VFX Project 1

High Dynamic Range Imaging

B03901056 Fan-Keng Sun, B03901119 Shang-Wei Chen

1 Description

In this project, we assemble high dynamic range (HDR) images from a series of photographs under various exposures, using a popular vision library, OpenCV, for image processing and I/O. The photographs are preprocessed using median threshold bitmap (MTB) algorithm for image alignment. With the aid of tone mapping algorithm, HDR images are reproduced to LDR images in a better human perceptual sense. We learned basic photographing theories and image processing skills from the project. The features we have implemented are:

- Image alignment: MTB algorithm.
- HDR imaging: Paul Debevec's method.
- Tone mapping: Erik Reinhard's method.
- Exposure fusion: Tom Mertens' method.
- Blob removal
- Ghost removal: EA Khan's method.
- Spotlighting

2 Implementation

2.1 Environment

- Camera: Sony A6000 (lens: Sony SELP1650)
- OS: Linux (Archlinux 4.10, Ubuntu 16.04)
- Tools/Libraries: gcc/g++ 6.3.1, OpenCV 3.2 (C++), Boost >= 1.5, Cmake >= 3.0

2.2 Image alignment

Before HDR processing, image alignment is involved for better performance. The algorithm we used is shown in **Algorithm 1**, which is an implementation of **median threshold bitmap (MTB) algorithm** [1]. The function

```
void MTB::process(vector<Mat>& res)
```

takes `cv::Mat` images for input and pass the result by reference. The alignment function transforms these images to 8-bit grayscale ones, while the others may approximately use only the green channel since this channel is higher weighted when converting color. Applying multi-scale techniques, we then form a pyramid of each grayscale photo in which the photos are scaled by a factor of two in each dimension. Consequently, filtering the photos to generate bitmaps thresholded by the median in each photo; thus, the bitmaps can be viewed as black-and-white

photos. We can compute the amount of differences between two bitmaps using bitwise XOR, and find the optimal way to shift photos. Finally, scale these photos to crop the blank pixels on the edges resulted from shifting. There might be noisy areas at which the pixel luminance is close to the median; hence, adding a mask to exclude these areas may help following alignment processes. These features are realized by the following functions:

```
void MTB::transform_bi(const Mat& m, Mat& bi, Mat& de, int max_denoise)
void MTB::shift(Mat& m, Mat& dst, const pair<int, int>& diff)
pair<int, int> MTB::align(const int j, int lev, const int max_level)
```

One may refer to `src/mtb.hpp` and `src/mtb.cpp` for more details.

Algorithm 1 MTB algorithm [1]

$pics \leftarrow$ images read via OpenCV
$max\_level \leftarrow$ user parameter for aligning iterations
$max\_denoise \leftarrow$ user parameter for setting noise area
function MtbProcess($pics, max\_level, max\_denoise$)
    $N \leftarrow$ number of $pics$
    $bi\_pics, masks \leftarrow$ new image array with dimension N and max_level
    for i in range(N) **do**
        $p \leftarrow pics[i]$
        for j in range(max_level) **do**
            MtbTransformBi($p, bi\_pics[N][j], masks[N][j], max\_denoise$)
            $pic \leftarrow pic$ with half size
        end for
    end for
    $offsets \leftarrow$ array of size N
    for all i in range(N) **do**
        $offsets[i] \leftarrow$MtbAlign(i, 0, max_level)
    end for
    Shift $pics$ with $(xshift, yshift)$ pairs in $offsets$
    Scale $pics$ to crop blank pixels
end function

function MtbTransformBi($p, bi, de, max\_denoise$)
    $bi, de \leftarrow p$ converted to gray scale
    $m \leftarrow$ median value of all pixel in bi
    for all pixel i in bi **do**
        if $i.value < m$ **then**
            $i.value \leftarrow gray.value.min$
        else
            $i.value \leftarrow gray.value.max$
        end if
    end for
    for all pixel i in de **do**
        if $i.value < m - max\_denoise$ or $i.value > m + max\_denoise$ **then**
            $i.value \leftarrow gray.value.max$
        else
            $i.value \leftarrow gray.value.min$
        end if
    end for
end function

