CS445: Computational Photography

Programming Project 4: Image-Based Lighting

Recovering HDR Radiance Maps

Load libraries and data

In [1]:
```
# System imports
from os import path
import math

# Third-Party Imports
import cv2
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import griddata

# import starter code
import utils
from utils.io import read_image, write_image, read_hdr_image, write_hdr_image
from utils.display import display_images_linear_rescale, rescale_images_linear
from utils.hdr_helpers import gsolve
from utils.hdr_helpers import get_equirectangular_image
from utils.bilateral_filter import bilateral_filter
```

Reading LDR images

You can use the provided samples or your own images. You get more points for using your own images, but it might help to get things working first with the provided samples.

In [2]:
```
# TODO: Replace this with your path and files

# imdir = 'samples'
# imfns = ['0024.jpg', '0060.jpg', '0120.jpg', '0205.jpg', '0553.jpg']
# exposure_times = [1/24.0, 1/60.0, 1/120.0, 1/205.0, 1/553.0]

imdir = 'data'
imfns = ['0025.jpg', '0050.jpg', '0125.jpg', '0200.jpg', '0500.jpg']
exposure_times = [1/25.0, 1/50.0, 1/125.0, 1/200.0, 1/500.0]

ldr_images = []
for f in np.arange(len(imfns)):
  im = read_image(imdir + '/' + imfns[f])
  if f==0:
    imsize = int((im.shape[0] + im.shape[1])/2) # set width/height of ball images
    ldr_images = np.zeros((len(imfns), imsize, imsize, 3))
  ldr_images[f] = cv2.resize(im, (imsize, imsize))

background_image_file = imdir + '/' + 'background.jpg'
background_image = read_image(background_image_file)
```

Naive LDR merging

Compute the HDR image as average of irradiance estimates from LDR images

In [3]:
```python
def make_hdr_naive(ldr_images: np.ndarray, exposures: list) -> (np.ndarray, np.ndarray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding exposure values.

    The steps to implement:
    1) Divide each image by its exposure time.
        - This will rescale images as if it has been exposed for 1 second.

    2) Return average of above images


    For further explanation, please refer to problem page for how to do it.

    Args:
        ldr_images(np.ndarray): N x H x W x 3  shaped numpy array representing
            N ldr images with width W, height H, and channel size of 3 (RGB)
        exposures(list): list of length N, representing exposures of each images.
            Each exposure should correspond to LDR images' exposure value.
    Return:
        (np.ndarray): H x W x 3 shaped numpy array representing HDR image merged using
            naive ldr merging implementation.
        (np.ndarray): N x H x W x 3  shaped numpy array represending log irradiances
            for each exposures

    '''
    N, H, W, C = ldr_images.shape
    # sanity check
    assert N == len(exposures)

    hdr_image = np.zeros((H, W, C))
    log_irradiances = np.zeros((N, H, W, C))
    for i in range(N):
        hdr_image += ldr_images[i] / exposures[i]
        log_irradiances[i] = np.log((ldr_images[i] + 0.1) / exposures[i])

    hdr_image /= N

    return hdr_image, log_irradiances
```

In [4]:
```python
def display_hdr_image(im_hdr):
    '''
    Maps the HDR intensities into a 0 to 1 range and then displays.
    Three suggestions to try:
      (1) Take log and then linearly map to 0 to 1 range (see display.py for example)
      (2) img_out = im_hdr / (1 + im_hdr)
      (3) HDR display code in a python package
    '''

    # Tonemap HDR image for display
    tonemap = cv2.createTonemapDrago(gamma=1.0, saturation=0.7, bias=0.85)
    res = tonemap.process(im_hdr.copy().astype(np.float32))
    plt.axis('off')
    plt.imshow((res*255).astype(np.uint8))
```

In [5]:
```python
# get HDR image, log irradiance
naive_hdr_image, naive_log_irradiances = make_hdr_naive(ldr_images, exposure_times)

# write HDR image to directory
write_hdr_image(naive_hdr_image, 'images/outputs/naive_hdr.hdr')

# display HDR image
print('HDR Image')
display_hdr_image(naive_hdr_image)
```

```
# display original images (code provided in utils.display)
display_images_linear_rescale(ldr_images)

# display log irradiance image (code provided in utils.display)
display_images_linear_rescale(naive_log_irradiances)
```

```
HDR Image
[ WARN:0@1.232] global matrix_expressions.cpp:1333 assign OpenCV/MatExpr: processing of multi-channel arrays might be changed in the future: https://github.com/opencv/opencv/issues/16739
/tmp/ipykernel_3421318/1536780538.py:14: RuntimeWarning: invalid value encountered in cast
  plt.imshow((res*255).astype(np.uint8))
```

In [6]:
```
# find the index that log irradiance is negative or zero
print(naive_log_irradiances.shape)
for i in range(5):
    print(np.where(naive_log_irradiances[i] <= 0))
```

```
(5, 1200, 1200, 3)
(array([], dtype=int64), array([], dtype=int64), array([], dtype=int64))
(array([], dtype=int64), array([], dtype=int64), array([], dtype=int64))
(array([], dtype=int64), array([], dtype=int64), array([], dtype=int64))
```

```
(array([], dtype=int64), array([], dtype=int64), array([], dtype=int64))
(array([], dtype=int64), array([], dtype=int64), array([], dtype=int64))
```

