

1. Введение

Визуализация с высоким динамическим диапазоном (HDRI) предлагает радикально новый подход к представлению цветов в цифровых изображениях и видео. Вместо использования диапазона цветов, воспроизводимых данным устройством отображения, методы HDRI используют и сохраняют все цвета и уровни яркости, видимые человеческому глазу. Поскольку видимый диапазон цветов намного превышает диапазон, достижимый камерами или дисплеями, цветовое пространство HDR, в принципе, представляет собой надмножество всех цветовых пространств, используемых в традиционном стандартном изображении с динамическим диапазоном.



Standard (Low) Dynamic Range



High Dynamic Range

up to 500 cd/m ²	peak brightness	2 000-10 000 cd/m ²
50 dB	camera dynamic range	120 dB
1:1 000	display contrast	1:1 000 000
from 8 to 16 bit	quantization	floating point or variable
display-referred	image representation	scene-referred
display-limited	fidelity	as good as the eye can see

Рисунок 1. Преимущества HDR по сравнению с LDR с точки зрения приложения. Качество изображения LDR было специально снижено, чтобы проиллюстрировать потенциальные различия между визуальным содержимым HDR и LDR, видимым на дисплее HDR.

2. Преимущества HDR

Улучшенный диапазон яркости: HDR-изображения могут отображать более детальные тени и светлые участки, что приближает изображение к тому, как человеческий глаз воспринимает реальный мир.

Улучшенный контраст и цветовая насыщенность: HDR улучшает контраст и делает цвета более насыщенными и живыми.

Уменьшение шума: Сочетание нескольких экспозиций может помочь уменьшить шум на изображении.

3. Метод Дебвека

Описание метода

Метод, разработанный Полом Дебвеком, используется для создания HDR-изображений путем восстановления функции отклика камеры. Это позволяет точно соединить несколько изображений с разной экспозицией в одно HDR-изображение.

Математические основы

Функция отклика камеры (CRF): Это функция, которая связывает известные значения пикселей (Z) с логарифмом их истинной экспозиции (логарифмическая экспозиция). Функция отклика камеры показывает, как каждое значение пикселя соответствует определенному уровню освещенности в сцене. Она важна, потому что разные камеры могут реагировать на свет по-разному, и эта функция помогает стандартизировать эти различия для точного воссоздания HDR-изображения.

Предположим у нас есть нелинейная функция f , отображающая экспозицию X в пиксельных значения Z ,

$$f(X) = Z$$

Затем реконструкция карты освещенности по значениям пикселей может быть получена путем применения обратной функции f^{-1} , где предполагается, что f является гладкой и непрерывной функцией.

$$X = f^{-1}(Z)$$

Если выражать через освещенность для каждого i пикселя и время экспозиции для каждого j изображения:

$$E_i \Delta t_j = f^{-1}(Z_{ij}),$$

где Z_{ij} – значение i -го пикселя (от 0 до 255 в 8-битовом изображении) j – го изображения.

Это выражение можно переписать как:

$$CRF(Z_{ij}) = \ln(E_i) + \ln(\Delta t_j),$$

где

$$CRF = \ln(f^{-1})$$

Взвешенная целевая функция для оптимизации CRF: Используя серию изображений с разными экспозициями, метод восстанавливает функцию отклика камеры.

$$\begin{aligned} O = & \sum_{i=1}^N \sum_{j=1}^P \omega(Z_{ij}) \cdot [CRF(Z_{ij}) - \ln(E_j) - \ln(\Delta t_i)]^2 \\ & + \lambda \sum_{z=z_{min}+1}^{z_{max}-1} [\omega(z) CRF''(z)]^2 \end{aligned}$$

где N – количество изображений, P – количество выбранных пикселей, $\omega(Z)$ – весовая функция, которая придает больше веса пикселям средней яркости, λ – параметр сглаживания, $CRF''(z)$ – вторая производная функции отклика.

Первый член гарантирует, что решение удовлетворяет уравнению $CRF(Z_{ij})$ в смысле наименьших квадратов. Второй член дает гладкость, где вторая производная задается в виде дискретной формы:

$$CRF''(z) = CRF(z - 1) - 2CRF(z) + CRF(z + 1),$$

и λ - это вес второго члена над первым.

$\omega(z)$ - это весовая функция для задания условий плавности и подгонки к середине кривой отклика, которая выбирается как простая треугольная функция, поскольку

$$\omega(z) = \begin{cases} z - Z_{min} , & \text{для } z \leq \frac{1}{2}(Z_{min} + Z_{max}) \\ Z_{max} - z , & \text{для } z > \frac{1}{2}(Z_{min} + Z_{max}) \end{cases}$$

При измерении кривой отклика расположение пикселей следует выбирать таким образом, чтобы они были достаточно равномерно распределены в пределах $\{Z_{min} \dots Z_{max}\}$, чтобы пиксели были хорошо отобраны из изображений.

Восстановление карты яркости HDR изображения

Как только кривые отклика будут получены, исходные изображения могут быть объединены в единое HDR-изображение с помощью

$$\ln(E_j) = \frac{\sum_{j=1}^P \omega(Z_{ij}) \cdot [g(Z_{ij}) - \ln(\Delta t_i)]}{\sum_{j=1}^P \omega(Z_{ij})},$$

где P – количество изображений, то есть количество различных периодов экспозиции, Z_{ij} – значение i -го пикселя на j -м изображении.

Комбинирование нескольких экспозиций приводит к снижению уровня шума.