function MtbAlign($i, level, max\_level$)
  if $level$ equals max_level **then**
    return $(0, 0)$
  end if
  $(xshift, yshift) \leftarrow$MtbAlign$(i, level + 1, max_level)$ ▷ Call this function recursively
  $fixed, moved \leftarrow bi[0][level], bi[i][level]$ ▷ Align $pic[i]$ to $pic[0]$
  $mask\_f, mask\_m \leftarrow masks[0][level], masks[i][level]$
  for $xshift, yshift$ in $[-1, 0, 1]$ **do**
    $moved\_sh \leftarrow moved$ with $(xshift, yshift)$
    $cnt_k \leftarrow$ # of diff between $fixed$ and $moved\_sh$ without $mask\_f$ and $mask\_m$ area
  end for
  $(xshift\_best, yshift\_best) \leftarrow (xshift, yshift)$ with $\min\{cnt_k\}$
  return $(xshift * 2 + xshift\_best, yshift * 2 + yshift\_best)$
end function

2.3 HDR Imaging

In our project, we have implemented two methods for HDR imaging: **Paul Debevec's method** [2] and **Tom Mertens' method** [3] (referred to **Algorithm 2** and **Algorithm 3**, respectively).

As mentioned in the class, we construct matrix A and vector $\mathbf{b}$, and then compute the solution of $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$, where the indice of A and $\mathbf{b}$ are calculated from the aligned photos combined with a hat weighting function. Once the response curve recovery is done, we can construct the high dynamic range radiance map according to [2]

$$\ln E_i = g(Z_{ij}) - \ln \Delta t_j$$

For robustness, we choose to reduce noise in the result using the following function rather than the above one

$$\ln E_i = \frac{\sum_{j=1}^{P} w(Z_{ij})(g(Z_{ij} - \ln \Delta t_j))}{\sum_{j=1}^{P} w(Z_{ij})}$$

Debevec's method is realized in

```
void DEBEVEC::process(
  const vector<Mat>& pics,
  const vector<double>& etimes,
  const vector<Mat>& gW,
  Mat& result
)
```

in which the parameter `gW` is involved for **ghost removal** feature. We will discuss it in the following sections.

Algorithm 2 HDR algorithm using Paul Debevec's method [2]

$etimes \leftarrow$ exposure time of each photos
$lambda \leftarrow$ user parameter for smoothness
$nSample \leftarrow$ user parameter for amount of sampling points
$nPics \leftarrow$ number of photos in $pics$
$points \leftarrow nSample$ points randomly from $pics[0]$
function HdrDebevec($pics, etimes$)
  $w \leftarrow$ hat weighting function with min = 1 and max = 128

```
    for ch in range(3) do
        X[ch] ← new array of size 256
        A ← new array of zeros with size (nSample * nPics + 257) × (nSample + 256)
        B ←new array of zeros with size (nSample * nPics + 257) × 1
        k ← 0
        for i in range(nSample) do
            for j in range(nPics) do
                val ← pics[point[i]]
                A[k][val] ← w[val]
                A[k][256 + i] ← −w[val]
                B[k][0] ← w[val] * log(etimes[j])
                k ← k + 1
            end for
        end for
        A[k][128] ← 1
        for i in range(254) do
            k ← k + 1
            A[k][i] ← lambda * w[i]
            A[k][i + 1] ← −2 * lambda * w[i]
            A[k][i + 2] ← lambda * w[i]
        end for
        X[ch] ← (A/B)[0 to 255]
    end for
    for ch in range(3) do
        for pixel i in a photo of channel ch) do
            nom ← sum{w[i.value] * (pics[p][i.value] − log(etimes[j])) for each photo pics[p]}
            denom ← sum{w[i.value] for each photo pics[p]}
            res[ch][i] ← exp(nom/denom)
        end for
    end for
    return res
end function
```

For Mertens' method [3], **exposure fusion**, the algorithm keeps the best parts of each photos under various exposure realizing with weighted blending. In the first part, we compute some quality measures, **contrast**, **saturation**, and **well-exposedness**, for the purpose of producing a weighting function.