Weighted LDR merging

Compute HDR image as a weighted average of irradiance estimates from LDR images, where weight is based on pixel intensity so that very low/high intensities get less weight

In [7]:
```python
def make_hdr_weighted(ldr_images: np.ndarray, exposure_times: list) -> (np.ndarray, np.n
    '''
    Makes HDR image using multiple LDR images, and its corresponding exposure values.

    The steps to implement:
    1) compute weights for images with based on intensities for each exposures
        - This can be a binary mask to exclude low / high intensity values

    2) Divide each images by its exposure time.
        - This will rescale images as if it has been exposed for 1 second.

    3) Return weighted average of above images


    Args:
        ldr_images(np.ndarray): N x H x W x 3 shaped numpy array representing
            N ldr images with width W, height H, and channel size of 3 (RGB)
        exposure_times(list): list of length N, representing exposures of each images.
            Each exposure should correspond to LDR images' exposure value.
    Return:
        (np.ndarray): H x W x 3 shaped numpy array representing HDR image merged without
            under - over exposed regions

    '''
    N, H, W, C = ldr_images.shape
    # sanity check
    assert N == len(exposure_times)

    # weighting function
    def weighting(z):
        return (128 - abs(z * 255 - 128))

    weighted_hdr_image = np.zeros((H, W, C))
    total_weight = 0
    for i in range(N):
        weights = weighting(ldr_images[i])
        total_weight += weights
        weighted_hdr_image += ldr_images[i] * weights / exposure_times[i]

    weighted_hdr_image /= total_weight

    return weighted_hdr_image
```

In [8]:
```python
# get HDR image, log irradiance
weighted_hdr_image = make_hdr_weighted(ldr_images, exposure_times)

# write HDR image to directory
write_hdr_image(weighted_hdr_image, 'images/outputs/weighted_hdr.hdr')

# display HDR image
display_hdr_image(weighted_hdr_image)
```

```
/tmp/ipykernel_3421318/1536780538.py:14: RuntimeWarning: invalid value encountered in ca
```

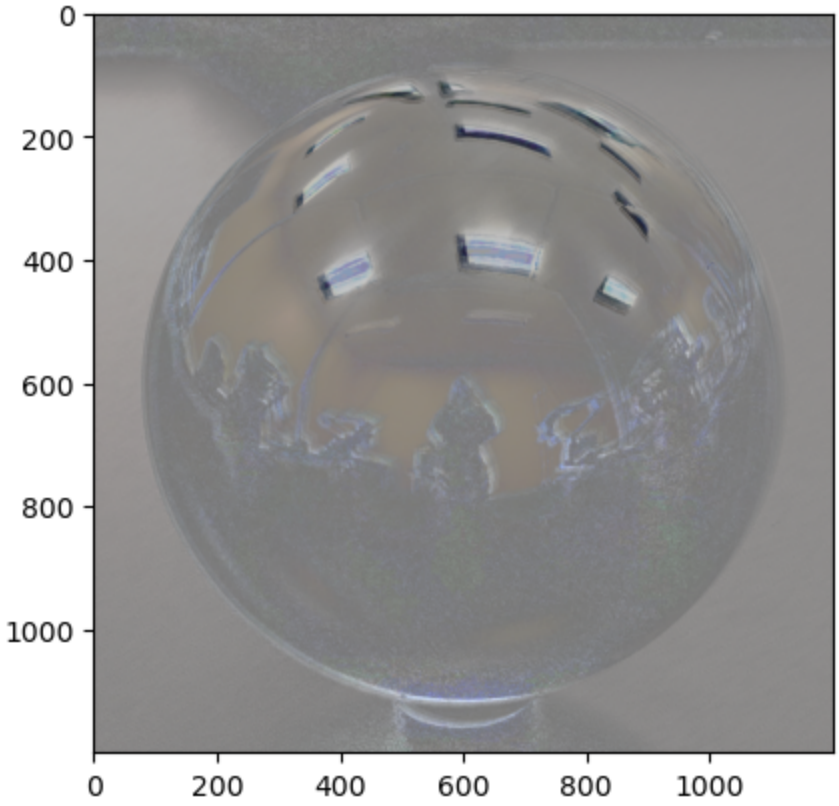
```
st
  plt.imshow((res*255).astype(np.uint8))
```

Display of difference between naive and weighted for your own inspection

Where does the weighting make a big difference increasing or decreasing the irradiance estimate? Think about why.

