INF573 Project: Christmas Decorator

Christophe Haddad and Marc Gantenbein

École Polytechnique, Palaiseau, France

Abstract. We present a Christmas Decorator for house facades implemented in C++ with OpenCV featuring feature extraction, day to night transforms, lights including glow, homography transforms and mistletoe decorations.

1 Introduction

The current Covid-19 pandemic prevents many of us to buy Christmas decorations, deemed non-essential, and bring our house into a festive mood. We thus decided to create a virtual ChristmasDecorator, an OpenCV pipeline that transforms an image of a house into a festive nighttime image including decorations of lights and mistletoes.

Our approach consists of finding important features of a house with an image translation cycleGAN[8], a homography transform to align the house to the image plane, day to night transfer by darkening the sky and the image, and finally adding decorations, including glow to our house.

While other approaches might have been possible as well, we decided to opt for a more classical approach of day to night transfer and decorations by employing the concepts learned in INF573.

2 Related works

We did not encounter any other works that took up the task of decorating facades with Christmas lights, thus we will introduce the underlying techniques and OpenCV functions used throughout our project in the following section.

2.1 cycleGAN and pix2pix image translation

Our first task was to find a suitable dataset for decorating and possibly some aid for the day to night transfer. We found publicly available trained Generative Adversarial Networks (GANs)[1] which were even equipped for translating images of facades to labels in the form of windows, doors, roof, columns etc.

GANs have been introduced by Goodfellow et al. [1] as a generative model approach where two *adversarial* networks are trained at the same time. There have been many variations of the GAN approach [3], [8] and they are currently,

[1] `https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix`

besides Variational Auto-Encoders, one of the most popular choices for a generative modeling approach.

GANs are based on a two fold training procedure. The first network, the *generative*, generates an image from the training data while the second network, the *discriminative* tries to guess whether the image generated by the first one comes from the training data or not. The first network is trained to maximize the probability that the second model makes a mistake and the second model tries to find out if it was fooled.

The goal of the architecture we used, cycleGAN[8], is to perform image-to-image translation. This could for example be bringing the image of a horse to the image of a zebra. While this sounds like a suitable task for a normal GAN, the issue is that paired training data is usually not available in sufficient quantity. The goal of cycleGAN is thus to find a mapping $G : X \leftarrow Y$ with adversarial training. Since there are no constraints, the adversarial loss is coupled with a so called consistency loss [8] such that an inverse mapping $F : Y \leftarrow X$ yields $F(G(X)) \approx X$.

For the process of translating facade images to labels, we employed a pre-trained cycleGAN architecture in the reverse order. The task of the trained cycleGAN is to generate a facade from a given label. However, during inference we used it in the other direction, thus generating labels from a given facade.

2.2 Image quantization

Quantization is usually known as bringing values from a large or continuous set to a smaller set, such as measuring a continuous signal and then rounding the measurements to a set of values.

Image quantization is the process of rounding colors of an image to a predefined set of values. This process already takes place when taking a photograph, where the continuous color spectrum is quantized to the RGB space. So taking a photo is actually quantizing to $(2^8)^3$ colors.

This space is still much to large for our purposes so we need to do requantization to less colors. Quantizing an image thus reduces the complexity of an image and can facilitate feature extraction since there are less colors.

Uniform quantization Uniform quantization is the process of choosing for example $K = 8$ colors and distributing these colors uniformly over the color space. We divide the color space into 8 equal parts and calculate the distance of the color of each pixel to these colors. To each pixel we then assign the color of the closest centroid.

While this approach is quite simple to implement, it fails on images where a lot of colors are very similar as is the case for facades. In this case, many pixels with similar colors are assigned to the same color and we lose a lot of features.

Image quantization with k-means An approach to alleviate this problem is the k-means clustering algorithm [5]. k-means can be used to find a clustering

on data in an euclidean space. The solution minimizes the squared euclidean distance between the points $\mathbf{x_i}$ and the assigned centroids $\mu_\mathbf{i}$.

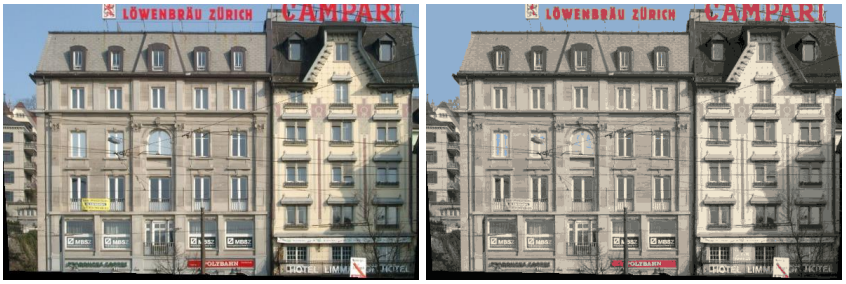
The goal is to find

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} ||\mathbf{x_i} - \mu_\mathbf{i}||^2 \tag{1}$$

While the problem is NP hard, a solution can be approximated with Lloyd's algorithm [4]. The solution consists of the sets $S_i \in S$ that contains the assignments to the clusters and the corresponding centroids $\mu_i \in \mathbb{R}^n$.