Рассмотри подробнее реализацию на языке Python:

1. Загрузка изображений

```
path = "uploads/"
filenames = ['image0.jpg', 'image1.jpg', 'image2.jpg']
exposure_times = [1/6.0, 1.3, 5.0]
mergeDebevec = HDR(path, filenames, exposure_times)
with ThreadPoolExecutor() as executor:
    self.images = list(executor.map(lambda fn:
        cv2.imread(''.join([path, fn])), filenames))
    self.times = np.array(exposure_times, dtype=np.float32)
    self.N = len(self.images)
    self.row = len(self.images[0])
    self.col = len(self.images[0][0])
    # Коэффициент веса для плавности кривых отклика
    self.l = 10
```

Начинаем мы с загрузки изображений с различным временем экспозиции, начиная с недоэкспонированного изображения (с наименьшим временем экспозиции) до переэкспонирования (с наибольшим временем экспозиции).

▼ Подробнее:

Размеры: 6720 × 4480
Изготовитель устройства: Canon
Модель устройства: Canon EOS 5D Mark IV
Цветовое пространство: RGB
Цветовой профиль: sRGB IEC61966-2.1
Фокусное расстояние: 16 мм
Альфа-канал: Нет
Красные глаза: Нет
Режим измерения: Матричный
Диафрагменное число: f/6,3
Программа экспозиции: Приоритет по диафрагме
Время экспозиции: 1,3

▼ Имя и расширение:

image1.jpg Скрыть расширение

➤ Комментарии:

➤ Открывать в приложении:

▼ Просмотр:



Рис.1. Информация об изображении

Вводим время экспозиции для каждого изображения соответственно. Приведенный выше блок содержит также объявление переменных, связанных с изображениями, $N = 3$ – количество изображений, $row = 6720$ и $col = 4480$ отвечают за высоту и ширину изображений в пикселях. l обозначает λ в качестве весового коэффициента для сглаживания CRF.



Рис.2. Исходные изображения

Далее воспользуемся библиотекой OpenCV для выравнивания изображений друг относительно друга на основе метода median threshold bitmap (MTB) для исключения ошибок и смазы.

```
# Выравнивание исходных изображений
alignMTB = cv2.createAlignMTB()
alignMTB.process(self.images, self.images)
```

2. Весовая функция

Ниже представлен блок для заполнения весовой функции, учитывая что мы имеем 8-битное изображение:

```
Zmin = 0 # Минимальное значение пикселя (черный).
Zmax = 255 # Максимальное значение пикселя (белый).
Zmid = (Zmax + Zmin) // 2

self.w = np.zeros((Zmax - Zmin + 1))

for z in range(Zmin, Zmax + 1):
    if z <= Zmid:
        self.w[z] = z - Zmin + 1
    else:
        self.w[z] = Zmax - z + 1
```

Построенная весовая функция показана ниже, и эта функция будет присваивать вес значениям пикселей около медианы 255 при восстановлении HDR-карты.

3. Кривые отклика камеры

Кривая отклика получается путем выборки пикселей из заданных изображений. Количество образцов определяется $n_{sampling}(P - 1) > Z_{max} - Z_{min}$, что обеспечивает достаточно переопределенную систему.

```
samples = math.ceil(255 * 2 / (self.N - 1)) * 2 # Количество образцов для выборки.
pixels = self.row * self.col # Общее количество пикселей в изображении.

# Определение шага выборки для равномерного распределения выборки по всему изображению.
step = int(np.floor(pixels / samples)) # Шаг выборки пикселей.
self.indices = list(range(0, pixels, step))[:-1] # Индексы выбранных пикселей.

# Создание плоского (одномерного) представления изображений для упрощения доступа к пикселям.
self.flattenImage = np.zeros((self.N, 3, pixels), dtype=np.uint8) # Массив для плоских изображений.

def flatten_channel(channel_index):
    for i in range(self.N):
        # Преобразование каждого цветового канала в одномерный массив.
        self.flattenImage[i, channel_index] = np.reshape(self.images[i][:, :, channel_index], (pixels,))

# Использование многопоточности для ускорения процесса плоского представления изображений
with ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(flatten_channel, range(3))

# Логарифмирование времен экспозиции для использования в математических расчетах.
X = np.log(self.times) # Логарифмические времена экспозиции.
```

Этот блок кода показывает, что мы отбирали пиксели с равным интервалом, и индексы пикселей хранятся в «sampleIndices». Следующий код показывает, что выборка производится по отдельным каналам.

```
# Массивы для хранения значений пикселей в выбранных местах
self.ZG = np.zeros((numSamples, self.N), dtype=np.uint8)
self.ZB = np.zeros((numSamples, self.N,), dtype=np.uint8)
self.ZR = np.zeros((numSamples, self.N), dtype=np.uint8)

# Получение выбранных значений пикселей
for k in range(self.N):
    self.ZB[:, k] = self.flattenImage[k, 0][self.sampleIndices]
    self.ZG[:, k] = self.flattenImage[k, 1][self.sampleIndices]
    self.ZR[:, k] = self.flattenImage[k, 2][self.sampleIndices]
```

```

ind = np.arange(0, numSamples) # Индексы всех выборок.

# Обновление массивов значений пикселей после исключения неподходящих
выборок.
self.ZB = self.ZB[ind]
self.ZG = self.ZG[ind]
self.ZR = self.ZR[ind]

# Массивы для хранения логарифмических времен экспозиции
r, c = self.ZG.shape[:2]
self.Bij = np.tile(np.log(self.times), (self.row * self.col, 1))

```

Отсев выборок, где значения пикселей уменьшаются с увеличением экспозиции, считается нелогичным, потому что в нормальных условиях, когда экспозиция увеличивается (то есть, когда на сенсор камеры попадает больше света), ожидается, что значения пикселей также увеличивается. Это основное предположение фотографии: если вы увеличиваете время, в течение которого свет воздействует на пиксель (экспозиция), то пиксель должен стать ярче, что означает увеличение его значения.