In [9]:
```
# display difference between naive and weighted

log_diff_im = np.log(weighted_hdr_image)-np.log(naive_hdr_image)
print('Min ratio = ', np.exp(log_diff_im).min(), '  Max ratio = ', np.exp(log_diff_im).m
plt.figure()
plt.imshow(rescale_images_linear(log_diff_im))
```

Out[9]:
```
Min ratio =  0.16029762542515003   Max ratio =  5.000000000000001
<matplotlib.image.AxesImage at 0x7f0c2b8f3130>
```



LDR merging with camera response function estimation

Compute HDR after calibrating the photometric reponses to obtain more accurate irradiance estimates from each image

Some suggestions on using `gsolve`:

- When providing input to gsolve, don't use all available pixels, otherwise you will likely run out of memory / have very slow run times. To overcome, just randomly sample a set of pixels (1000 or so can suffice), but make sure all pixel locations are the same for each exposure.
- The weighting function w should be implemented using Eq. 4 from the paper (this is the same function that can be used for the previous LDR merging method).
- Try different lambda values for recovering *g*. Try lambda=1 initially, then solve for *g* and plot it. It should be smooth and continuously increasing. If lambda is too small, g will be bumpy.
- Refer to Eq. 6 in the paper for using g and combining all of your exposures into a final image. Note that this produces log irradiance values, so make sure to exponentiate the result and save irradiance in linear scale.

In [10]:
```
def make_hdr_estimation(ldr_images: np.ndarray, exposure_times: list, lm)-> (np.ndarray,
    '''
    Makes HDR image using multiple LDR images, and its corresponding exposure values.
    Please refer to problem notebook for how to do it.

    **IMPORTANT**
    The gsolve operations should be ran with:
        Z: int64 array of shape N x P, where N = number of images, P = number of pixels
        B: float32 array of shape N, log shutter times
        l: lambda; float to control amount of smoothing
        w: function that maps from float intensity to weight
    The steps to implement:
    1) Create random points to sample (from mirror ball region)
    2) For each exposures, compute g values using samples
```

```
    3) Recover HDR image using g values


    Args:
        ldr_images(np.ndarray): N x H x W x 3 shaped numpy array representing
            N ldr images with width W, height H, and channel size of 3 (RGB)
        exposures(list): list of length N, representing exposures of each images.
            Each exposure should correspond to LDR images' exposure value.
        lm (scalar): the smoothing parameter
    Return:
        (np.ndarray): H x W x 3 shaped numpy array representing HDR image merged using
            gsolve
        (np.ndarray): N x H x W x 3 shaped numpy array represending log irradiances
            for each exposures
        (np.ndarray): 3 x 256 shaped numpy array represending g values of each pixel int
            at each channels (used for plotting)
    '''
    N, H, W, C = ldr_images.shape
    # sanity check
    assert N == len(exposure_times)

    # TO DO: implement HDR estimation using gsolve
    # gsolve(Z, B, l, w) -> g, lE

    np.random.seed(0) # set random seed for reproducibility

    hdr_image = np.zeros((H, W, C))
    log_irradiances = np.zeros((N, H, W, C))
    g = np.zeros((C, 256))

    # sample points for gsolve
    randsize = 1000
    randx = np.random.randint(0, W, randsize)
    randy = np.random.randint(0, H, randsize)

    # inputs for gsolve
    Z = np.zeros((N, randsize)) # pixel intensity values
    B = np.log(exposure_times).astype(np.float32) # log shutter times
    l = lm # the smoothing parameter
    w = lambda z: (128 - abs(z - 128)) # weighting function

    # estimate g for each RGB channel
    for c in range(C):
        for n in range(N):
            for i in range(randsize):
                Z[n, i] = ldr_images[n, randy[i], randx[i], c] * 255
        g[c], _ = gsolve(Z.astype(np.int64), B, l, w)

    # calculate hdr_image and log_irradiances
    total_weight = np.zeros((H, W, C))
    for c in range(C):
        for n in range(N):
            p = ldr_images[n, :, :, c] * 255
            p = p.astype(np.int64)
            hdr_image[:, :, c] += (g[c, p] - B[n]) * w(p)
            log_irradiances[n, :, :, c] = g[c, p] - B[n]
            total_weight[:, :, c] += w(p)
        hdr_image[:, :, c] /= total_weight[:, :, c]

    hdr_image = np.exp(hdr_image)

    return hdr_image, log_irradiances, g
```

In [11]:

```
lm = 5
# get HDR image, log irradiance
```

```
calib_hdr_image, calib_log_irradiances, g = make_hdr_estimation(ldr_images, exposure_tim
# sanity check
plt.imshow(g)
```

Out[11]:

```
<matplotlib.image.AxesImage at 0x7f0c2b8d6e80>
```

In [12]:

```
lm = 10
# get HDR image, log irradiance
calib_hdr_image, calib_log_irradiances, g = make_hdr_estimation(ldr_images, exposure_tim

# write HDR image to directory
write_hdr_image(calib_hdr_image, 'images/outputs/calib_hdr.hdr')

# display HDR image
display_hdr_image(calib_hdr_image)
```

```
/tmp/ipykernel_3421318/1536780538.py:14: RuntimeWarning: invalid value encountered in ca
st
  plt.imshow((res*255).astype(np.uint8))
```

The following code displays your results. You can copy the resulting images and plots directly into your report where appropriate.