$$W_{ij,k} = (C_{ij,k})^{\omega_C} * (S_{ij,k})^{\omega_S} * (E_{ij,k})^{\omega_E}$$

in which the qualities, contrast (C), saturation (S), and well-exposedness (E) are given with different weighting exponents ω_C, ω_S, and ω_E, respectively. In the next part, fusion, we compute N normalized weight maps from above quantities

$$\hat{W}_{ij,k} = \left[\sum_{k'=1}^{N} W_{ij,k'}\right]^{-1} W_{ij,k}$$

The method then compute Laplacian pyramids and Gaussian pyrymids of the input images. Summing up their product of each photos to obtain the fused pyramid, we can recover the final image:

$$\mathbf{L}\{R\}_{ij}^{l} = \sum_{k=1}^{N} \mathbf{G}\{\hat{W}\}_{ij,k}^{l} \mathbf{L}\{I\}_{ij,k}^{l}$$

where R stands for the result image, and l for the level of pyramids. Finally, collapse the Laplacian pyramids to obtain HDR result. Mertens' method is realized in

```
void MERTENS::process(
  const vector<Mat>& pics,
  const vector<Mat>& gW,
  Mat& result
)
```

One may refer to `src/hdr.hpp` and `src/hdr.cpp` for more details.

Algorithm 3 HDR algorithm using Tom Mertens's method [3]

$wC, wS, wE \leftarrow$ user parameters for weighting functions
$max\_level \leftarrow$ user parameter for building pyramids
function HdrMertens($pics$)
    $nPics \leftarrow$ number of photos in $pics$
    $sPics \leftarrow pics$ where the value of pixels are scaled from int 0-255 to float number 0-1
    for i in range($nPics$) **do**
        $grayPic \leftarrow$ convert $sPics[i]$ to gray scale
        $C \leftarrow$ abs(LapacianFilter($grayPic$))
        $S \leftarrow$ standard_deviation(values of pixels in three channels of $sPics[i]$)
        $E \leftarrow$ GaussCurve($sPics[i].pixel.intensity - 0.5, \sigma = 0.2$)
        $W \leftarrow \text{pow}(C, wC) * \text{pow}(S, wS) * \text{pow}(E, wE)$
    end for
    $W \leftarrow W/\text{sum}\{W\}$
    $pyrPics \leftarrow$ BuildPyramid($sPics$)
    $pyrW \leftarrow$ BuildPyramid(W)
    $pyrRes \leftarrow$ pyramid of zeros with the same shape of $pyrPics$
    $picUp \leftarrow$ new image
    for i in range($nPics$) **do**
        for j in range(max_level) **do**
            PyramidUp($pyrPics[i][j+1]$, $picUp$, $pyrPics[i][j].size$)
            $pyrPics[i][j] \leftarrow pyrPics[i][j] - picUp$
        end for
        for j in range($max\_level + 1$) **do**
            $pyrPics[i][j] \leftarrow pyrPics[i][j] * pyrW[i][j]$
            $pyrRes[j] \leftarrow pyrRes[j] + pyrPic[i][j]$
        end for
    end for
    for i from max_level to 0 **do**
        PyramidUp($pyrRes[i+1]$, $picUp$, $pyrPics[i][j].size$)
        $pyrRes[i] \leftarrow pyrRes[i] + picUp$
    end for
    return $pyrRes[0]$
end function

2.4 Tone Mapping

We realize this part by **Erik Reinhard's method** [4], which is shown in **Algorithm 4**. The algorithm we used is an inspiration from photoreceptor physiology, which can be separated into two concepts: global and local reproduction. The part of global operators considers overall characteristics, such as contrast, brightness, color saturation, and etc., for visual perceptions.

On the other hand, the part of local operators focuses on local modifications, such as haloing and ringing.

```
void TONEMAP::process(Mat& input, Mat& output)
```

One may refer to `src/tonemap.hpp` and `src/tonemap.cpp` for more details.