Кроме того, время воздействия необходима для построения кривой отклика. Предполагается, что E равно 1, таким образом, вклад E в экспозицию X не учитывается в виде логарифма $\log(E) = 0$.

Для нахождения функции отклика камеры и логарифмической освещенности решается система уравнений методом наименьших квадратов. Метод наименьших квадратов ищет такое решение x , которое минимизирует сумму квадратов разностей между левой и правой частями уравнений системы.

Давайте разберемся, почему матрица U и вектор V заполняются именно таким образом в коде ниже и рассмотрим формулу метода наименьших квадратов.

```

n = 256

s1, s2 = Z.shape
U = np.zeros((s1 * s2 + n + 1, n + s1))
V = np.zeros((U.shape[0], 1))

```

```

# Здесь создаются матрица U и вектор V, которые будут использоваться для
решения системы линейных уравнений.
# Размеры этих матриц зависят от количества пикселей в изображениях
(self.Z.shape) и количества возможных значений пикселей (256 для 8-битного
изображения).

# Включение уравнений для соответствия данным
k = 0
for i in range(s1):
    for j in range(s2):
        wij = self.w[Z[i, j]]
        U[k, Z[i, j]] = wij
        U[k, n + i] = -wij
        V[k] = wij * self.Bij[i, j]
        k += 1

# Этот двойной цикл заполняет матрицу U и вектор V.
# Для каждого пикселя в каждом изображении создается уравнение, которое
связывает значение пикселя с соответствующим значением времени экспозиции.
# Веса wij используются для уменьшения влияния пикселей с очень высокой
или очень низкой освещенностью.

# Фиксация кривой путем установки ее среднего значения в ноль
U[k, 129] = 0
k += 1

# Включение уравнений для плавности
for i in range(1, n - 2):
    U[k, i] = self.l * self.w[i + 1]
    U[k, i + 1] = -2 * self.l * self.w[i + 1]
    U[k, i + 2] = self.l * self.w[i + 1]
    k += 1
# Здесь добавляются уравнения для гладкости функции отображения камеры.
# Это помогает гарантировать, что функция будет изменяться плавно, что
важно для точного восстановления освещенности.

```

Заполнение матрицы U и вектора V

1. Соответствие данным:

Каждая строка матрицы U и соответствующий элемент вектора V представляют уравнение, связывающее значение пикселя с его логарифмическим временем экспозиции. Это уравнение выглядит как

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

2. Веса $\omega(Z_{ij})$:

Веса используются для уменьшения влияния пикселей с очень высокой или очень низкой освещенностью, которые могут быть менее точными из-за ограничений камеры.

3. Уравнения для плавности:

Добавляются для обеспечения плавности функции отклика камеры. Это помогает избежать резких перепадов в функции отклика, что важно для точного восстановления освещенности.

$$U[k, Z_{ij}] = \omega(Z_{ij})$$

соответствует значению функции отклика Z_{ij} .

Он умножается на весовой коэффициент

$$U[k, n + i] = -\omega(Z_{ij})$$

соответствует логарифму освещенности

Он умножается на отрицательный весовой коэффициент

$$V[k]$$

соответствует логарифму времени экспозиции

Пример:

Допустим у нас есть $s_1 = 3$ пикселя и $s_2 = 2$ изображения и значения пикселей равны:

$$Z_{00} = 1, Z_{01} = 2$$

$$Z_{10} = 2, Z_{11} = 3$$

$$Z_{20} = 3, Z_{21} = 1$$

Также предположим $n = 4$ возможных значений функции отклика и $\lambda = 1$ для упрощения.

Тогда матрица U будет выглядеть так:

U	$g(1)$	$g(2)$	$g(3)$	$g(4)^*$	$\ln E_1$	$\ln E_2$	$\ln E_3$
<i>Пиксель 1, изображение 1</i>	ω_{11}	0	0	0	$-\omega_{11}$	0	0
<i>Пиксель 1, изображение 2</i>	0	ω_{12}	0	0	$-\omega_{12}$	0	0

<i>Пиксель 2, изображение 1</i>	0	ω_{21}	0	0	0	$-\omega_{21}$	0
<i>Пиксель 2, изображение 2</i>	0	0	ω_{22}	0	0	$-\omega_{22}$	0
<i>Пиксель 3, изображение 1</i>	0	0	ω_{31}	0	0	0	$-\omega_{31}$
<i>Пиксель 3, изображение 2</i>	ω_{32}	0	0	0	0	0	$-\omega_{32}$
<i>Фиксация кривой</i>	0	0	0	0	0	0	0
<i>Плавность 1**</i>	λ	-2λ	λ	0	0	0	0
<i>Плавность 2**</i>	0	λ	-2λ	λ	0	0	0

*Значения пикселей варьируются от 1 до 3, поэтому $g(4)$ везде нулевой

**Количество строк плавности определяется так, чтобы охватить весь диапазон значений функции отклика .

А вектор V :

V
$\omega_{11} \cdot \ln \Delta t_1$
$\omega_{12} \cdot \ln \Delta t_2$
$\omega_{21} \cdot \ln \Delta t_1$
$\omega_{22} \cdot \ln \Delta t_2$
$\omega_{31} \cdot \ln \Delta t_1$
$\omega_{32} \cdot \ln \Delta t_2$
0
0
0

Когда система переопределена (то есть уравнений у нас больше, чем количество неизвестных), то точного решения у нас может не существовать. В таком случае мы ищем решение , которое минимизирует сумму квадратов ошибок , то есть решает задачу минимизации: $(Ux - V)^2 \rightarrow \min$.