For image quantization of a colored image, we take each pixel as a vector thus $\mathbf{x} \in \mathbb{R}^{\text{height} \times \text{width} \times 3}$ and $\mathbf{x}_i \in \mathbb{R}$.

We then choose a number of clusters k and run Lloyds algorithm to obtain the assignments S and the centroids $\mu \in \mathbb{R}^{k \times 3}$. To every $\mathbf{x_i}$ we then assign the color of the corresponding centroid to the pixel. This then yields an image quantized to k colors. An example can be found in Fig. 1



(a) Original image (b) Image quantized to 12 cluster

Fig. 1: Quantizing the input image

Although we used the RGB colorspace, the LAB color space might be a preferable choice for image quantization where the colors are in a *perceptually uniform* space such that the clustering and centroid assignment matches our perception. The LAB space is divided into the axis L for lightness and a and b for the green-red and yellow-blue axis. However, since we did not have special requirements for our quantization, we used the RGB colorspace.

OpenCV provides an implementation of k-means where an image has to be fed pixelwise to obtain labellings and centroids.

2.3 Homography and projective transformations

In the context of projective transformations, a plane from a certain perspective of an image is obtained from warping another plane perspective. The coordinates of the planar points are related by a matrix H, the homography matrix or planar

perspective map matrix. Using normalization [6], the relation between the source points (x, y) and the destination points (x', y') is given by:

$$\begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \approx \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

With normalization, we have 8 degrees of freedom. To solve for the matrix H, we use four pairs of points (source and destination) and then find the h_{ij}'s with Least Squares Regression [6].

We implemented the projective transformation to align distorted facades using a technique for selecting and computing source/destination points detailed later in the *Facade alignment using homography* section. In Fig.2 , we showcase our method aligning facades using homography.

(a) Input facade (distorted)

(b) Selected source points (green)

(c) Aligned facade

Fig. 2: Facade alignment using homography

2.4 Gradients and filtering

We used gradient-inspired filters to blur, sharpen, and detect edges in an image. Each of these filters is applied to the image by a convolution operation.

To blur an image, the Gaussian operator is used. The Gaussian kernel is computed using:

$$G_\sigma = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

Increasing the value of σ increases the spread of the blur. The corresponding OpenCV function we used is called `GaussianBlur`.

The Laplacian operator is useful for sharpening images. It computes the second order derivatives of the image pixel values I:

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

It can be approximated by a small kernel, the two common ones shown in Table 1.

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

Table 1: Laplacian kernel approximations

The Sobel operator approximates the derivative of a Gaussian kernel [6]. It is mainly used for detecting edges, as it emphasizes regions of high spatial frequency (approximating the gradient magnitude at each point). Its kernels for both the x-direction and the y-direction are given in Table 2.

-1	0	1
-2	0	2
-1	0	1

(a) s_x

1	2	1
0	0	0
-1	-2	-1

(b) s_y

Table 2: Laplacian kernel approximations

The `blur()` function of OpenCV convolves the image with a kernel of all ones normalized to the size. The filter is also known as *normalized box filter*.

For denoising the smoothed pictures, e.g. as done for the preprocessing of the masks we employed the function `fastNlMeansDenoising()` that leverages non-local means denoising[**?**], where each pixel is replaced by the average of the pixels similar and close to it.

2.5 Intensity transform

To do an intensity transform we used the `gammaCorrection()` function that changes the luminance of the image according to a power law expression that matches the perception of luminance of the human eye. Thus by using Gamma transforms we can create realistic changes of the luminance of an image.

2.6 Color transforms in images

RGB colorspace By only scaling the red, blue or green channel we can create a color shift. By e.g. scaling the blue channel we can thus create a blue shift of the image.

HSV colorspace The HSV colorspace is composed of *hue*, *saturation* and *value*. By changing value, we can change the perceived brightness of the image and saturation can be decreased to create nighttime effects.