Algorithm 4 Tone mapping algorithm [4]

$f \leftarrow$ user parameter for intensity
$m \leftarrow$ user parameter for contrast
$a \leftarrow$ user parameter for light adaption
$c \leftarrow$ user parameter for chromatic adaption
function TONEMAP($image\_in$, $image\_out$)
    $L\_map \leftarrow$ luminance of each pixel in $image\_in$
    $Cav[3] \leftarrow$ mean value of each channels of $image\_in$
    $Lav \leftarrow$ mean value of L_map
    $L\_min \leftarrow$ minimal value in L_map
    $L\_max \leftarrow$ maximal value in L_map
    for all channel n of $image\_in$ **do**
        for all pixel i in $image\_in[n]$ **do**
            $L \leftarrow$ value of the same position in L_map
            $I\_local \leftarrow c * image\_in[n][i] + (1 - c) * L$
            $I\_global \leftarrow c * Cav[n] + (1 - c) * Lav$
            $I\_adaption \leftarrow a * I\_local + (1 - a) * I\_global$
            $image\_out[n][i] \leftarrow image\_in[n][i]/(image\_in[n][i] + \text{pow}(f * I\_adaption, m))$
        end for
    end for
    Normalize the value of pixels in $image\_out$ to an integer in the range from 0 to 255
end function

2.5 Blob Removal

Owing to imperfection of original photos or some irregular situations that the above algorithms do not handle, there might be blobs on the photos. We implement blob removal based on a feature in OpenCV libraries, `cv::SimpleBlobDetector`. Using the following function (need to include `features2d.hpp`)

```
static Ptr<SimpleBlobDetector> cv::create(
  const SimpleBlobDetector::Params &parameters=SimpleBlobDetector::Params()
)
```

simply convert RGB color to gray scale only, which may not detect the blobs completely; what's more, splitting RGB color into three channels and thresholding by each channel are not effective enough. Hence, we transform RGB color to **Lab color space** and threshold the image by **L** (lightness), **a** (green-red color component), and **b** (blue-yellow color component) channels separately, which may have better performance for blob detection, and then use the original `cv::SimpleBlobDetector` functions to detect the blobs.

This feature is realized by the following function in `src/util.cpp`

```
void blob_removal(const Mat& pic, Mat& result)
```

Since `cv::SimpleBlobDetector` provides many parameters for blob detection, such as blob size, circularity, inertia ratio, convexity, and for convenience, we create a panel for tuning these

parameters (Fig. 1). One of the advantages is that we can manually figure out the correctness of blob detection, and thus prevent the detection from some unwanted results if performing automatically. However, iterations is required for completeness since different blobs may have their own threshold values.