В методе наименьших квадратов используется псевдообратная матрица U^+ для матрицы U

Формула метода наименьших квадратов для решения системы линейных уравнений $Ux = V$ выглядит следующим образом:

$$x = (U^+ U)^{-1} U^+ V$$

Где:

- U - матрица коэффициентов системы уравнений
- U^+ - псевдообратная матрица, так что $U^+ U$ максимально близка к единичной матрице,
- V - вектор свободных членов.
- x - вектор неизвестных, который мы хотим найти (в нашем случае, это объединение функции отклика камеры и логарифмических значений освещенности).

Где

$$(Ux - V)^2 \rightarrow \min$$

```
# Решение системы с использованием метода наименьших квадратов
M = np.dot(np.linalg.pinv(U), V)
CRF = M[0:n]
logE = M[n: len(M) ]

return CRF, logE
```

На рисунке ниже показаны построенные кривые для трех цветовых каналов.

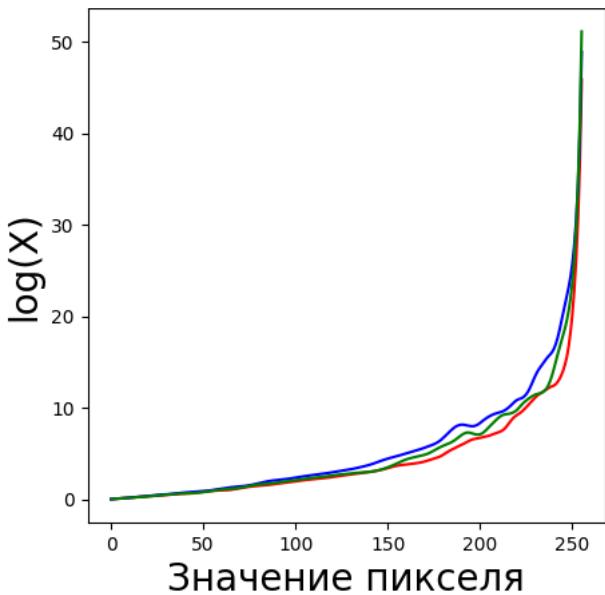


Рис.3. Кривые отклика

4. Объединение изображений

```

def recover_HDR_RadianceMap(self):
    m = np.zeros((self.flattenImage.shape[1:]))
    wsum = np.zeros(self.flattenImage.shape[1:])
    hdr = np.zeros(self.flattenImage.shape[1:])

    lnDt = np.log(self.times)

    for i in range(self.N):
        wij_B = self.w[self.flattenImage[i, 0]]
        wij_G = self.w[self.flattenImage[i, 1]]
        wij_R = self.w[self.flattenImage[i, 2]]

        wsum[0, :] += wij_B
        wsum[1, :] += wij_G
        wsum[2, :] += wij_R

        m0 = np.subtract(self.gB[self.flattenImage[i, 0]], lnDt[i])[:, 0]
        m1 = np.subtract(self.gG[self.flattenImage[i, 1]], lnDt[i])[:, 0]
        m2 = np.subtract(self.gR[self.flattenImage[i, 2]], lnDt[i])[:, 0]

        hdr[0] = hdr[0] + np.multiply(m0, wij_B)
        hdr[1] = hdr[1] + np.multiply(m1, wij_G)
        hdr[2] = hdr[2] + np.multiply(m2, wij_R)

    hdr = np.divide(hdr, wsum)
    hdr = np.exp(hdr)
    hdr = np.reshape(np.transpose(hdr), (self.row, self.col, 3))

    self.imgf32 = (hdr / np.amax(hdr) * 255).astype(np.float32)
    self.save_hdr_image() # Сохраняем HDR-изображение перед тоновым
    отображением

```



Изображения объединяются, в результате чего получается восстановленная карта яркости, как показано ниже.

Здесь создаются массивы **m**, **wsum**, и **hdr** для хранения промежуточных и конечных результатов. **lnDt** - это логарифм времен экспозиции, используемых для съемки серии изображений. Для каждого изображения в серии вычисляются веса **wij_R**, **wij_G**, и **wij_B** для каждого канала (синего, зеленого и красного соответственно). Эти веса используются для уменьшения влияния пикселей с очень высокой или очень низкой освещенностью. **m0**, **m1**, и **m2** вычисляются как разность между значениями функции отображения камеры (**gR**, **gG**, **gB** для каждого цветового канала) и логарифмом времени экспозиции. Эти значения умножаются на соответствующие веса и добавляются к **hdr**, что помогает восстановить карту яркости. **hdr** делится на сумму весов **wsum**, чтобы нормализовать результаты, а затем применяется экспоненциальная функция для преобразования логарифмических значений обратно в освещенность. После этого **hdr** преобразуется в формат изображения.