2.7 Harris corner detection

Corners are regions in the image where there is a large variation in pixel intensity in all directions. The Harris corner detector is a popular approach for finding corners[2]. It consists of a moving window around an image point through which the difference in intensity is computed. It is expressed as:

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} [\underbrace{I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

We must maximize *E(u,v)* to find corners. By rearranging the expression (with a Taylor Expansion and some steps), we get:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

where

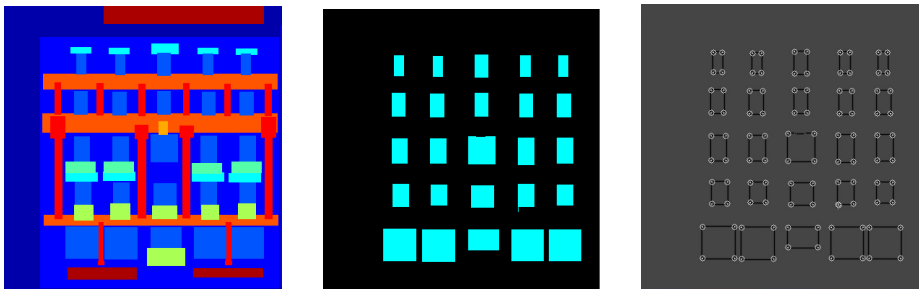
$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Here, I_x and I_y are from the gradient of the image (obtained with the Sobel operator for instance). Then, a score R is computed to decide if the window contains a corner or not, with:

$$R = det(M) - k(trace(M))^2$$

- When $|R|$ is small (chosen threshold), the region is flat.
- When $R < 0$, the region is an edge.
- When R is large (chosen threshold), the region is a corner.

We used the Harris corner detector to find the corners of the various labels generated by the cycleGAN, since the corners coordinates (of each label) are not provided. This allowed us to later identify the relevant regions of each label (with their corners coordinates) and add decorations. In Fig. 3, we showcase the use of the Harris corner detector for finding the windows corners.



(a) Input labels (b) Windows labels (c) Detected corners

Fig. 3: Harris corner detection on windows labels

3 Methods

After giving an overview of the techniques to be employed, we now give an overview of our decoration pipeline.

Our ChristmasDecorator pipeline can be roughly divided into the following steps:

1. Align facade to camera plane with a homography transform
2. Extract labels from a facade
3. Do day to night transfer
4. Decorate the image
5. Realign image to original plane

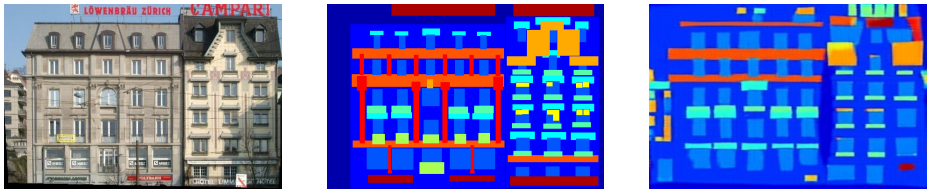
The first challenge is, where to put decorations. To create the impression of realistic decorations, we need to choose features of a house to decorate. We can for example decorate all boundaries of windows, rooftops only, signs on a facade, doors etc. This would lead us to a rather challenging segmentation task, which does not have an obvious solution.

3.1 Extraction of labels from a facade

The first task is thus to extract features from an image of choice.

CMP_Facade_Database We chose to work with the CMP_Facade_Database which includes 606 images from various sources [7]. Tylecek et. al [7] also provides a segmentation algorithm, however, with the fast advent of neural network segmentation architectures, we decided to employ a newer architecture which promised to yield better segmentation results.

Image segmentation of facades For translating an image of a house, we employed a cycleGAN [8], which provides pretrained architectures for various datasets. Although the original purpose of the trained cycleGAN network facades2label is to make image translation from labels to a facade, we used the cycleGAN to segment a facade into labels.



(a) Facade (b) Ground truth labels (c) Segmentation result

Fig. 4: Segmentation result of pretrained cycleGAN

Looking at the CMP_Facade_Database in Fig. 4, we can see that the ground truth labels provide exact boundaries for all windows. The output of the trained cycleGAN, however, provides us with a blurred version of the labels. Although for the eye the difference between the two images is not obvious, extracting mask from labels with continuous colors requires some post-processing. This happens due to the fact, that not all labels are present in an image and sometimes multiple labels are assigned to the same feature as seen for the orange label in Fig. 4c. Thus before obtaining a single label, the images needs to be quantized.

Quantizing segmentation result with k-means To obtain a defined number of different colors, we decided to quantize our image to a predefined number of clusters. The choice of the numbers of clusters defines how many labels can be obtained from the segmentation image.

Obtaining individual labels as mask The next step is to decide, which labels should be taken into account for decorating. The choice of labels to decorate was done by hand, e.g. we generated decorations for all labels.

Post-processing of the individual masks The extracted masks might port artefacts. We thus blured and deblurred the mask to remove high frequency noise.

