another and were written as a helpful tool during the implementation process, not as a part of the final product, and are therefore rather simple.

From gpsfixfolder.sh

```
ORIGINAL_PATH=$(pwd)
cd "$(dirname $0)"
SCRIPT_PATH=$(pwd)
cd "$ORIGINAL_PATH"
cd "$1" FROM_FOLDER=$(pwd)
cd "$ORIGINAL_PATH"
cd "$2"
TO_FOLDER=$(pwd)

IFS=$'\n'

for image in $(ls -p -1 "$FROM_FOLDER" | grep -v "/")
do
    . "$SCRIPT_PATH"/gpsfixfile.sh \
    "$FROM_FOLDER/$image" "$TO_FOLDER/$image"
done

IFS="$OLD_IFS"
```

`gpsfixfolder.sh` is really just a generic script that runs through all items in a folder that are not folders themselves. In this case it passes `FROM_FOLDER` and `TO_FOLDER` as arguments to `gpsfixfile.sh`, but this can be substituted by any other procedure.

From gpsfixfile.sh

```
exiftool \
-overwrite_original \
-exif:gpslatitude="$(exiftool -gpslatitude -T -n "$1")" \
-exif:gpslongitude="$(exiftool -gpslongitude -T -n "$1")" \
$2
```

We see here that `gpsfixfile.sh` really is the script that does all the work on the files.

Chapter 2

Documentation

2.1 Planning

Planning a small one man project is not easy, in that there is no-one to answer to except yourself. For this project there was only one external factor: the delivery date. Nevertheless, I have managed to stick to the schedule set forth in the pre-project report[1].

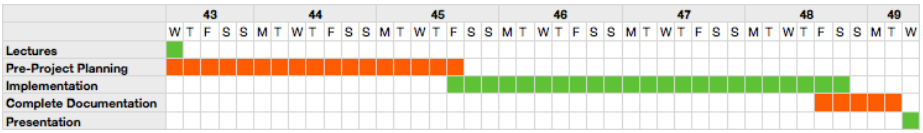

Figure 2.1: Gantt diagram from the pre-project report

Though there have been some anomalies, like spending more than the 30 hours I viewed as "fair" for the project and documenting changes/milestones on the website[3], there now exists a usable implementation of KML and KMZ generation from a set of unordered, but well described, images.

2.1.1 Research Questions

The research questions I chose to pursue were the following:

1. How much of the workflow of creation and publishing KML and dependent files can be automated using only standard UNIX shell scripting?
2. Is fully automating the workflow justifiable with regards to computing time and performance versus doing some steps manually?

Answers to these will be explored in the conclusion (2.4).

2.2 Implementation

As I viewed scripting as the most critical part of the project, writing the scripts to automate the workflow was done before collecting, describing and transforming any of the images related to this project. Images from my private library served as dummy images until I was confident enough to leave scripting alone and go outside to collect images of Gjøvik.

Considering that this project is relatively small and that I would be working on it on my own, I resorted to cowboy code[16] the whole implementation. This allowed me to focus on the task at hand rather than figure out how to adhere to the specifications of a chosen development methodology.

As with most of my solo projects, less time was spent figuring out the technical aspects of the workflow while more time was spent debugging and tweaking.

2.3 End product

The end product consists of a multitude of scripts, but only `engage.sh` needs to be run in order to generate a complete and working KMZ. This script takes three arguments:

1. The name desired for the KML and KMZ
2. The folder of images
3. A valid and reachable URL where the web versions will reside

Argument two, the path to the folder of images, can be either relative or absolute. Argument three, the URL, is optional. If left out, the thumbnails included in the KMZ will not be clickable links.

The script will produce appropriate output while it is running.

Example output

```
Making images
    - thumbnails
    - web versions
Done
Making KML & KMZ
    - creating gjovik.kml
    - sorting images by location
    - adding images to "gjovik.kml"
    - adding "Hunn Gård"
    - adding "Gjøvik Skole"
    - adding "Storgata"
    - adding "Gjøvik Arbeidersamfunn"
    - adding "Panorama fra Keglebarhaugen"
    - adding "Storgata"
    - adding "Gjøvik Gård"
    - adding "Marcussengården"
    - adding "Kauffeldtgården"
    - adding "Bensinstasjon"
    - adding "Stasjonen"
    - adding "Fredevik"
    - closing "gjovik.kml"
    - making "gjovik.kmz"
    - adding images to "gjovik.kmz"
Done
```

2.3.1 Only generating images

If it is of interest to only generate thumbnails and web versions of a set of images, this can be done using `makimages.sh`. This script only takes on argument: a path to a folder of images. It will the generate and fill the folders "thumbs" and "web" next to the original image folder.

2.3.2 Only generating KML and KMZ

Like `makeimages.sh`, `makekml.sh` can also be run on its own taking the same three arguments as `engage.sh` (name, image folder, external URL). Note that the images located in the specified image folder are the images that will be the ones referenced in the KML and included in the KMZ, and will not be processed in any way.

2.4 Conclusion

2.4.1 Process

Although I am used to, and prefer, working as part of a team, I feel that working alone did not hinder the process on this particular project. I attribute this so cowboy coding[16] the implementation and then moving my focus over to fixing bugs and making it more user friendly.