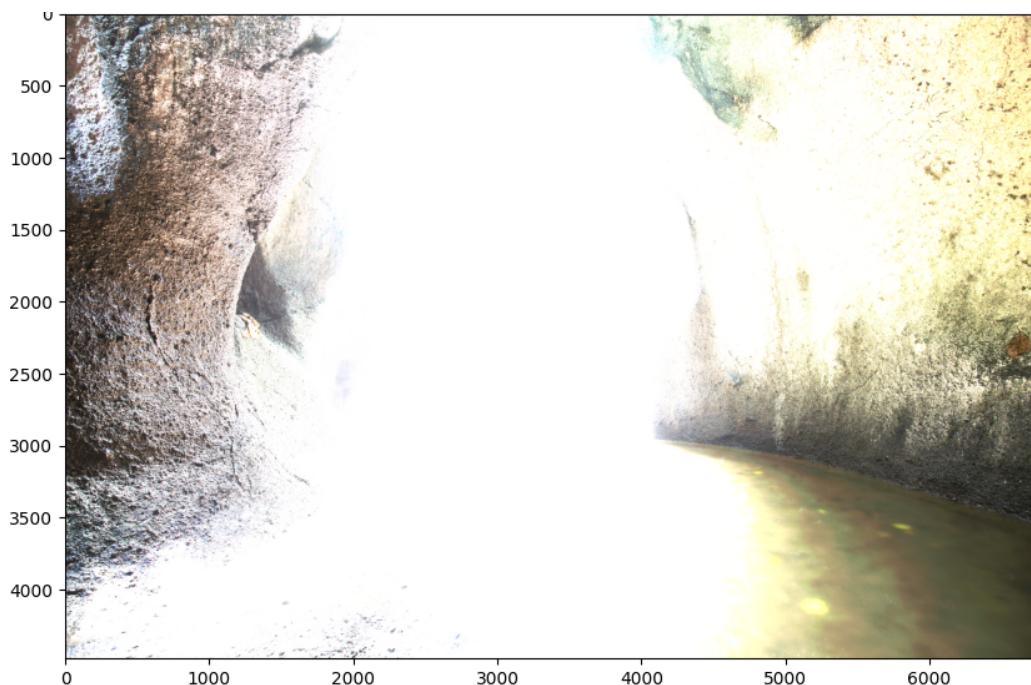


Рис.4. Карта освещенности HDR изображения

Список литературы:

1. Paul Debevec, Jitendra Malik: Recovering high dynamic range radiance maps from photographs. University of California at Berkeley
2. Li Zhang, Alok Deshpande, Xin Chen: Denoising vs. Deblurring: HDR Imaging Techniques Using Moving Cameras. University of Wisconsin, Madison
3. Rafał K. Mantiuk, Karol Myszkowski, Hans-Peter Seidel: High Dynamic Range Imaging
4. Satya Mallick: High Dynamic Range (HDR) Imaging using OpenCV (C++/Python)
5. F. Drago, K. Myszkowski, T. Annen, N. Chib: Adaptive Logarithmic Mapping For Displaying High Contrast Scenes. Iwate University, Morioka, Japan. MPI Informatik, Saarbrücken, Germany.

Приложение 1. Полный код проекта

```
import cv2
import os
import numpy as np
from IPython.display import Image
import matplotlib.pyplot as plt
from matplotlib import gridspec
import math
import warnings
warnings.filterwarnings("ignore")
import time
from concurrent.futures import ThreadPoolExecutor
from flask import Flask, render_template, request, redirect, url_for,
send_file, session, send_from_directory
from flask_session import Session # Добавьте эту библиотеку
from werkzeug.utils import secure_filename

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = 'uploads/'
app.config['RESULTS_FOLDER'] = 'results/'
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)

os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)
os.makedirs(app.config['RESULTS_FOLDER'], exist_ok=True)
path = app.config['UPLOAD_FOLDER']
filenames = ['uploads/image0.jpg', 'uploads/image1.jpg',
'uploads/image2.jpg']
exposure_times = [1/6.0, 1.3, 5.0]
```

```

class HDR:

    def __init__(self, path, filenames, exposure_times):
        """
        Определение класса HDR, который будет обрабатывать изображения для
        создания HDR-изображения.
        Конструктор загружает изображения, устанавливает времена экспозиции и
        выравнивает их с помощью алгоритма MTB (Median Threshold Bitmap).
        Класс HDR предназначен для создания изображений с высоким
        динамическим диапазоном (HDR) из серии фотографий с разной экспозицией.
        Это достигается путем объединения информации о свете и тени из разных
        изображений для создания одного изображения с более широким диапазоном
        яркости.
        """

        with ThreadPoolExecutor() as executor:
            self.images = list(executor.map(cv2.imread, filenames))

        self.times = np.array(exposure_times, dtype=np.float32)
        self.N = len(self.images)
        self.row = len(self.images[0])
        self.col = len(self.images[0][0])

        # Выравнивание исходных изображений
        alignMTB = cv2.createAlignMTB()
        alignMTB.process(self.images, self.images)

        # Коэффициент веса для плавности кривых отклика
        self.l = 10

    def weightingFunction(self):
        """
        Метод weightingFunction создает весовую функцию, которая придает
        больший вес хорошо экспонированным пикселям и меньший вес плохо
        экспонированным.
        Это важно для минимизации шума и артефактов в конечном HDR-
        изображении.
        """

        # Определение минимального и максимального значения пикселей, которые
        # могут быть в изображении.
        Zmin = 0 # Минимальное значение пикселя (черный).
        Zmax = 255 # Максимальное значение пикселя (белый).
        Zmid = (Zmax + Zmin) // 2

        # Инициализация массива для весовой функции, который будет хранить
        # вес каждого возможного значения пикселя.
        self.w = np.zeros((Zmax - Zmin + 1))

        # Заполнение весовой функции значениями.
        # Веса распределяются таким образом, чтобы значения пикселей, близкие
        # к среднему (128), имели больший вес,
        # а значения, близкие к краям диапазона (0 или 255), имели меньший
        # вес.
        for z in range(Zmin, Zmax + 1):
            if z <= Zmid:
                self.w[z] = z - Zmin + 1
            else:
                self.w[z] = Zmax - z + 1

    def samplingValues(self):
        """
        Метод samplingValues выбирает пиксели из каждого изображения для

```

определения функции отклика камеры.