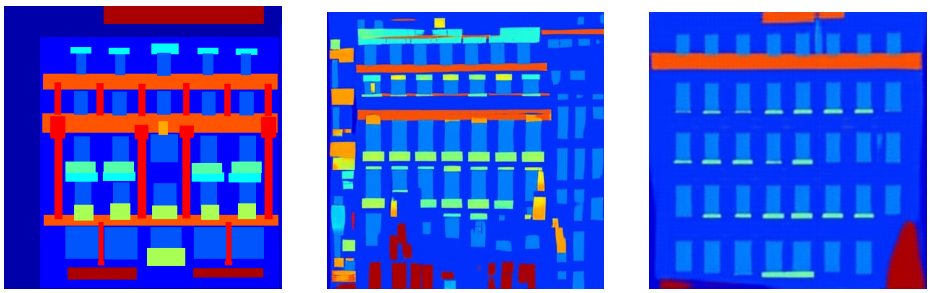
3.2 Adding decorations to day images

Before doing the day to night transfer, we decorated the image with some mistletoe, so that they blend well in the final night image. Specifically, we decorated

the windows of a facade with mistletoe (hanging in from the top of window, or from a balcony). We identified the window regions from their corresponding labels, obtained from the previous quantizing step.

Selecting window labels First, we extracted the window labels using our `selectColor` function, which filters the labels image by keeping only the color of the window labels (light blue shade). It creates a binary mask (the pixels are set when the corresponding pixels match the window label color) and multiplies it with the labels image. Fig.3b shows a sample output of this step.

Initially, we provided a single color to select, based on the window colors of the ground truth labels. After several tests, we noticed the output of the GAN generated slightly different colors for labels (Fig. 5). We accounted for this variation by using lower and upper bounds for the selected color of window labels; we got those values from the minimum and maximum values of our generated labels dataset.



(a) Ground truth labels (b) Facade (c) Segmentation result

Fig. 5: Variation of label colors

Computing the windows regions We needed the exact window regions and their corner coordinates to correctly decorate them with mistletoes. Unfortunately, the output of the GAN was only a color-encoded image of labels, which didn't provide the exact coordinates of the window regions. Therefore, after selecting the window labels, we had to manually extract those regions.

The first step was to use the Harris corner detector to get the corners of the window regions. Then, we implemented an algorithm to search for rectangle regions in the output of Harris. The function `findRectRegions` is detailed below:

1. Sort the corners along the rows and columns in $pByY$ and $pByX$ resp.
2. Loop through $pByY$; if the current corner is not already inside a window region (any of the prev. computed ones until now) select it as $curCorner$
3. Find the first next corner along $pByY$ such that it has the same x as $curCorner$ (meaning they are vertically aligned) this is then the $leftCorner$

4. Find $curCorner$ in $pByX$ (efficiently using binary search), and similarly search for $rightCorner$ (having the same y as $curCorner$)
5. If both $leftCorner$ and $rightCorner$ are found, add a window region to the current ones characterized as: a rectangle with corner $curCorner$, width $(rightCorner.x - curCorner.x)$ and height $(leftCorner.y - curCorner.y)$

Improvements to the window region search The algorithm assumes that window corners are exactly aligned, and that there are no 'junk' corners generated by Harris. Obviously, this initial approach was not sufficient for correctly finding windows regions for diverse inputs. We added the following improvements to this component:

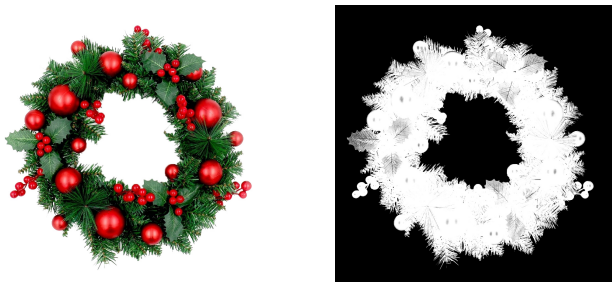
- We allowed the search for vertical and horizontal corners to account for slight variations. The $cornersMarginThresh$ hyper-parameter dictates the horizontal/vertical margin allowed (expressed as a ratio of the image dimensions) to look around when searching for a corner. Surely, we updated the computation of width and height when the found corner was 'outside' $curCorner$
- The Harris corner detector often generated duplicate corners that were very close to each other. We tackled this issue by filtering the Harris corners: the $minCornerDistance$ hyper-parameter dictates the minimum allowed distance (experessed as a ratio of the image dimensions) between any two corners. So the ones very close to each other were removed
- We enforced a minimum width/height of window regions with the $minDimRatio$ hyper-parameter (also a ratio of the image dimensions). In the search, we don't consider corners that would yield a window height/width smaller than our minimum
- To get uniform window sizes, we looked at the cumulative average of window region dimensions. If there's a significant difference between the current dimensions and the average one (at least twice more), then we skip this candidate window region
- Sometimes, the windows were too small for the mistletoes. We tackled this issue by creating the `scaleRegions` function which scales the windows regions (respecting the boundaries)

Adding mistletoes to windows regions After the previous step, given a list of the windows regions (and their coordinates), we can decorate them with our mistletoes. In Fig. 6, we show the mistletoe image and the custom mask we created for adding it to a window region. We centered the mistletoe in the square region (mid-height and mid-width) and used the `addWeighted` function with an *alpha* of 0.9 to realistically blend the mistletoe in the image.