In [21]:

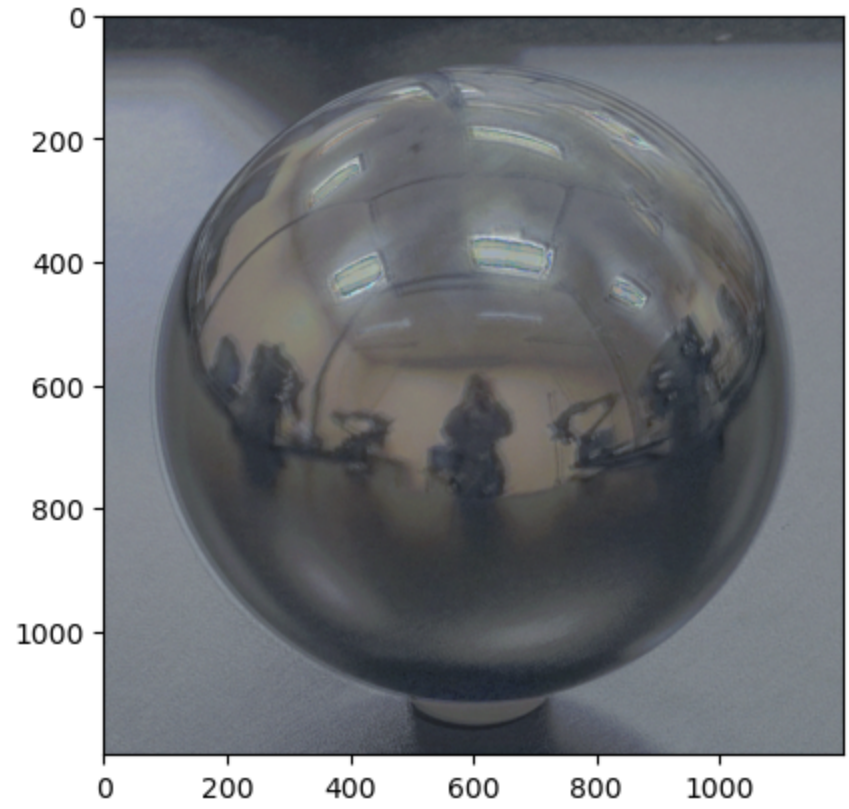
```
# display difference between calibrated and weighted
log_diff_im = np.log(calib_hdr_image/calib_hdr_image.mean())-np.log(weighted_hdr_image/w
print('Min ratio = ', np.exp(log_diff_im).min(), '  Max ratio = ', np.exp(log_diff_im).m
plt.figure()
plt.imshow(rescale_images_linear(log_diff_im))

# display original images (code provided in utils.display)
display_images_linear_rescale(ldr_images)

# display log irradiance image (code provided in utils.display)
display_images_linear_rescale(calib_log_irradiances)

# plot g vs intensity, and then plot intensity vs g
```

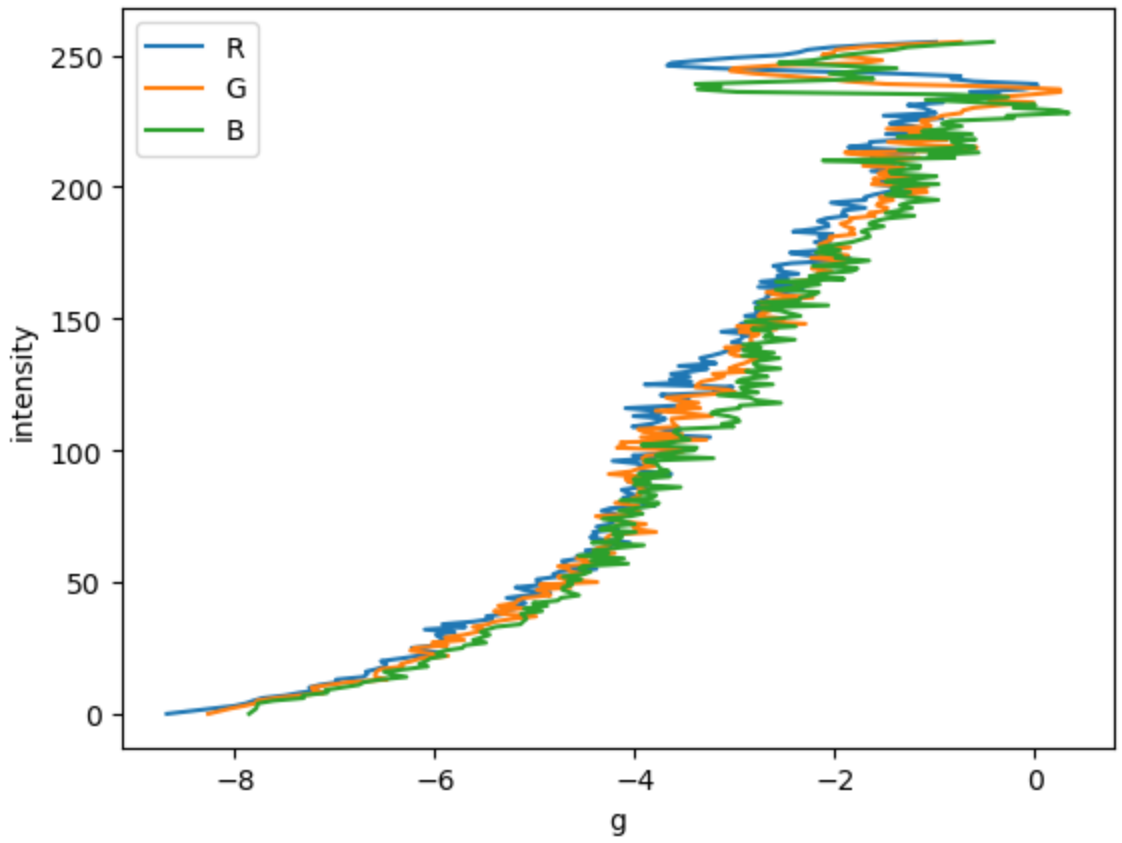
```
N, NG = g.shape
labels = ['R', 'G', 'B']
plt.figure()
for n in range(N):
    plt.plot(g[n], range(NG), label=labels[n])
plt.gca().legend(('R', 'G', 'B'))
plt.xlabel('g')
plt.ylabel('intensity')

plt.figure()
for n in range(N):
    plt.plot(range(NG), g[n], label=labels[n])
plt.gca().legend(('R', 'G', 'B'))
plt.xlabel('intensity')
plt.ylabel('g')
```