Это ключевой шаг в процессе создания HDR, так как функция отклика камеры необходима для правильного объединения различных экспозиций.

```
'''  
    # Определение количества образцов для выборки на основе количества изображений и диапазона значений пикселей.  
    samples = math.ceil(255 * 2 / (self.N - 1)) * 2 # Количество образцов для выборки.  
    pixels = self.row * self.col # Общее количество пикселей в изображении.  
  
    # Определение шага выборки для равномерного распределения выборки по всему изображению.  
    step = int(np.floor(pixels / samples)) # Шаг выборки пикселей.  
    self.indices = list(range(0, pixels, step))[:-1] # Индексы выбранных пикселей.  
  
    # Создание плоского (одномерного) представления изображений для упрощения доступа к пикселям.  
    self.flattenImage = np.zeros((self.N, 3, pixels), dtype=np.uint8) # Массив для плоских изображений.  
  
    def flatten_channel(channel_index):  
        for i in range(self.N):  
            # Преобразование каждого цветового канала в одномерный массив.  
            self.flattenImage[i, channel_index] =  
                np.reshape(self.images[i][:, :, channel_index], (pixels,))  
  
        # Использование многопоточности для ускорения процесса плоского представления изображений  
        with ThreadPoolExecutor(max_workers=3) as executor:  
            executor.map(flatten_channel, range(3))  
  
        # Логарифмирование времен экспозиции для использования в математических расчетах.  
        X = np.log(self.times) # Логарифмические времена экспозиции.  
  
        # Массивы для хранения значений пикселей в выбранных местах  
        self.ZB = np.zeros((samples, self.N), dtype=np.uint8)  
        self.ZG = np.zeros((samples, self.N), dtype=np.uint8)  
        self.ZR = np.zeros((samples, self.N), dtype=np.uint8)  
  
        # Получение выбранных значений пикселей  
        for k in range(self.N):  
            self.ZB[:, k] = self.flattenImage[k, 0][self.indices]  
            self.ZG[:, k] = self.flattenImage[k, 1][self.indices]  
            self.ZR[:, k] = self.flattenImage[k, 2][self.indices]  
  
        ind = np.arange(0, samples) # Индексы всех выборок.  
  
        # Обновление массивов значений пикселей после исключения неподходящих выборок.  
        self.ZB = self.ZB[ind]  
        self.ZG = self.ZG[ind]  
        self.ZR = self.ZR[ind]  
  
        # Массивы для хранения логарифмических времен экспозиции  
        r, c = self.ZG.shape[:2]  
        self.Bij = np.tile(np.log(self.times), (self.row * self.col, 1))  
  
def CRFsolve(self, Z):  
    '''
```

Метод *CRFsolve* решает систему уравнений для получения функции отклика камеры и логарифмических значений освещенности.

Это математически сложный процесс, который требует решения большого количества уравнений для получения точной функции отклика.

Задан набор значений пикселей, наблюдаемых для нескольких пикселей на нескольких изображениях с разным временем экспозиции,

эта функция возвращает функцию отклика камеры g , а также логарифмические значения освещенности для наблюдаемых пикселей

Входные значения:

$Z(i, j)$: пиксельные значения местоположений пикселей с номером i на изображении j

$B(j)$: логарифмическая дельта t для изображения j

λ : лямбда, константа, определяющая степень сглаживания

$w(z)$: весовая функция значения пикселя z

Выходные значения:

$g(z)$: логарифмическая экспозиция, соответствующая значению пикселя z

$lE(i)$: логарифмическая освещенность в местоположении пикселя i

'''

`n = 256`

`s1, s2 = Z.shape`

`U = np.zeros((s1 * s2 + n + 1, n + s1))`

`V = np.zeros((U.shape[0], 1))`

Здесь создаются матрица U и вектор V , которые будут использоваться для решения системы линейных уравнений.

Размеры этих матриц зависят от количества пикселей в изображениях (`self.Z.shape`) и количества возможных значений пикселей (256 для 8-битного изображения).

Включение уравнений для соответствия данным

`k = 0`

`for i in range(s1):`

`for j in range(s2):`

`wij = self.w[Z[i, j]]`

`U[k, Z[i, j]] = wij`

`U[k, n + i] = -wij`

`V[k] = wij * self.Bij[i, j]`

`k += 1`

Этот двойной цикл заполняет матрицу U и вектор V .

Для каждого пикселя в каждом изображении создается уравнение, которое связывает значение пикселя с соответствующим значением времени экспозиции.

Веса wij используются для уменьшения влияния пикселей с очень высокой или очень низкой освещенностью.

Фиксация кривой путем установки ее среднего значения в ноль

`U[k, 129] = 0`

`k += 1`

Включение уравнений для плавности

`for i in range(1, n - 2):`

`U[k, i] = self.l * self.w[i + 1]`

`U[k, i + 1] = -2 * self.l * self.w[i + 1]`

`U[k, i + 2] = self.l * self.w[i + 1]`

`k += 1`

Здесь добавляются уравнения для гладкости функции отображения камеры.