The complete pipeline for adding the mistletoe is shown below in Fig. 7.

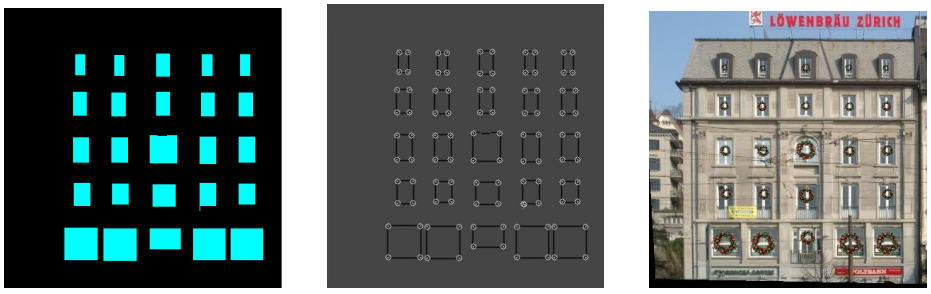
3.3 Day to night transfer of an image

Since Christmas decorations are usually lighted when it is dark outside, the next step is to transform an image taken at day to a night time image. Since we



(a) Mistletoe (b) Mistletoe mask

Fig. 6: Mistletoe template



(a) Selected window labels (b) Harris corners (c) Decorated windows with mistletoes

Fig. 7: Complete day decorations pipeline

already employed cycleGAN for our purposes, we decided to try day to night transfer with a GAN as well.

Day to night transfer with GAN The first approach was employing a pix2pix architecture trained on day to night transfer. This approach failed as can be seen in Fig.8. Although the image looks more like a night image, it can be seen that sky has been added to the top and the image has been heavily distorted. This might be due to the fact that the original network was mostly trained on landscapes and images which also include sky, trees and other features that are not found on the facade.

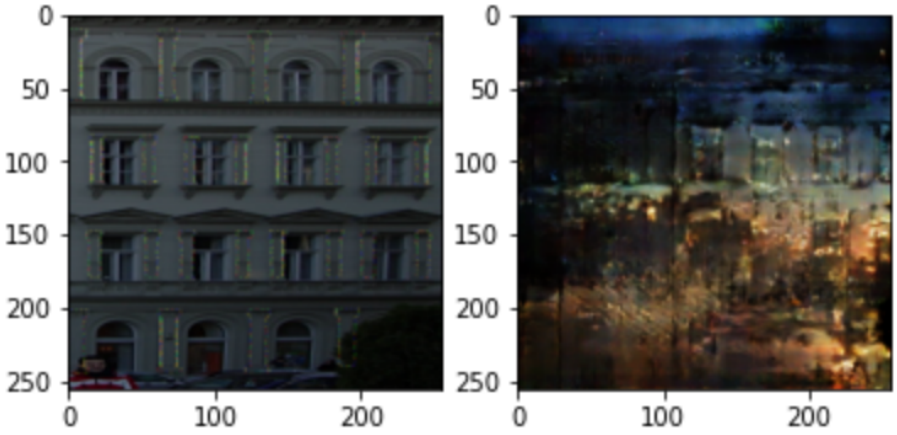


Fig. 8: day2night transfer with pretrained pix2pix

We thus refrained from doing day to night transfer with a already trained architecture, since training a new GAN would require a suitable dataset for day to night transfer on facades and enough time to perform the actual training.

Sky darkening One of the most important steps for creating a night time image, is having a sky that is relatively dark in relation to the facade to be decorated.

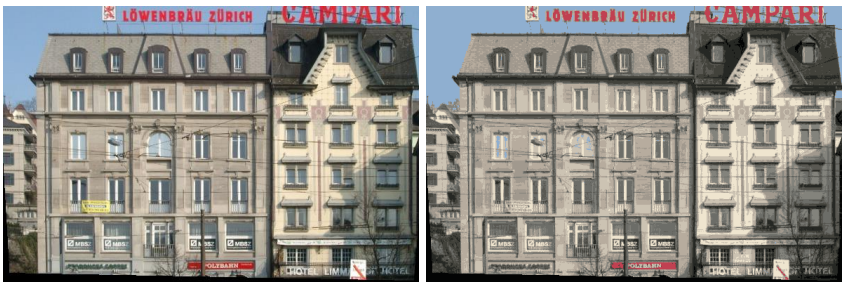
The first step was thus to detect the sky in an image and replace it with a dark color, that creates the impression of being night. Intuition suggests, that the sky fulfills the following properties:

- The sky is in the upper part of the image.
- The sky has the same color.
- There is not much change of colors or other features in the sky.