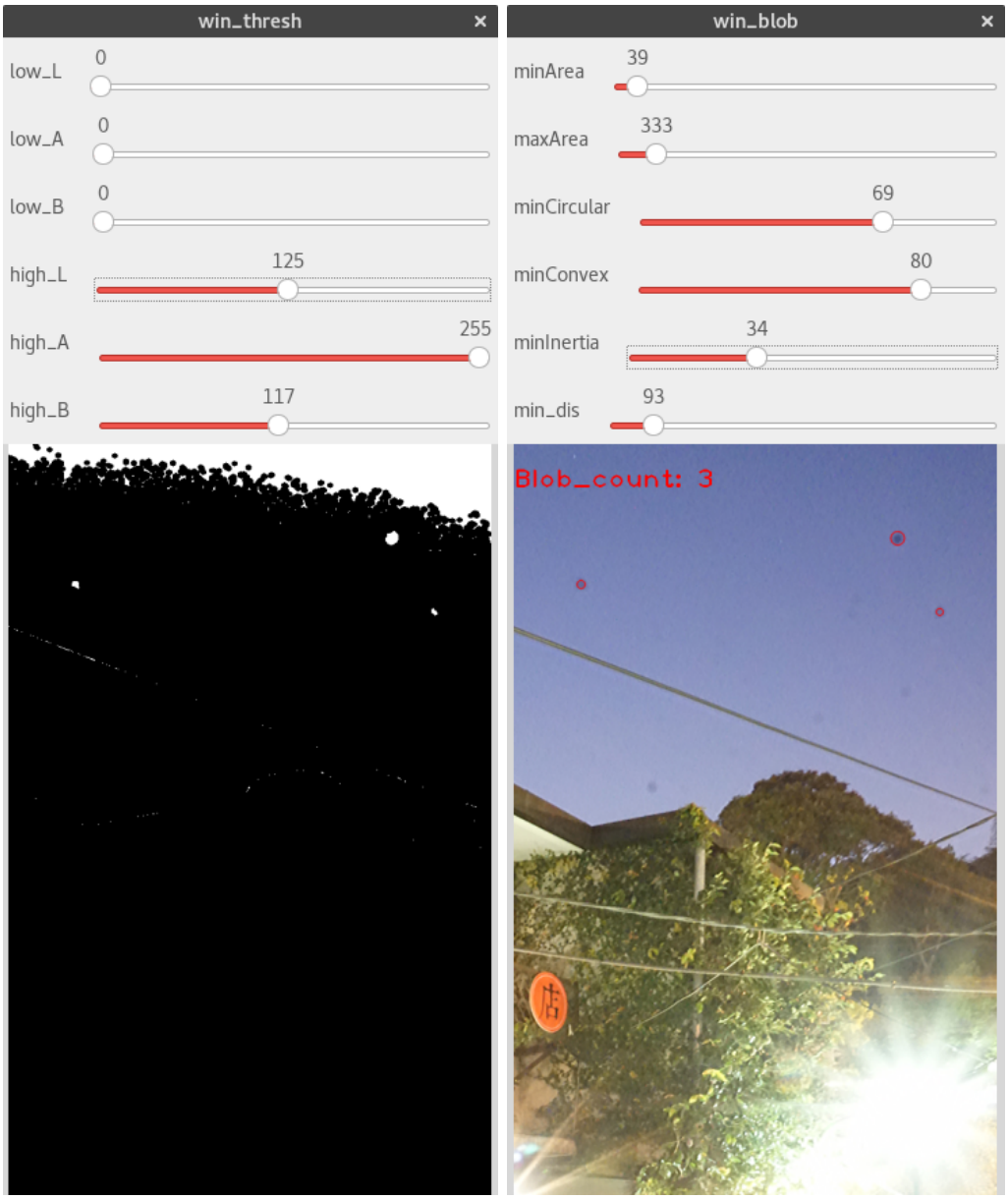


Figure 1: Panel for blob removal

Once the blobs are detected, we approximately fill out these areas with the average colors nearby. For more details, one can refer to `src/util.hpp` and `src/util.cpp`.

2.6 Ghost Removal

The feature is realized in the following function

```
void ghost_removal(const vector<Mat>& pics, int iter, vector<Mat>& result)
```

which is an implementation of **EA Khan's method** [5]. The algorithm iteratively calculate the possibility of each pixel based on its weight, and then calculate the weight for the next

iteration by its possibility. For a series of R photos under different exposure times, we assign a vector $\mathbf{x}_{ijr} = (L, a, b, i, j) \in \mathbb{R}^5$ to each pixel on a photo, where L, a, b represent its color, i, j represent its position, and r represents the number of photo that contains it. Define the neighborhood of the vector $\mathbf{x}_{ijr}$

$$F = \{\mathbf{y}_{pqs} | (p, q, s) \in \mathbf{N}(\mathbf{x}_{ijr}), (p, q) \neq (i, j), s = 1, 2, \cdots, R\}$$

In the first iteration, define the weight by a hat function (Fig. 2)

$$w(Z) = 1 - \left(2 \cdot \frac{Z}{255} - 1\right)^{12}$$

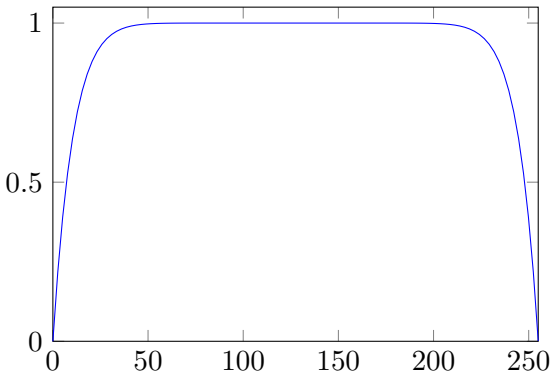


Figure 2: Hat function for ghost removal weighting

where Z is the average value of three channels. Then, calculate the possibility by