Это помогает гарантировать, что функция будет изменяться плавно,

```
ЧТО ВАЖНО ДЛЯ ТОЧНОГО ВОССТАНОВЛЕНИЯ ОСВЕЩЕННОСТИ.
```

```
# Решение системы с использованием метода наименьших квадратов
M = np.dot(np.linalg.pinv(U), V)
CRF = M[0:n]
logE = M[n: len(M)]

return CRF, logE

def plot_ResponseCurves(self):
    """
    Метод plot_ResponseCurves отображает графики функций отклика для каждого цветового канала.
    Эти кривые показывают, как каждый пиксель камеры реагирует на различные уровни освещенности.

    """
    px = list(range(0, 256))
    fig = plt.figure(constrained_layout=False, figsize=(5, 5))
    plt.plot(px, np.exp(self.CRFB), 'b')
    plt.plot(px, np.exp(self.CRGF), 'g')
    plt.plot(px, np.exp(self.CRFR), 'r')
    plt.ylabel("log(CRF)", fontsize=20)
    plt.xlabel("Значение пикселя", fontsize=20)
    plt.title("Кривые отклика", fontsize=20)
    # plt.show()
    output_path = os.path.join('results', 'curvesCRF.png')
    fig.savefig(output_path)

def process(self):
    """
    Вызывает предыдущие методы для построения функции отклика и подготовки данных для восстановления карты освещенности HDR.
    """
    self.weightingFunction()
    self.samplingValues()

    with ThreadPoolExecutor() as executor:
        futures = [executor.submit(self.CRFsolve, self.ZR),
                   executor.submit(self.CRFsolve, self.ZG),
                   executor.submit(self.CRFsolve, self.ZB)]

        self.CRFB, self.lEB = futures[0].result()
        self.CRGF, self.lEG = futures[1].result()
        self.CRFR, self.lER = futures[2].result()

class PostProcess(HDR):
    """
    Класс PostProcess наследует HDR и добавляет методы для сохранения и тонового отображения HDR-изображения.
    Тоновое отображение необходимо, потому что стандартные дисплеи не могут отображать HDR напрямую.

    """
    def __init__(self, HDR):
        super().__init__(path, filenames, exposure_times)

    def save_hdr_image(self, filename='results/hdr_image.hdr'):
        """
        Метод save_hdr_image сохраняет HDR-изображение в формате, который может быть использован другими программами и устройствами, поддерживающими HDR.
        
```

```

    """
    cv2.imwrite(filename, self.imgf32)

def recover_HDR_RadianceMap(self):
    """
    Метод recover_HDR_RadianceMap восстанавливает карту освещенности HDR,
    которая представляет собой внутреннее представление освещенности сцены.
    Это позволяет сохранить всю информацию о свете в сцене, что является
    ключевым аспектом HDR-изображений.

    """
    m = np.zeros((self.flattenImage.shape[1:]))
    wsum = np.zeros(self.flattenImage.shape[1:])
    hdr = np.zeros(self.flattenImage.shape[1:])
    # Здесь создаются массивы m, wsum, и hdr для хранения промежуточных и
    # конечных результатов.
    # lnDt - это логарифм времен экспозиции, используемых для съемки
    # серии изображений.

    lnDt = np.log(self.times)

    for i in range(self.N):
        wij_B = self.w[self.flattenImage[i, 0]]
        wij_G = self.w[self.flattenImage[i, 1]]
        wij_R = self.w[self.flattenImage[i, 2]]

        wsum[0, :] += wij_B
        wsum[1, :] += wij_G
        wsum[2, :] += wij_R

        # В этом цикле для каждого изображения в серии вычисляются веса
        # wij_B, wij_G, и wij_R для каждого канала (синего, зеленого и красного
        # соответственно).
        # Эти веса используются для уменьшения влияния пикселей с очень
        # высокой или очень низкой освещенностью.

        m0 = np.subtract(self.CRFB[self.flattenImage[i, 0]], lnDt[i])[:, 0]
        m1 = np.subtract(self.CRFG[self.flattenImage[i, 1]], lnDt[i])[:, 0]
        m2 = np.subtract(self.CRFR[self.flattenImage[i, 2]], lnDt[i])[:, 0]

        hdr[0] = hdr[0] + np.multiply(m0, wij_B)
        hdr[1] = hdr[1] + np.multiply(m1, wij_G)
        hdr[2] = hdr[2] + np.multiply(m2, wij_R)

        # Здесь m0, m1, и m2 вычисляются как разность между значениями
        # функции отображения камеры (CRFB, CRFG, CRFR для каждого цветового канала)
        # и логарифмом времени экспозиции. Эти значения умножаются на
        # соответствующие веса и добавляются к hdr, что помогает восстановить карту
        # освещенности.

        hdr = np.divide(hdr, wsum)
        hdr = np.exp(hdr)
        hdr = np.reshape(np.transpose(hdr), (self.row, self.col, 3))

        # Здесь hdr делится на сумму весов wsum, чтобы нормализовать
        # результаты,
        # а затем применяется экспоненциальная функция для преобразования
        # логарифмических значений обратно в освещенность.

        self.imgf32 = (hdr / np.amax(hdr) * 255).astype(np.float32)
        self.save_hdr_image() # Сохраняем HDR-изображение перед тоновым