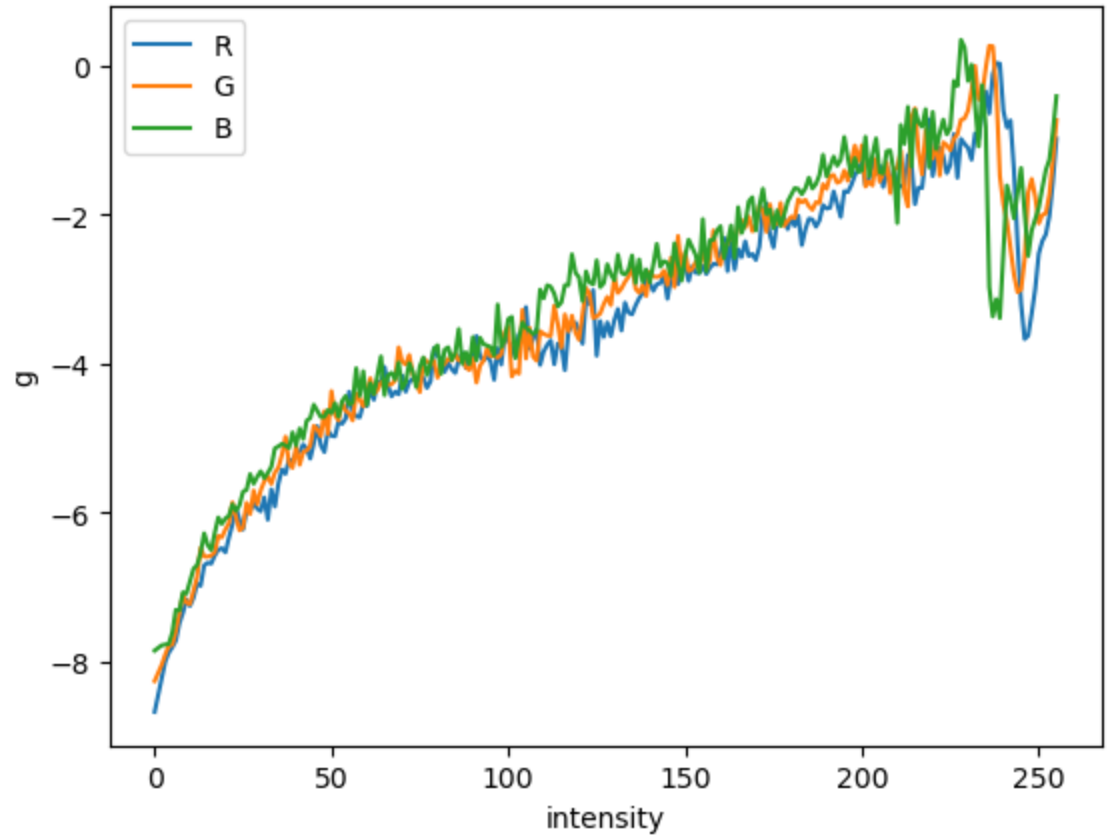
```
Min ratio =  0.07149055001942646   Max ratio =  11.978758689366833
```

Out[21]:

```
Text(0, 0.5, 'g')
```







In [14]:
```
def weighted_log_error(ldr_images, hdr_image, log_irradiances):
  # computes weighted RMS error of log irradiances for each image compared to final log
  N, H, W, C = ldr_images.shape
  w = 1-abs(ldr_images - 0.5)*2
  err = 0
  for n in np.arange(N):
    err += np.sqrt(np.multiply(w[n], (log_irradiances[n]-np.log(hdr_image))**2).sum()/w[
  return err
```

```
# compare solutions
err = weighted_log_error(ldr_images, naive_hdr_image, naive_log_irradiances)
print('naive:  \tlog range = ', round(np.log(naive_hdr_image).max() - np.log(naive_hdr_i
err = weighted_log_error(ldr_images, weighted_hdr_image, naive_log_irradiances)
print('weighted:\tlog range = ', round(np.log(weighted_hdr_image).max() - np.log(weighte
err = weighted_log_error(ldr_images, calib_hdr_image, calib_log_irradiances)
print('calibrated:\tlog range = ', round(np.log(calib_hdr_image).max() - np.log(calib_hd

# display log hdr images (code provided in utils.display)
display_images_linear_rescale(np.log(np.stack((naive_hdr_image/naive_hdr_image.mean(), w
```

```
naive:          log range =  9.125      avg RMS error =  0.76
weighted:       log range =  8.324      avg RMS error =  0.764
calibrated:     log range =  10.823     avg RMS error =  0.549
```