Note that this properties only applies for a sky with uniform color. We can translate this properties into the following algorithm:

1. Quantize the input image.
2. Calculate absolute value of gradient G.
3. Blur G.
4. Inverse threshold G.
5. Find dominant color in upper part of image.
6. Replace region with dominant color in original image with dark color.

We start by quantizing the image with k-means presented in the previous section. This yields an image with N distinct colors and variations in the color of the sky can thus be omitted when detecting the sky. (Fig. 20b).



(a) Original image (b) Image quantized to 12 cluster

Fig. 9: Quantizing the input image

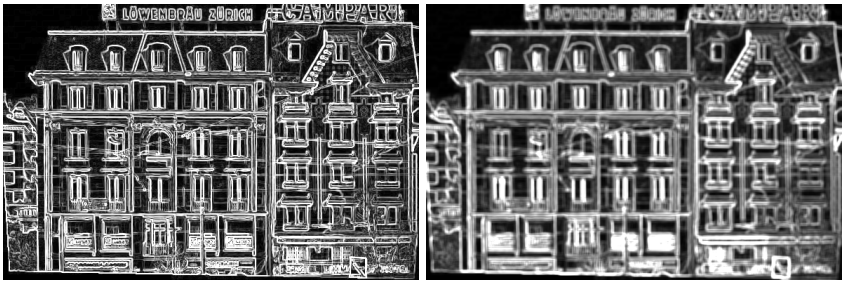
To account for a uniform color of the sky, we calculate the Sobel gradient into x- and y- direction of our original image (Fig. 10a). To filter out possible artifacts or noise in the sky, the sky was blurred with a kernel of size 5 as can be seen in Fig. 10b.

The blurred gradient was then thresholded to only keep areas of the image where the gradient is below the threshold. In the area of the threshold the major color was found by considering each pixel located in the upper part of the image and which is on the mask and choosing the color with the largest amount.

The colored mask was then darkened by multiplying each pixel with a factor of 0.6. The boundaries of the mask were blurred and each pixel was then merged with the original image by the following formula:

$$\text{pixel} = (1 - \text{maskpixel}) * \text{imagepixel} + \text{maskpixel} * \text{skycolor} \tag{2}$$

The darkened sky can thus be seen in Fig. 12b. While the sky looks realistic for a night time image, the building itself is still too bright. We thus employed the transforms described in the following paragraphs.



(a) Absolute value of gradient (b) Gradient blured with filter of size 5

Fig. 10: Gradient and blurring of gradient

(a) Inverse thresholding gradient (b) Mask on quantized image

Fig. 11: Extracting sky as colored mask

Bluelight shift on image As sometimes seen in old movies, creating a night time impression can be achieved by adding a blueshift to the image. Fig.13b

Brightness change We employed a brightness change by transforming the blueshifted image to the HSV color space where we scaled the V channel of each pixel with 0.8. Fig.13c

Saturation change The saturation decrease was achieved by scaling the S channel of each pixel by 0.8. Fig.13d

Gamma correction As a final method we employed Gamma correction with a factor of 2 to the image with all the previously mentioned transforms already applied. Fig.13e

3.4 Adding decorations to night image

For decorating the night images we make use of the masks obtained in the previous section.

(a) normal image (b) Image with darkened sky

Fig. 12: Applying colored mask to original image

(a) (b) (c) (d) (e)

Fig. 13: Transforms used to go from day to night

For the Christmas lights we employ the following steps.

1. Preprocess mask and then obtain boundaries of labels as edges.
2. Position the lights on the edges.
3. Create lights glow.
4. Create windows glow.
5. Crop glow to desired region.

Preprocessing of mask For the next step, we ideally have edges on all the boundaries of our masks. and we do not want any noise in between the edges. If for example there are areas of the mask with very convoluted structures, this might create a random appearance of lights in the next steps.

The mask was thus first blurred with the `blur()` function to remove artifacts of a small scale. However, since we care about having sharp boundaries on our mask, we also employed deblurring with `fastNlMeansDenoisingColored()`.

We then extract the edges of the masks by calculating the `Laplacian()`. To further remove artifacts of the boundaries and make the edges even thinner, the Sobel derivatives in x- and y- direction were calculated and thresholded, that then lead to the final mask.

Positioning the lights on the edges For adding the lights, we create a zero matrix with the same size as the image to be decorated. We then iterate over all pixels of the mask from before. If a pixel is non-zero and no light has been added in a region around our current pixel, we draw a random color from a predefined

(a) (b) (c)

Fig. 14: Preprocessing of mask

set of colors(e.g. red and white) and randomly position a circle in proximity of our pixel. The randomization of the position creates an effect that the lights seem more natural.