```

```

отображением

    fig = plt.figure(constrained_layout=False, figsize=(10, 10))
    # plt.title("Карта освещенности HDR изображения", fontsize=20)
    plt.imshow(cv2.cvtColor(self.imgf32, cv2.COLOR_BGR2RGB))
    # plt.show()
    output_path = os.path.join('results', 'radiancemap.png')
    fig.savefig(output_path)

def tone_mapping(self):
    """
    Метод tone_mapping преобразует карту освещенности HDR в формат,
    пригодный для отображения на стандартных дисплеях.
    Это включает в себя сжатие диапазона яркости и контраста, чтобы
    изображение выглядело естественно на не-HDR дисплеях.
    """
    hdr_image = cv2.imread('results/hdr_image.hdr', cv2.IMREAD_ANYDEPTH)

    # Применение тонирования для отображения на стандартном дисплее
    tonemap = cv2.createTonemap(3.0) # Гамма-коррекция
    ldr_image = tonemap.process(hdr_image)

    # Отображение изображения
    plt.imshow(cv2.cvtColor(ldr_image, cv2.COLOR_BGR2RGB))
    # plt.axis('off') # Убрать оси координат
    # plt.show()
    # Сохранение отображенного изображения
    ldr_image = cv2.normalize(ldr_image, None, alpha=0, beta=255,
    norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)
    output_path = os.path.join(app.config['RESULTS_FOLDER'],
    'tone_mapping_ldr.jpg')
    cv2.imwrite(output_path, ldr_image)

    return ldr_image

def photograph_tone_mapping(self):
    """
    Фотографический метод тонового отображения для преобразования HDR-
    изображения в LDR.
    """
    hdr_image = cv2.imread('results/hdr_image.hdr', cv2.IMREAD_ANYDEPTH)

    # Преобразование HDR-изображения в градации серого
    lum = cv2.cvtColor(hdr_image, cv2.COLOR_BGR2GRAY)

    # Вычисление глобальной средней освещенности
    Lavg = np.exp(np.mean(np.log(lum + 1e-6)))

    # Применение фотографического метода тонового отображения
    a = 0.18 # Коэффициент масштабирования
    Ld = (a / Lavg) * lum
    Ld = Ld / (1 + Ld) # Компрессия динамического диапазона

    # Применение отраженных тонов обратно к цветам
    ldr_image = np.zeros_like(hdr_image)
    for c in range(3):
        ldr_image[:, :, c] = hdr_image[:, :, c] * (Ld / lum)

    alpha = 0 # Коэффициент контраста
    beta = 400 # Яркость
    # Нормализация и преобразование в формат, подходящий для отображения
    ldr_image = cv2.normalize(ldr_image, None, alpha=alpha, beta=beta,
    norm_type=cv2.NORM_MINMAX)
    ldr_image = np.clip(ldr_image, 0, 255).astype('uint8')

```

```

# Сохранение отображенного изображения
output_path = os.path.join(app.config['RESULTS_FOLDER'],
'photo_ldr.jpg')
cv2.imwrite(output_path, ldr_image)

return ldr_image

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        try:
            exposure_times_strings = request.form.getlist('exposure_times')
            exposure_times = [float(time_str) for time_str in
exposure_times_strings]
            session['exposure_times'] = exposure_times # Сохраняем в сессии

            uploaded_files = request.files.getlist('images')
            filenames = []
            for file in uploaded_files:
                filename = secure_filename(file.filename)
                file_path = os.path.join(app.config['UPLOAD_FOLDER'],
filename)
                file.save(file_path)
                filenames.append(file_path)
            session['filenames'] = filenames # Сохраняем в сессии

        except Exception as e:
            return str(e), 400
        return redirect(url_for('process'))

    return render_template('index.html')

@app.route('/process', methods=['GET', 'POST'])
def process():
    try:
        # Извлекаем данные из сессии
        exposure_times_strings =
request.form.get('exposure_times').split(',')
        exposure_times = []
        for time_str in exposure_times_strings:
            if '/' in time_str:
                numerator, denominator = time_str.split('/')
                exposure_times.append(float(numerator) / float(denominator))
            else:
                exposure_times.append(float(time_str))

        uploaded_files = request.files.getlist('images')
        filenames = []
        images = []
        for file in uploaded_files:
            filename = secure_filename(file.filename)
            images.append(filename)
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
            file.save(file_path)
            filenames.append(file_path)

        # Обработка HDR
        mergeDebevec = HDR(app.config['UPLOAD_FOLDER'], filenames,
exposure_times)
        postProcess = PostProcess(mergeDebevec)
        postProcess.process()
        postProcess.plot_ResponseCurves()
        postProcess.recover_HDR_RadianceMap()

    
```

```
ldr_image1 = postProcess.tone_mapping()
ldr_image2 = postProcess.photograph_tone_mapping()

# Сохранение и отправка результата
output_path = os.path.join(app.config['RESULTS_FOLDER'],
'tone_mapping_ldr.jpg')
cv2.imwrite(output_path, ldr_image1)
# Сохранение и отправка результата
output_path = os.path.join(app.config['RESULTS_FOLDER'],
'photo_ldr.jpg')
cv2.imwrite(output_path, ldr_image2)
return render_template('process.html')
except Exception as e:
    # Обработка исключений
    print(e)
    return str(e), 500

# Если ни один из вышеуказанных путей не выполнен
return render_template('process.html', {
    'filenames': images
})
)

@app.route('/results/<filename>')
def results(filename):
    return send_from_directory('results', filename)

@app.route('/uploads/<filename>')
def uploads(filename):
    return send_from_directory('uploads', filename)

if __name__ == '__main__':
    app.run()
```