Panoramic transformations

Compute the equirectangular image from the mirrorball image

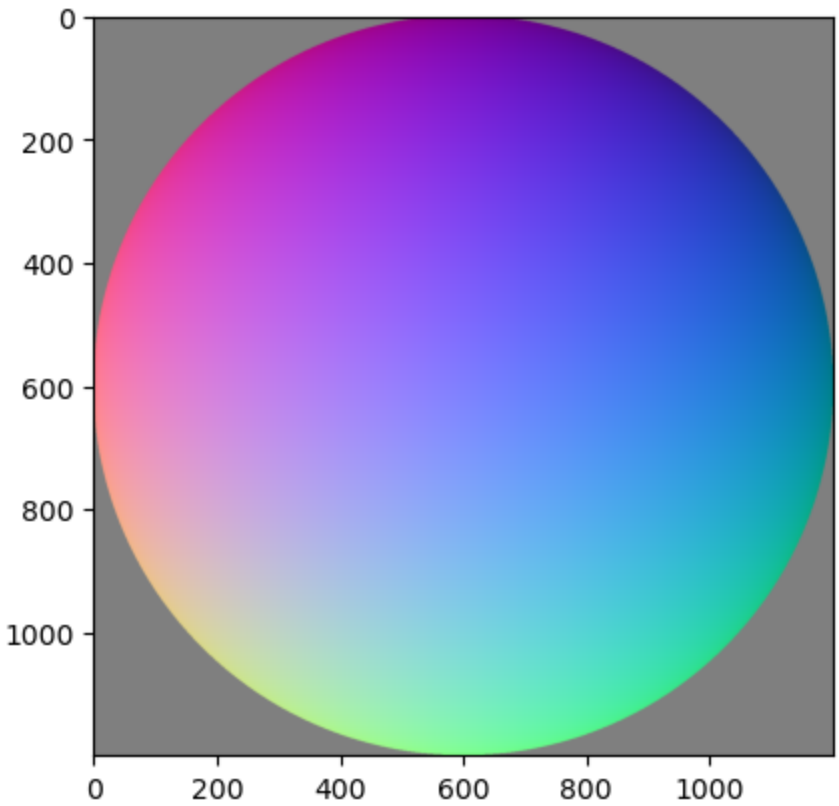
In [15]:

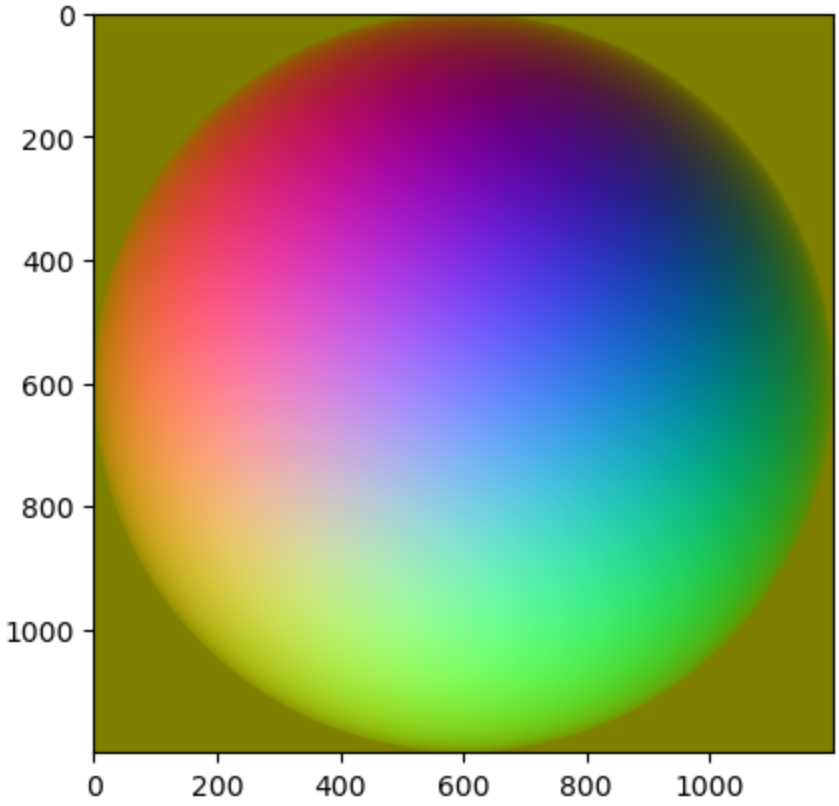
```
def panoramic_transform(hdr_image):
    '''
    Given HDR mirror ball image,

    Expects mirror ball image to have center of the ball at center of the image, and
    width and height of the image to be equal.

    Steps to implement:
    1) Compute N image of normal vectors of mirror ball
    2) Compute R image of reflection vectors of mirror ball
    3) Map reflection vectors into spherical coordinates
    4) Interpolate spherical coordinate values into equirectangular grid.

    Steps 3 and 4 are implemented for you with get_equirectangular_image

    '''
    H, W, C = hdr_image.shape
    assert H == W
    assert C == 3

    # TO DO: compute N and R
    N = np.zeros((H, W, C)) # normal vector
    V = np.zeros((H, W, C)) # viewing direction
    R = np.zeros((H, W, C)) # reflection vector

    for y in range(H):
        for x in range(W):
            # normalized pixel coordinates
            nx = -(x - W / 2) / (W / 2)
            ny = (y - H / 2) / (H / 2)
            nz = np.sqrt(1 - np.clip(nx ** 2 + ny ** 2, 0, 1))
```

```
            N[y, x] = np.array([nx, ny, nz])
            # V[y, x] = -N[y, x] # viewing direction is opposite to normal vector
            V[y, x, 2] = -1

    # mask the pixels that are on the mirror ball
    mask = np.where(N[:, :, 2] > 0, 1, 0)
    for i in range(2):
        N[:, :, i] *= mask

    # reflection vector
    R  = V - 2 * (V * N) * N
    for i in range(3):
        R[:, :, i] = V[:, :, i] + 2 * N[:, :, 2] * N[:, :, i]

    plt.imshow((N+1)/2)
    plt.show()
    plt.imshow((R+1)/2)
    plt.show()

    equirectangular_image = get_equirectangular_image(R, hdr_image)
    return equirectangular_image
```