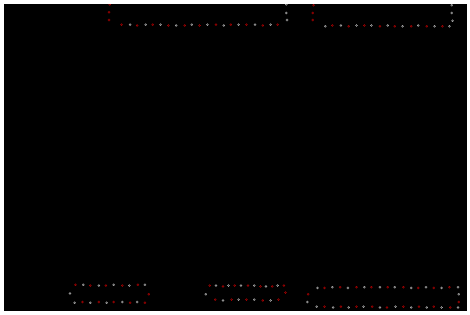
Create lights glow The previous step yields an image where the boundaries are decorated in regular intervals with the colors specified. To create a glowing effect, the `blur()` function was used with kernel sizes of (3,3), (5,5), (6,6), (10,10) and (30,30). The blurred lights where then overlayed with the `max()` function.

Create windows glow The windows glow, if desired for a mask, consists of taking the full mask and replacing it with yellow. This mask is then merged with addWeighted() to the lights.

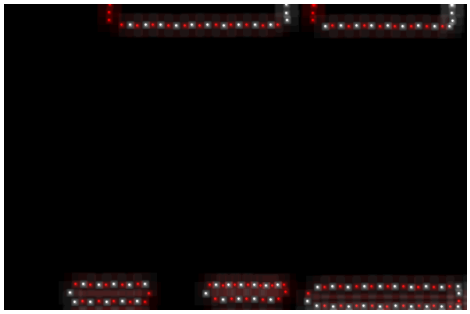
Crop glow to desired regions If we want to create an effect that the lights appear to be on the inside of a window or a similar label, we employ the `copyTo` function with the mask from the windows.

Merge lights and glow with image to be decorated For this purpose, we employed the `addWeighted` function.

Additionally to the lights mentioned above, we also added guirlandes. Adding the guirlandes is similar to adding lights, but instead of processing every boundary we only process horizontal boundaries and instead of a single light a whole string of lights with a random length is added.



(a) Lights without glow



(b) Lights with glows added

(c) Decorated image

Fig. 15: Adding lights and glow

3.5 Facade alignment using homography (optional)

The following is a description of the optional homography component of our pipeline, which aligns distorted facades and allows us to decorate them.

The user selects four corners of the 'distorted' facade (*left mouse button*) and confirms (*right mouse button*). The corners are C1, C2, C3, and C4 with C1 being the top-left corner and the others are selected clockwise. The aligned facade region is estimated as follows:

1. Calculate *leftH* and *rightH*, the left and right heights of the input region resp.
2. $maxH = \max\{leftH, rightH\}$
3. $width = C2.x - C1.x$ (fair approximation, assuming the facade is moderately tilted)
4. $corner = (C1.x, \max\{C1.y, C2.y\})$
5. The aligned facade region is a rectangle with top-left corner *corner*, height *maxH*, and width *width*

We then used the corners of the aligned rectangle region (D1, D2, D3, and D4) as the destination points for the homography transform. We use the OpenCV function `getPerspectiveTransform` to find the matrix H. Then, we warp the distorted image into the aligned image using the `warpPerspective` function.

To save time and allow for reversal of perspective, we implemented a *save* functionality which automatically saves the matrix H into a `.xml` file. This allows us to both quickly re-warp an input image, and recover the original alignment of the image after decorating it.

4 Results and Discussion

In this section, we present some fully decorated images for various facades.

To showcase our fully-automated approach, we did not adapt any thresholds or parameters used. This allows us to judge, how well our fully-automated approach works.

We will first present some images with lights and windows lights.

Fig.16 presents an image from the facade dataset where the balcony label has been decorated. Since the sky does not have a totally uniform color, there is a slight non-ideality in the day to night transform.

(a) Original image (b) Decorated image

Fig. 16: Image from the facade dataset

The next Fig.17 looks convincing, however, the parameters chosen for other images make the result slightly too dark. Fig.18 uses the same configuration as

(a) Original image (b) Decorated image

Fig. 17: Image from the facade dataset with decorated windows

(a) Original image (b) Decorated image

Fig. 18: Image from the facade dataset with lighted windows

above, also decorating and lighting windows.

The guirlande approach in Fig.19 was discarded after some tries, however, we include the decoration here for completeness.

Since we support any colors for decoration, we decided to include a random color decoration aswell, even if it is not very festive, Fig.20.

In Fig.21 while the lights look convincing, the attentive reader may notice that the sky has not been uniformly darkened. This is due to the clouds in the sky and the sky darkening fails to detect the sky as one color. This could be circumvented by decreasing the number of clusters in the quantized image. Since we however decided to evaluate our approach without parameter tuning, we include the example here.

The image we are most proud of, including the full pipeline, can be found in Fig.22.