$$P(\mathbf{x}_{ijr}|F) = \frac{\sum_{p,q,s \in N(\mathbf{x}_{ijr})} w_{pqs} K_{\mathbf{H}}(\mathbf{x}_{ijr} - \mathbf{y}_{pqs})}{\sum_{p,q,s \in N(\mathbf{x}_{ijr})} w_{pqs}}$$

where

$$K_{\mathbf{H}}(\mathbf{x}) = |\mathbf{H}|^{-1/2} (2\pi)^{-5/2} \exp(-\frac{1}{2} \mathbf{x}^T \mathbf{H}^{-1} \mathbf{x})$$

and $\mathbf{H}$ is an identity matrix. Afterwards, calculate the weight for the next iteration

$$w_{pqs,t+1} = w(Z_s(p, q)) \cdot P(\mathbf{x}_{ijr}|F)$$

in which $w(Z_s(p, q))$ stands for the initial weight.

2.7 Spotlighting

Since there might be some objects moving irregularly when shooting photographs, and can barely rebuild the image by the previous ghost removal method, we use the function

```
void cv::seamlessClone(InputArray src, InputArray dst, InputArray mask, Point p,
    OutputArray blend, int flags)
```

in OpenCV libraries to recover the waving parts in the photos by simply replacing them based on one photo. The feature is realized in the following function

```
void add_spotlight(vector<Mat>& pics, const vector<double>& para)
```

One may refer to `src/util.cpp` for more detail.

3 Results

3.1 Response Curve

The recovered response curves are shown in Fig. 3, where the smoothness parameter λ is varied.

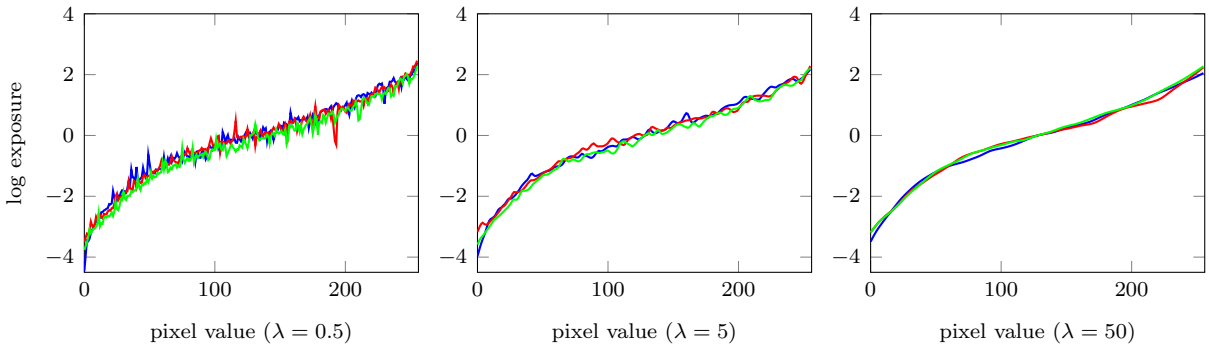


Figure 3: Response curve for Sony A6000

3.2 Blob removal

As shown in Fig. 4, the blobs in the photograph are removed successfully.

(a) Before (blobs are circled) (b) After

Figure 4: Result of blob removal

3.3 Spotlighting

As shown in Fig. 5, there exists an area of motion blur in (a). However, applying ghost removal can barely solidify the motion since the people have different postures overlap to each others in the photos. Put the SPOTLIGHT on the moving object in a single photo, and clone the spotlighted area seamlessly to other photos before HDR processes. As illustrated in (b), areas to be transplanted is recovered well.

(a) With motion blur (b) Without motion blur

Figure 5: Result of spotlighting

3.4 HDR Imaging and Tone Mapping

The proper user parameters for Reinhard's algorithm, f (intensity), m (contrast), a (light adaption), and c (chromatic adaption) can be various for different photos. Fig. 6 shows the LDR patterns of varying a and c.