Following a plan outlining periods with planning, implementation and then documentation was a challenge, as I am used to spending the vast majority of time implementing things using an agile development methodology relying on user stories and multiple smaller deliveries. Looking back on the informal progress log I kept for myself[17], I see that I can loosely divide the implementation process into what felt like iterations:

- KML and KMZ creation for single images
- KML and KMZ creation for multiple images
- Collecting and describing images
- Image processing
- Combining image processing and KML and KMZ creation

Going about the implementation in a familiar manner like this helped me keep focused and kept me from growing tired. I also view the development of the project website[3] as part of the final product, but coding it was something I used as breaks between coding the actual scripts.

2.4.2 Research Questions

How much of the workflow of creation and publishing KML and dependent files can be automated using only standard UNIX shell scripting?

The creation of the KML itself is done with standard BASH[5] using `echo` and redirecting its output to a file (>>). The other facets of KML and KMZ production rely on external programs such as ExifTool[7] and ImageMagick[8].

A complete workflow can therefore not be automated using only standard UNIX shell scripting.

A complete workflow can, however, be automated using only standard UNIX shell scripting if one allows for external dependencies.

Is fully automating the workflow justifiable with regards to computing time and performance versus doing some steps manually?

The full workflow using all 24 test images takes roughly 50 seconds to complete. That's about two seconds per image. Manually opening, resizing to web version metrics and saving one image using a popular image manipulation program[20] took me 35 seconds. This translates into 14 minutes for all 24 images and this is only resizing to web version metrics.

Fully automating the workflow, as can be observed by using `engage.sh`, is therefore justifiable to a large degree.

2.4.3 Product

The product itself differs from what was, in the pre-project report[1], put forth as an example.

- KML generator with a cron job
  - Check drop box/input folder for new images
  - Move new images to another folder
  - Manipulate images so that they are suitable for web use
  - Append data to KML file
    - HTML templates with search and replace

We see that watching a folder where images would be dropped and then processing and appending them to an existing KML file was presented. Actually implementing this would require several other external dependencies:

- A command line XML parser[19], as standard BASH[5] does not provide tools for quickly and easily traversing and modifying XML
- Cron[18], or similar, to enable running a script that watches a folder
- A method for quick and reliable search and replace

With the implementation the way it is now, it makes static files and does not update them, but also only has two dependencies (ExifTool[7] and ImageMagick[8]) and no need for configuration. I view this simplicity of use and portability as a worthy tradeoff for expanded functionality. Actually implementing the full functionality of the example would have been considered if I had felt that the final product would be user friendly enough, and codable in the short implementation time given.

2.4.4 Suggested future work

As I would like to see this script mature to include the features put forth in the example above, actually implementing these features is a logical suggestion for future work. Using this script as a base to start work on addressing of temporal media through the spatial domain, as described in the pre-project report[1], is also a possibility.

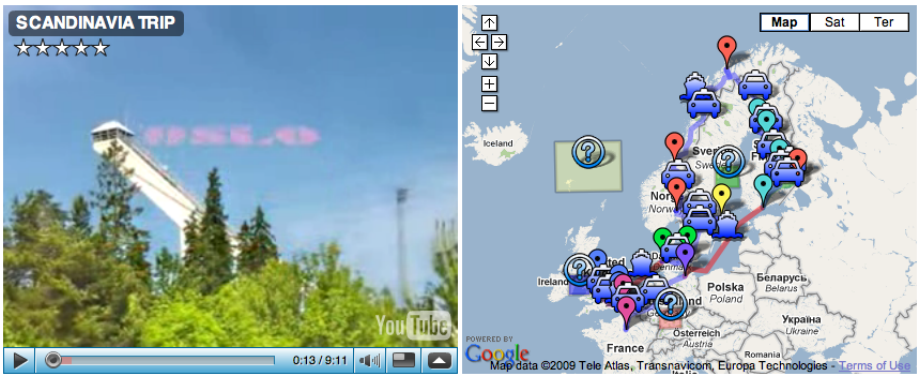

Figure 2.2: Addressing temporal media. Example from the pre-project report

Notes

[1] http://pellebjerkestrand.net/school/imt4951/kml/assets/pre-project_report_bjerkestrand_091217.pdf
[2] http://code.google.com/apis/kml/documentation/kmzarchives.html
[3] http://pellebjerkestrand.net/school/imt4951/kml/
[4] http://creativecommons.org/licenses/by-nc-sa/3.0/
[5] http://www.gnu.org/software/bash/
[6] http://code.google.com/apis/kml/
[7] http://www.sno.phy.queensu.ca/~phil/exiftool/
[8] http://www.imagemagick.org/
[9] http://maps.google.com/
[10] http://www.nb.no/gallerinor/e_sok.php
[11] http://www.gjovikmuseet.no/index.php?action=fotos&param=display
[12] http://imaging.nikon.com/products/imaging/lineup/digitalcamera/slr/d3000/
[13] http://www.nikonusa.com/Find-Your-Nikon/Product/Miscellaneous/25396/GP-1-GPS-Unit.html
[14] http://www.apple.com/iphone/
[15] http://en.wikipedia.org/wiki/Raw_image_format
[16] http://en.wikipedia.org/wiki/Cowboy_coding
[17] http://pellebjerkestrand.net/school/imt4951/kml/progress.html
[18] http://en.wikipedia.org/wiki/Cron
[19] http://xmlstar.sourceforge.net/
[20] http://www.pixelmator.com/