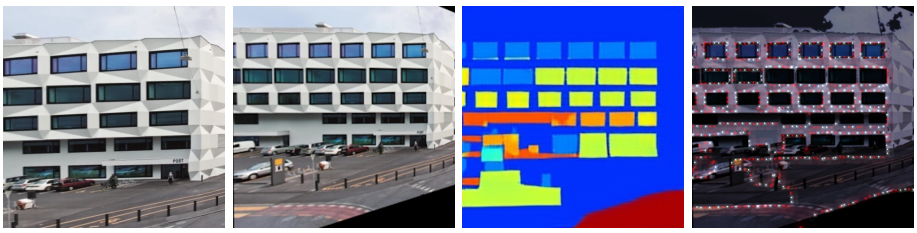
More results can be found in the attached folder `examples_christmasdecorator`.

(a) Original image (b) Decorated image

Fig. 19: Image decorated with guirlandes

(a) Original image (b) Decorated image

Fig. 20: Image decorated with random colors

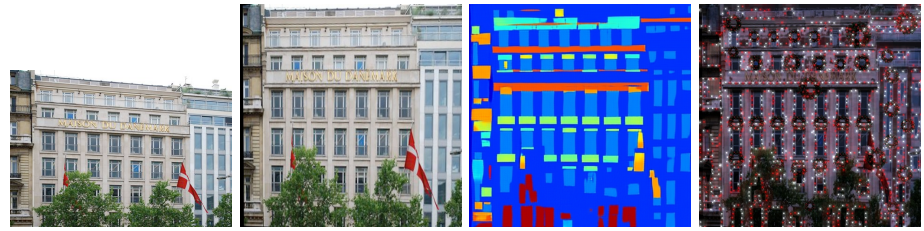


(a) (b) (c) (d)

Fig. 21: Full pipeline including homography transform

4.1 Pitfalls for decorations and imperfections

A key constraint to creating nicely looking decorations, is the resolution of the input images. While searching for the right parameters for adding Christmas lights, we often encountered very artificial looking lights. An idea to circumvent this issue would be to copy a template for an individual light. However, since resolution of our buildings from the CMP_facade_database was between 256x256



(a) (b) (c) (d)

(e)

Fig. 22: Best full pipeline including homography transform

and 512x512 pixels, we were quite constrained on the approaches to be done. The results, where thus with circles drawn by hand and template lights, looking as in Fig.23.

An idea to circumvent this, would be to upsample the image to a higher resolution and add the lights. However, if we do not downsample afterwards again, we end up with a blurred image. If we choose to downsample, the lights are than again as before and might look unrealistic.

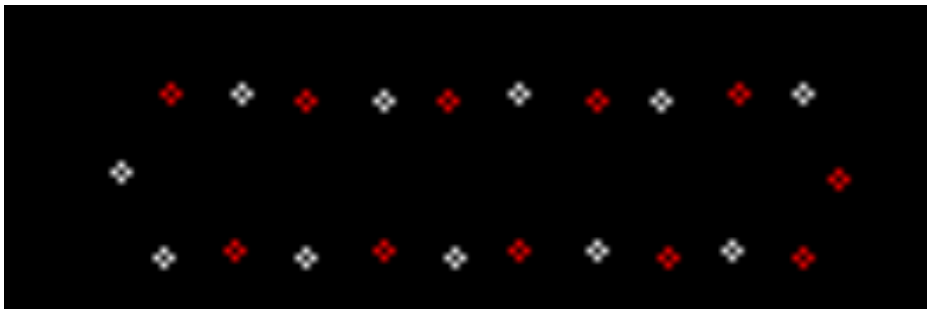


Fig. 23: Circles drawn on low-resolution image

5 Conclusion

We presented a fully automated method to bring images of buildings to nighttime images of buildings including Christmas decorations. While modern trends might suggest to pursue this topic with a deep learning approach, we decided to pursue a more traditional approach.

Our approach consist of a homography transform, segementation of our image into different regions, cleaning the masks obtained and performing day to night transfer with sky darkening, brightness, saturation, color and contrast changes.

References

1. Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks (2014)
2. Harris, C.G., Stephens, M., et al.: A combined corner and edge detector. In: Alvey vision conference. vol. 15, pp. 10–5244. Citeseer (1988)
3. Karras, T., Laine, S., Aila, T.: A style-based generator architecture for generative adversarial networks (2019)
4. Lloyd, S.P.: Least squares quantization in pcm. IEEE Transactions on Information Theory **28**, 129–137 (1982)
5. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics. pp. 281–297. University of California Press, Berkeley, Calif. (1967), `https://projecteuclid.org/euclid.bsmsp/1200512992`
6. Szeliski, R.: Computer Vision: Algorithms and Applications. Springer (2010)

7. Tyleček, R., Šára, R.: Spatial pattern templates for recognition of objects with regular structure. In: Proc. GCPR. Saarbrucken, Germany (2013)
8. Zhu, J.Y., Park, T., Isola, P., Efros, A.A.: Unpaired image-to-image translation using cycle-consistent adversarial networks (2020)