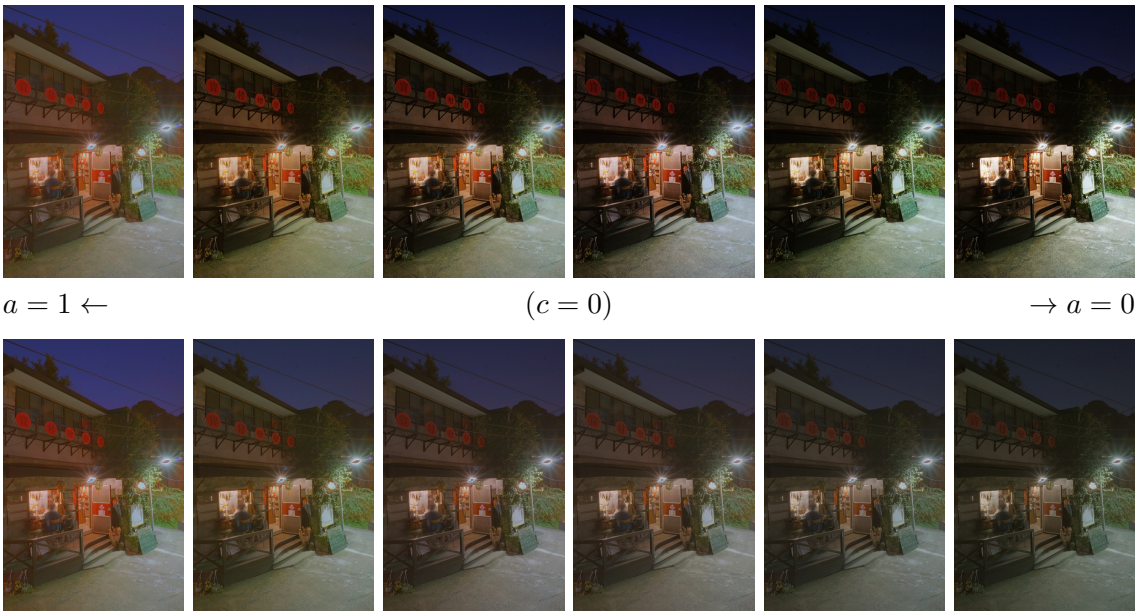


Figure 6: LDR patterns with various a and c

We can compare the results of two HDR method from the photos shown in Fig. 7.

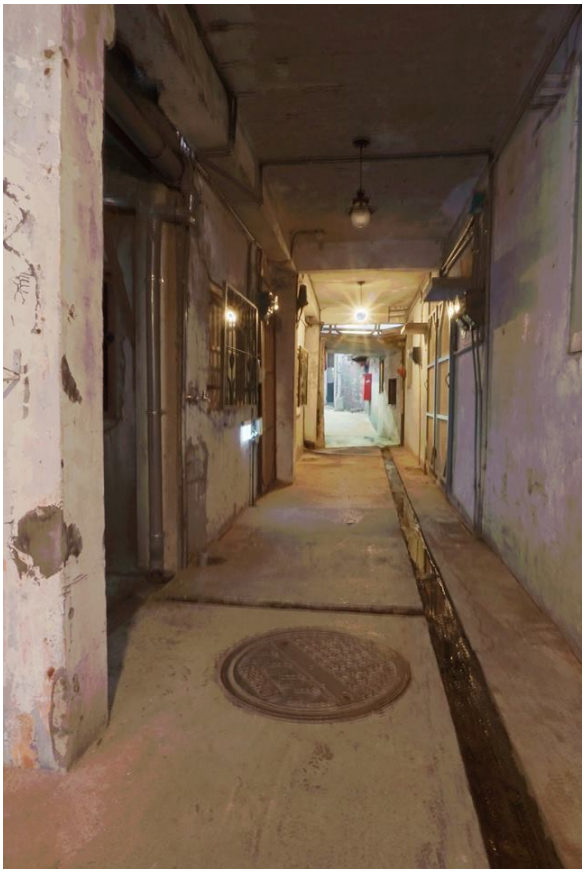
Comparing (a) and (b), Mertens' exposure and fusion method gives a better performance, where outer glows are produced at the edges of objects, and hence the subject of photo is more outstanding since more contrast. On the other hand, Debevec's method combining with Reinhard's tone mapping function preserve more details on color saturation; nevertheless, abnormal stains are produced inside the flares.