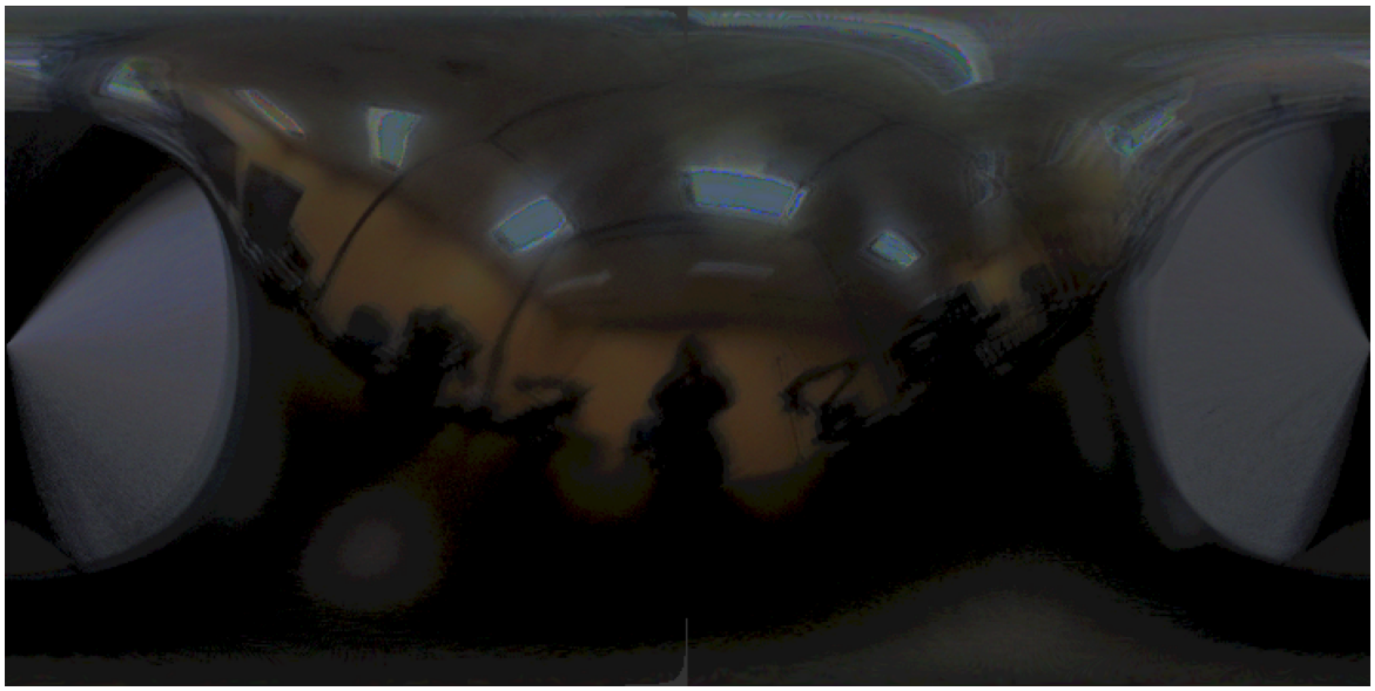
In [16]:

```
hdr_mirrorball_image = read_hdr_image('images/outputs/calib_hdr.hdr')
eq_image = panoramic_transform(hdr_mirrorball_image)

write_hdr_image(eq_image, 'images/outputs/equirectangular.hdr')

plt.figure(figsize=(15,15))
display_hdr_image(eq_image)
```






```
/tmp/ipykernel_3421318/1536780538.py:14: RuntimeWarning: invalid value encountered in ca
st
  plt.imshow((res*255).astype(np.uint8))
```



Rendering synthetic objects into photographs

Use Blender to render the scene with and with objects and obtain the mask image. The code below should then load the images and create the final composite.