Comparing (c) and (d), where photos with middle exposure times are discarded for experiment, we discover that Mertens' method results in unnatural shadow distribution, and we speculate

that the algorithm considers extraordinary dark and bright parts to be outliers, so the they are fused with lower weight. For Debevec's and Reinhard's methods, contrast and brightness are recovered well but there are some abnormal color distributions in the result.

(a) Debevec

(b) Mertens



(c) Debevec

(d) Mertens

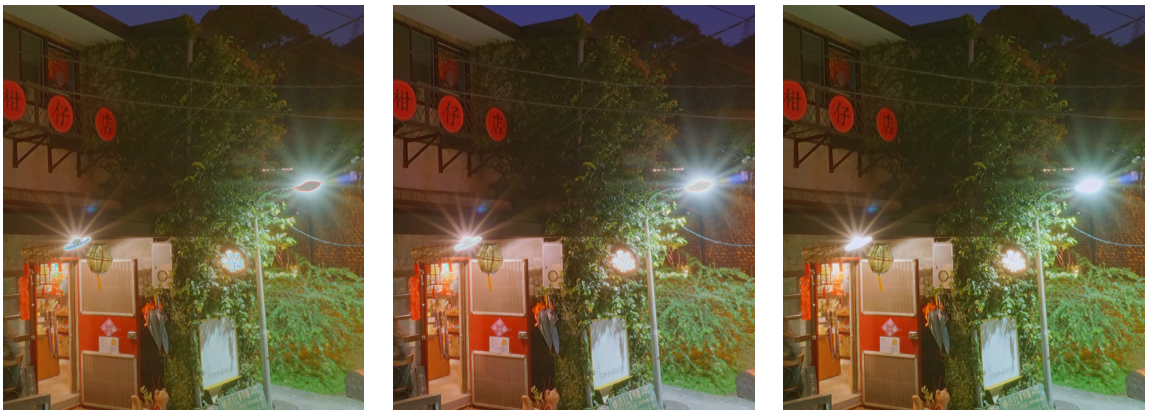
Figure 7: Comparison of the HDR methods

Trying to fix the problem of the occurence of dark stains in bright flares (where there might be in the extreme bright color), we modify Debevec's method by varying the weighting function. We can compare the result in Fig. 8. We replace the original hat function with the similar one to what we use in ghost removal:

$$w(Z) = 1 - \left(2 \cdot \frac{Z}{255} - 1\right)^{12} + d$$

where the difference between them, d, can be view as a step function applied in the original

one. Increasing d slightly in order to avoid failure in recovery of response curve, we find that the brightness distortion areas are distinguished.



(a) $d = 0$ (b) $d = 0.15$ (c) $d = 0.3$

Figure 8: Results of modifying the weight function in Debevec's method

Our favorite artifact is shown in Fig. 9.

Figure 9: Best artifact

4 Reference

[1] Greg Ward. Fast, robust image registration for compositing high dynamic range photographs from handheld exposures. *JOURNAL OF GRAPHICS TOOLS*, 8:17–30, 2003.

[2] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. *SIGGRAPH*, pages 369–378, 1997.

[3] T. Mertens, J. Kautz, and F. V. Reeth. Exposure fusion. *Computer Graphics and Applications, 2007. PG '07. 15th Pacific Conference on*, pages 382–390, 2007.

[4] Erik Reinhard and Kate Devlin. Dynamic range reduction inspired by photoreceptor physiology. *IEEE TVCG*, 11(1):13–24, 2005.

[5] E. A. Khan, A. O. Akyuz, and E. Reinhard. Ghost removal in high dynamic range images. *IEEE International Conference on Image Processing*, pages 2005–2008, 2006.