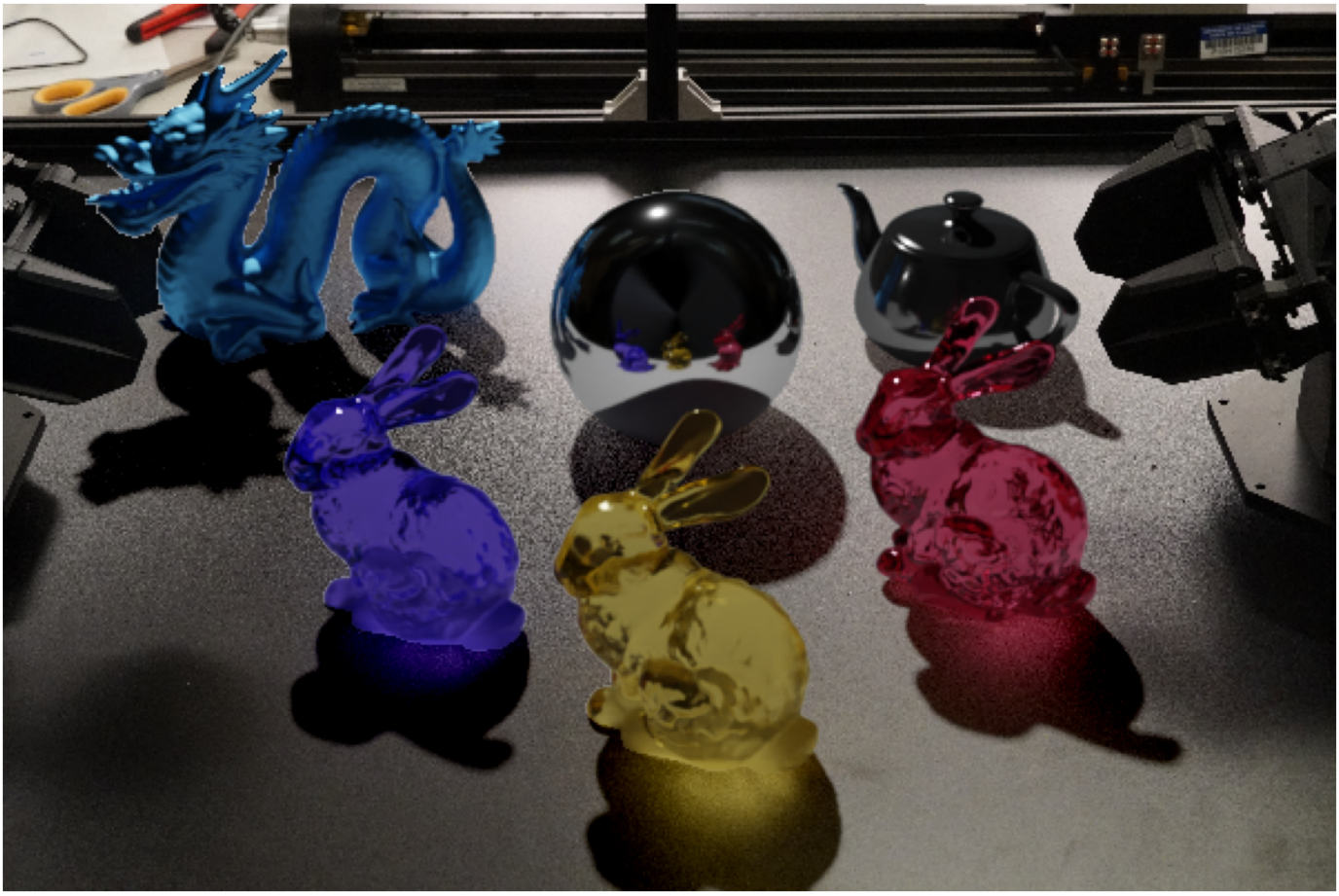
In [18]:
```
# Read the images that you produced using Blender.  Modify names as needed.
O = read_image('images/proj4_objects.png')
E = read_image('images/proj4_empty.png')
```

```
M = read_image('images/proj4_mask.png')
M = M > 0.5
I = background_image
I = cv2.resize(I, (M.shape[1], M.shape[0]))
```

In [20]:

```
# TO DO: compute final composite
# result = []
c = 1
result = M*O + (1-M)*I + (1-M)*(O-E)*c

plt.figure(figsize=(20,20))
plt.axis('off')
plt.imshow(result)
plt.show()

write_image(result, 'images/outputs/final_composite.png')
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
```



Bells & Whistles (Extra Points)

Additional Image-Based Lighting Result

Other panoramic transformations

Photographer/tripod removal

Local tonemapping operator

In [ ]: