CS445: Computational Photography

Programming Project 4: Image-Based Lighting

Due Date: 11:59pm on Tuesday, Oct. 29, 2019

This is a template solution file. Please feel free to use this for the base of your report.

Recovering HDR Radiance Maps (40 pts)

We start by loading in necessary libraries used for this section of the problem.

In [1]:

```
# jupyter extension that allows reloading functions from imports without clearin
g kernel :D
%load_ext autoreload
%autoreload 2
```

In [2]:

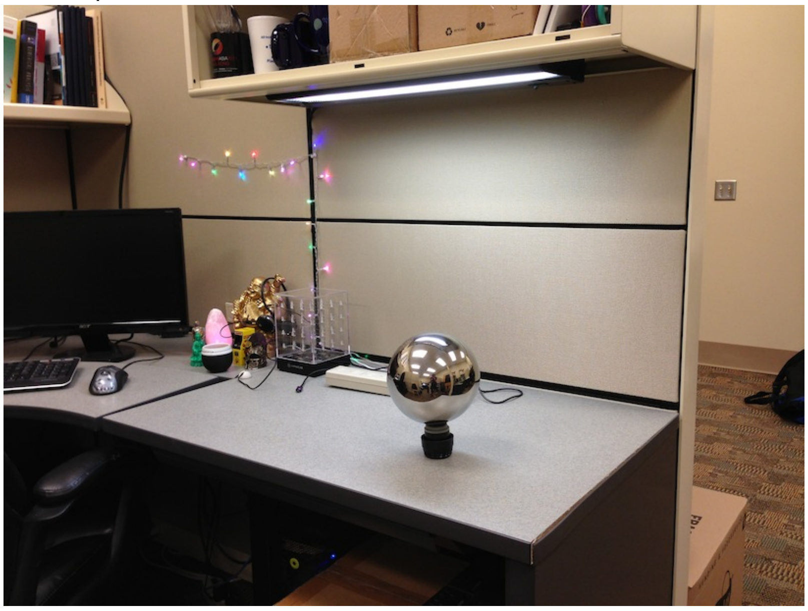
```
# System imports
from os import path
import math
import os

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

# local imports
from utils.io import read_image, write_image, read_hdr_image, write_hdr_image
from utils.display import display_hdr_image_linear, display_hdr_image, display_l
og_irradiances
from utils.hdr_helpers import gsolve
from utils.bilateral_filter import bilateral_filter
from utils.meta import TODO
```

In [3]:

```
def plot_no_frame(img, cmap=None, title=None):
    fig = plt.figure(frameon=False)
    ax = plt.Axes(fig, [0., 0., 1., 1.])

    if cmap is not None:
        ax.imshow(img, cmap=cmap)
    else:
        ax.imshow(img)

    ax.set_axis_off()
    fig.add_axes(ax)

    dpi = fig.get_dpi()
    fig.set_size_inches(img.shape[1]/float(dpi), img.shape[0]/float(dpi))
    return fig
```

Data collection (10 points)

1. Find a good scene to photograph. The scene should have a flat surface to place your spherical mirror on. Either indoors or outdoors will work.

2. Find a fixed, rigid spot to place your camera. A tripod is best, but you can get away with less. I used the back of a chair to steady my phone when taking my images.
3. Place your spherical mirror on a flat surface, and make sure it doesn't roll by placing a cloth/bottle cap/etc under it. Make sure the sphere is not too far away from the camera -- it should occupy at least a 256x256 block of pixels.

4. Photograph the spherical mirror using at least three different exposure times. Make sure the camera does not move too much (slight variations are OK, but the viewpoint should generally be fixed). For best results, your exposure times should be at least 4 times longer and 4 times shorter (±2 stops) than your mid-level exposure (e.g. if your mid-level exposure time is 1/40s, then you should have at least exposure timess of 1/10s and 1/160s; the greater the range the better). Make sure to record the exposure times.
5. Remove the mirror from the scene, and from the same viewpoint as the other photographs, take another picture of the scene at a normal exposure level (most pixels are neither over- or under-exposed). This will be the image that you will use for object insertion/compositing (the "background" image).
6. After you copy all of the images from your camera/phone to your computer, load the spherical mirror images (from step 4) into your favorite image editor and crop them down to contain only the

sphere.

7. Small alignment errors may occur (due to camera motion or cropping). One way to fix these is through various alignment procedures, but for this project, we won't worry about these errors. If there are substantial differences in camera position/rotation among the set of images, re-take the photographs.

In [4]:

```
# TODO: Replace this with actual file path!
low_exposure_mirror_ball_image_file = "samples/1_160_cut.png"
mid_exposure_mirror_ball_image_file = "samples/1_40_cut.png"
high_exposure_mirror_ball_image_file = "samples/1_5_cut.png"
background_image_file = "samples/my_empty.jpg"

# low_exposure_mirror_ball_image_file = "samples/0553.jpg"
# mid_exposure_mirror_ball_image_file = "samples/0120.jpg"
# high_exposure_mirror_ball_image_file = "samples/0024.jpg"
# background_image_file = "samples/empty.jpg"


# TODO: Extract exposure values for each images
low_exposure = 1 / 160
mid_exposure = 1 / 40
high_exposure = 1 / 5

# low_exposure = 1 / 553
# mid_exposure = 1 / 120
# high_exposure = 1 / 24

# These images will be used for LDR mergings
low_exposure_mirror_ball_image = read_image(low_exposure_mirror_ball_image_file)
mid_exposure_mirror_ball_image = read_image(mid_exposure_mirror_ball_image_file)
high_exposure_mirror_ball_image = read_image(high_exposure_mirror_ball_image_fil
e)
background_image = read_image(background_image_file)
```

In [5]:

```
# resize mirror ball images:
# TODO: change size of N to your favorite value!
N = 256
low_exposure_mirror_ball_image = cv2.resize(low_exposure_mirror_ball_image, (N,
N))
mid_exposure_mirror_ball_image = cv2.resize(mid_exposure_mirror_ball_image, (N,
N))
high_exposure_mirror_ball_image = cv2.resize(high_exposure_mirror_ball_image, (N
, N))


ldr_images = np.stack((low_exposure_mirror_ball_image,
                       mid_exposure_mirror_ball_image,
                       high_exposure_mirror_ball_image))
exposures = [low_exposure, mid_exposure, high_exposure]
```

Naive LDR merging (10 points)

After collecting data, load the cropped images, and resize them to all be square and the same dimensions (e.g. cv2.resize(ldr,(N,N)) N is the new size). Either find the exposure times using the EXIF data (usually accessible in the image properties, or via matlab's imfinfo), or refer to your recorded exposure times. To put the images in the same intensity domain, divide each by its exposure time (e.g. ldr1_scaled = ldr1 / exposure_time1). After this conversion, all pixels will be scaled to their approximate value if they had been exposed for 1s.

The easiest way to convert your scaled LDR images to an HDR is simply to average them. Create one of these for comparison to your later results.

To save the HDR image, use given write_hdr_image function. To visualize HDR image, use given display_hdr_image function.

In [6]:

```
def save_images(hdr_image, log_irradiances, prefix):
    hdr_path = os.path.join('images/outputs/', f'{prefix}_hdr.png')
    hdr_copy = np.array(hdr_image).astype(np.float32)
    hdr_copy = (hdr_copy - np.min(hdr_copy)) / (np.max(hdr_copy) - np.min(hdr_co
py))
    hdr_fig = plot_no_frame(hdr_copy)
    hdr_fig.savefig(hdr_path)
    plt.close(hdr_fig)

    log_irrd_copy = np.array(log_irradiances).astype(np.float32)
    log_irrd_min, log_irrd_max = np.min(log_irrd_copy), np.max(log_irrd_copy)
    log_irrd_copy = (log_irrd_copy - log_irrd_min) / (log_irrd_max - log_irrd_mi
n)
    for i in range(3):
        fig = plot_no_frame(log_irrd_copy[i])
        path = os.path.join('images/outputs/', f'{prefix}_{i}')
        fig.savefig(path)
        plt.close(fig)
```

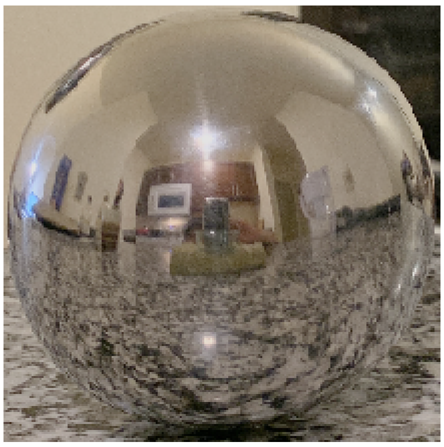
In [7]:

```
def make_hdr_naive(ldr_images: np.ndarray, exposures: list) -> (np.ndarray, np.n
darray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding exposure va
lues.

    The steps to implement:
    1) Divide each images 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 image
s.
            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 irradian
ces
            for each exposures

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

    # TODO: Implement ldr_images + exposures -> HDR image function here
    # np_exposures = N, shaped array
    hdr_image = np.zeros_like(ldr_images[0])
    log_irradiances = []
    for ldr, t in zip(ldr_images, exposures):
        irrd = ldr / t
        log_irradiances.append(np.log(irrd + 1e-6))
        hdr_image += irrd
    hdr_image /= N

    log_irradiances = np.asarray(log_irradiances)
    return hdr_image, log_irradiances
```

In [8]:

```
# get HDR image, log irradiance
naive_hdr_image, naive_log_irradiances = make_hdr_naive(ldr_images, exposures)

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

# display HDR image
_ = display_hdr_image(naive_hdr_image)
plt.show()

# display log irradiance image
_ = display_log_irradiances(naive_log_irradiances)
plt.show()

save_images(naive_hdr_image, naive_log_irradiances, "naive")
```

```
Warning: Negative / Inf values found in hdr image. Clamping to neare
st valid value
```



LDR merging without under- and over-exposed regions (5 points)

The naive method has an obvious limitation: if any pixels are under- or over-exposed, the result will contain clipped (and thus incorrect) information. A simple fix is to find these regions (e.g. a pixel might be considered over exposed if its value is less than 0.02 or greater than 0.98, assuming [0,1] images), and exclude them from the averaging process. Another way to think about this is that the naive method is extended using a weighted averaging procedure, where weights are 0 if the pixel is over/under-exposed, and 1 otherwise. Note that with this method, it might be the case that for a given pixel it is never properly exposed (i.e. always either above or below the threshold in each exposure).

There are perhaps better methods that achieve similar results but don't require a binary weighting. For example, we could create a weighting function that is small if the input (pixel value) is small or large, and large otherwise, and use this to produce an HDR image. In python, such a function can be created with:

```
w = lambda z: float(128-abs(z-128))
```

assuming pixel values range in [0,255].

In [9]:

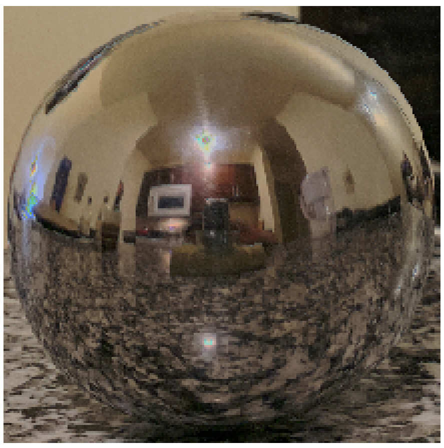
```
def make_hdr_filtered(ldr_images: np.ndarray, exposures: list) -> (np.ndarray, n
p.ndarray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding exposure va
lues.
    Please refer to problem notebook for how to do it.

    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)
        exposures(list): list of length N, representing exposures of each image
s.
            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
        (np.ndarray): N x H x W x 3 shaped numpy array represending log irradian
ces
            for each exposures
    '''
    ldrs = (ldr_images * 255).astype(np.int64)

    hdr_image = np.zeros_like(ldr_images[0])
    log_irradiances = []
    masks = []

    for ldr, t in zip(ldrs, exposures):
        mask = (128-abs(ldr-128))

        irrd = ldr / t
        log_irradiances.append(np.log((irrd + 1e-6)))
        hdr_image += irrd * mask

        masks.append(mask)

    hdr_image /= np.sum(masks, axis=0)
    hdr_image = np.nan_to_num(hdr_image)

    log_irradiances = np.asarray(log_irradiances)
    return hdr_image, log_irradiances
```

In [10]:

```
# get HDR image, log irradiance
filtered_hdr_image, filtered_log_irradiances = make_hdr_filtered(ldr_images, exp
osures)

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

# display HDR image
_ = display_hdr_image(filtered_hdr_image)
plt.show()

# display log irradiance image
_ = display_log_irradiances(filtered_log_irradiances)
plt.show()

save_images(filtered_hdr_image, filtered_log_irradiances, "filtered")
```

```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:42: Run
timeWarning: invalid value encountered in true_divide
```

```
Warning: Negative / Inf values found in hdr image. Clamping to neare
st valid value
```



LDR merging and response function estimation (15 points)

Nearly all cameras apply a non-linear function to recorded raw pixel values in order to better simulate human vision. In other words, the light incoming to the camera (radiance) is recorded by the sensor, and then mapped to a new value by this function. This function is called the film response function, and in order to convert pixel values to true radiance values, we need to estimate this response function. Typically the response function is hard to estimate, but since we have multiple observations at each pixel at different exposures, we can do a reasonable job up to a missing constant.

The method we will use to estimate the response function is outlined in this paper. Given pixel values Z at varying exposure times t, the goal is to solve for g(Z) = ln(R*t) = ln(R)+ln(t). This boils down to solving for R (irradiance) since all other variables are known. By these definitions, g is the inverse, log response function. The paper provides code to solve for g given a set of pixels at varying exposures (we also provide gsolve in utils). Use this code to estimate g for each image channel (r/g/b). Then, recover the HDR image using equation 6 in the paper.

Some hints 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 (100 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, i.e. w = lambda z: float(128-abs(z-128))).
- 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 radiance values, so make sure to exponentiate the result and save absolute radiance.

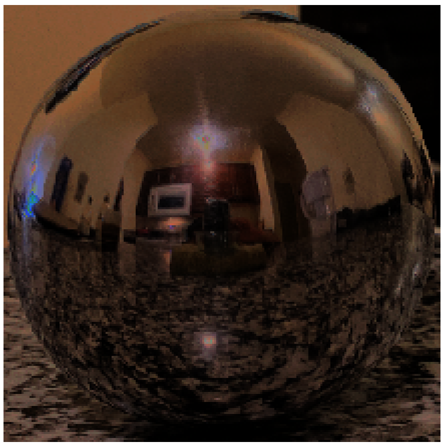
In [11]:

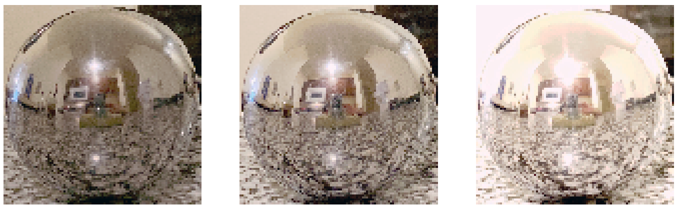
```
def make_hdr_estimation(ldr_images: np.ndarray, exposures: list,
                        n_point=500, lamb=30)-> (np.ndarray, np.ndarray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding exposure va
lues.
    Please refer to problem notebook for how to do it.

    **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,
        l: Number
        W: function that takes int and returns float

    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 image
s.
            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
            gsolve
        (np.ndarray): N x H x W x 3 shaped numpy array represending log irradian
ces
            for each exposures
        (np.ndarray): N x 256 shaped numpy array represending g values of each p
ixel intensities
            at each exposures (used for plotting)
    '''
    ldrs = (ldr_images * 255).astype(np.int64)

    N, H, W, C = ldrs.shape
    center_h, center_w = H // 2, W // 2
    R = H // 2
    # sanity check
    assert N == len(exposures)

    # Implement HDR estimation using gsolve
    # Log exposure time, weighting function
    B = np.log(exposures)
    weight_func = lambda z: (128-abs(z-128))

    # Return variables
    hdr = np.zeros_like(ldrs[0], dtype=np.float32)
    estimated_g = []
    log_irradiances = np.asarray([np.zeros_like(hdr) for _ in range(N)])

    # Randomly sample 100 points in images to create z
    mask = np.ones((H, W), dtype=np.bool)
    for h in range(H):
        for w in range(W):
            if np.sqrt((center_h - h)**2 + (center_w - w)**2) > R:
                mask[h, w] = False

    h, w = np.where(mask == 1)
    ind = np.random.choice(len(h), n_point)
    chosen_h, chosen_w = h[ind], w[ind]

    # Perform estimation for each channel
    for c in range(C):
        Z = np.zeros((N, n_point), dtype=np.int64)
        for l_id, ldr in enumerate(ldrs):
            Z[l_id] = np.reshape(ldr[chosen_h, chosen_w, c], -1)

        g, le = gsolve(Z, B, lamb, weight_func)
        estimated_g.append(g)

        # Reconstruct HDR image
        hdr_delim = np.zeros(hdr.shape[:-1])
        for l_id, (ldr, exp) in enumerate(zip(ldrs, B)):
            ldr_one_c = ldr[:, :, c]
            hdr[:, :, c] += weight_func(ldr_one_c).astype(np.float32) * (g[ldr_o
ne_c] - exp)
            hdr_delim += weight_func(ldr_one_c).astype(np.float32)

            log_irradiances[l_id, :, :, c] = (g[ldr_one_c] - exp)
        hdr[:, :, c] = hdr[:, :, c] / hdr_delim

    hdr = np.exp(hdr)
    hdr = np.nan_to_num(hdr)

    return hdr, log_irradiances, np.asarray(estimated_g)
```

In [12]:

```
# get HDR image, log irradiance
estimated_hdr_image, estimated_log_irradiance, estimated_g = make_hdr_estimation
(ldr_images, exposures)
# make_hdr_estimation(ldr_images, exposures)

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

# display HDR image
_ = display_hdr_image(estimated_hdr_image)
plt.show()

# display log irradiance image
_ = display_log_irradiances(estimated_log_irradiance)
plt.show()

save_images(estimated_hdr_image, estimated_log_irradiance, "estimation")
```

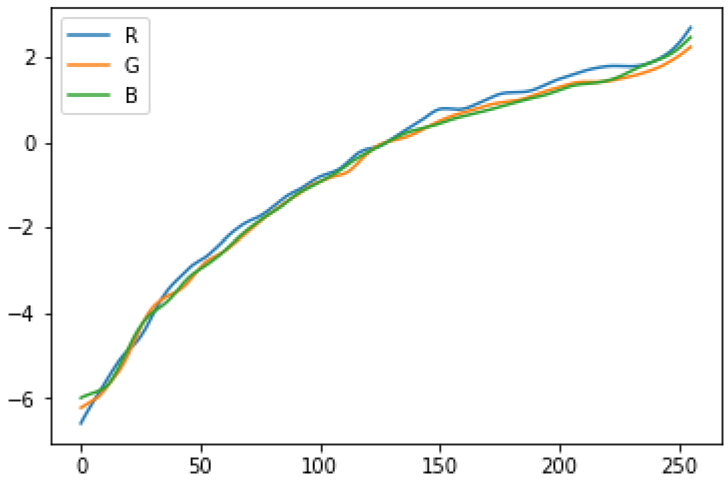
```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:79: Run
timeWarning: invalid value encountered in true_divide

Warning: Negative / Inf values found in hdr image. Clamping to neare
st valid value
```





In [13]:

```
# display G function for each intensity values
N, NG = estimated_g.shape
labels = ['R', 'G', 'B']
fig = plt.figure()
for n in range(N):
    plt.plot(range(NG), estimated_g[n], label=labels[n])
plt.gca().legend(('R', 'G', 'B'))
plt.show()
fig.savefig("images/outputs/estimated_functions.png")
plt.close(fig)
```



Panoramic transformations (20 pts)

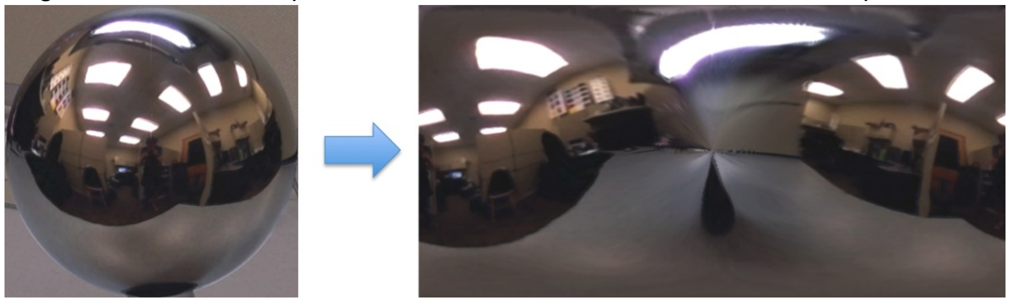
Now that we have an HDR image of the spherical mirror, we'd like to use it for relighting (i.e. image-based lighting). However, many programs don't accept the "mirror ball" format, so we need to convert it to a different 360 degree, panoramic format (there is a nice overview of many of these formats here). For this part of the project, you should implement the mirror ball to equirectangular (latitude longitude) transformation. Most rendering software accepts this format, including Blender's Cycles renderer, which is what we'll use in the next part of the project.

To perform the transformation, you need to figure out the mapping between the mirrored sphere domain and the equirectangular domain. Hint: calculate the normals of the sphere (N) and assume the viewing direction (V) is constant. You can calculate reflection vectors with R = V - 2 ***dot(V,N)*** N, (NOTE that you'd have to implement channel-wise dot product). which is the direction that light is incoming from the world to the camera after bouncing off the sphere. The reflection vectors can then be converted to, providing the latitude and longitude (phi and theta) of the given pixel (fixing the distance to the origin, r, to be 1). Note that this assumes an orthographic camera (which is a close approximation as long as the sphere isn't too close to the camera).

Next, the equirectangular domain can be created by making an image in which the rows correspond to theta and columns correspond to phi in spherical coordinates, e.g.

```
EH, EW = 360, 720
phi_fst_half = np.arange(math.pi, 2*math.pi, math.pi / (EW // 2))
phi_snd_half = np.arange(0 * math.pi, math.pi, math.pi / (EW // 2))

theta_range = np.arange(0, math.pi, math.pi / EH)
phi_ranges = np.concatenate((phi_fst_half, phi_snd_half))
phis, thetas = np.meshgrid(phi_ranges, theta_range)
```

Note that by choosing 360 as EH and 720 as EW, we are making every pixel in equirectangular image to correspond to area occupied by 0.5 degree x 0.5 degree in spherical coordinate. Now that you have the phi/theta for both the mirror ball image and the equirectangular domain, use scipy's scipy.interpolate.griddata function to perform the transformation. Below is an example transformation.



Note that the following portion of the project depends on successfully converting your mirror ball HDR image to the equirectangular domain. If you cannot get this working, you can request code from the instructors at a 20 point penalty (i.e. no points will be awarded for this section, but you can do the later sections).

In [14]:

```
hdr_mirrorball_image = read_hdr_image('images/outputs/estimated_hdr.hdr')
```

In [15]:

```
def get_reflection(V, N):
    # R = V - 2 * dot(V, N) * N
    Vx, Vy, Vz = V
    Nx, Ny, Nz = N

    dot = Vx * Nx + Vy * Ny + Vz * Nz
    Rx = Vx - 2 * dot * Nx
    Ry = Vy - 2 * dot * Ny
    Rz = Vz - 2 * dot * Nz

    return (Rx, Ry, Rz)
```

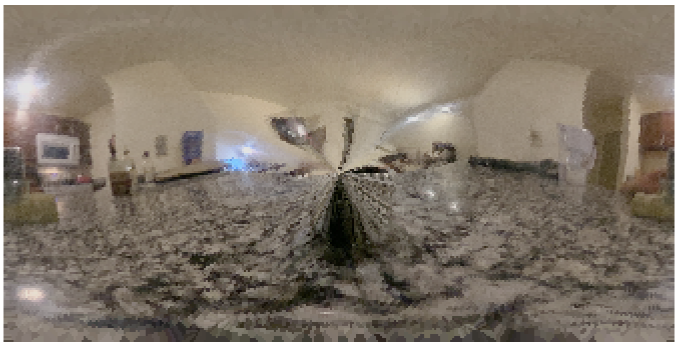
In [16]:

```
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 normal vector from mirror ball
    2) Compute reflection vector of mirror ball using given equation
    3) Map reflection vectors into spherical coordinates
    4) Interpolate spherical coordinate values into equirectangular grid.
      - hint: use scipy.interpolate.griddata


    '''
    H, W, C = hdr_image.shape
    assert H == W
    assert H % 2 == 0
    assert C == 3
    R = H // 2

    EH, EW = 360, 720
    phi_fst_half = np.arange(math.pi, 2*math.pi, math.pi / (EW // 2))
    phi_snd_half = np.arange(0 * math.pi, math.pi, math.pi / (EW // 2))
    theta_range = np.arange(0, math.pi, math.pi / EH)
    phi_ranges = np.concatenate((phi_fst_half, phi_snd_half))
    phis, thetas = np.meshgrid(phi_ranges, theta_range)

    points, r_pixels, g_pixels, b_pixels = [], [], [], []
    center_h, center_w = H/2, W/2
    for h in range(H):
        for w in range(W):
            Ny = (center_h - h) / R
            Nx = (w - center_w) / R

            Nxy_sqr_sum = Ny**2 + Nx**2
            if math.sqrt(Nxy_sqr_sum) > 1: # Outside the sphere
                continue
            Nz = math.sqrt(1 - (Nxy_sqr_sum))

            Rx, Ry, Rz = get_reflection((0, 0, -1), (Nx, Ny, Nz))
            phi = np.arctan2(Rz, Rx) + np.pi/2
            phi %= np.pi * 2
            theta = np.arccos(Ry)
            points.append([phi, theta])

            r_pixels.append(hdr_image[h, w, 0])
            g_pixels.append(hdr_image[h, w, 1])
            b_pixels.append(hdr_image[h, w, 2])

    points = np.asarray(points, dtype=np.float64)
    r_pixels = np.asarray(r_pixels, dtype=np.float64)
    g_pixels = np.asarray(g_pixels, dtype=np.float64)
    b_pixels = np.asarray(b_pixels, dtype=np.float64)

    r = griddata(points, r_pixels, (phis, thetas), method='nearest')
    g = griddata(points, g_pixels, (phis, thetas), method='nearest')
    b = griddata(points, b_pixels, (phis, thetas), method='nearest')

    equirectangular_image = np.rollaxis(np.asarray([r, g, b]), 0, 3)
    return equirectangular_image
```

In [17]:

```
eq_image = panoramic_transform(naive_hdr_image)
display_hdr_image(eq_image.astype(np.float32))
plt.show()
write_hdr_image(eq_image, 'images/outputs/equirectangular.hdr')

eq_copy = (eq_image - np.min(eq_image)) / (np.max(eq_image) - np.min(eq_image))
fig = plot_no_frame(eq_copy)
fig.savefig("images/outputs/equirectangular.png")
plt.close(fig)
```

```
Warning: Negative / Inf values found in hdr image. Clamping to neare
st valid value
```



Rendering synthetic objects into photographs (30 pts)

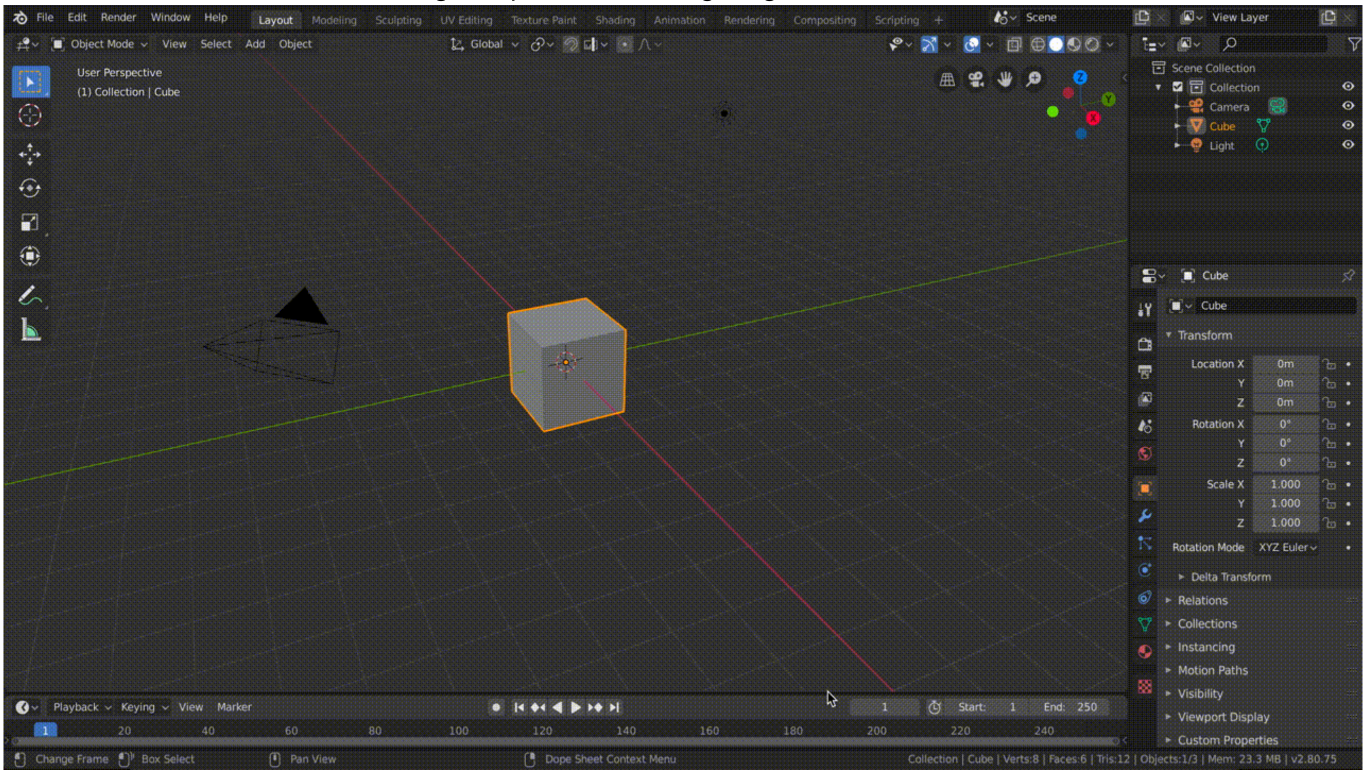
Next, we will use our equirectangular HDR image as an image-based light, and insert 3D objects into the scene. This consists of 3 main parts: modeling the scene, rendering, and compositing. Specific instructions follow below; if interested, see additional details in Debevec's paper.

Begin by downloading/installing the Blender. This template assumes that you have version 2.8 of the blender; if you are using version 2.7x, please refer to project webpage for detailed steps. The course webpage has tutorial with sample blend file, while this tutorial assumes that you create your own blend file from scratch. Please right click, open in new tab to view GIFs in full size.

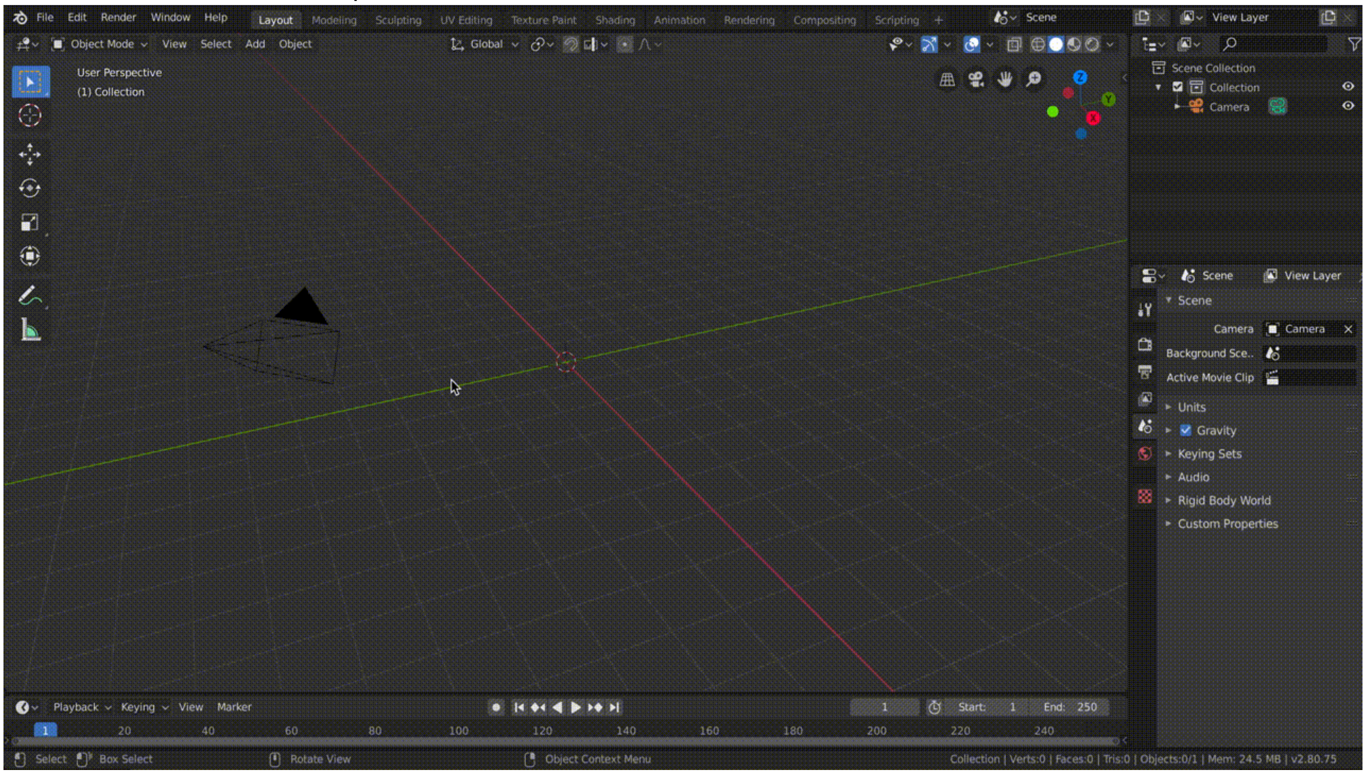
Modeling the scene

To insert objects, we must have some idea of the geometry and surface properties of the scene, as well as the lighting information that we captured in previous stages. In this step, you will manually create rough scene geometry/materials using Blender.

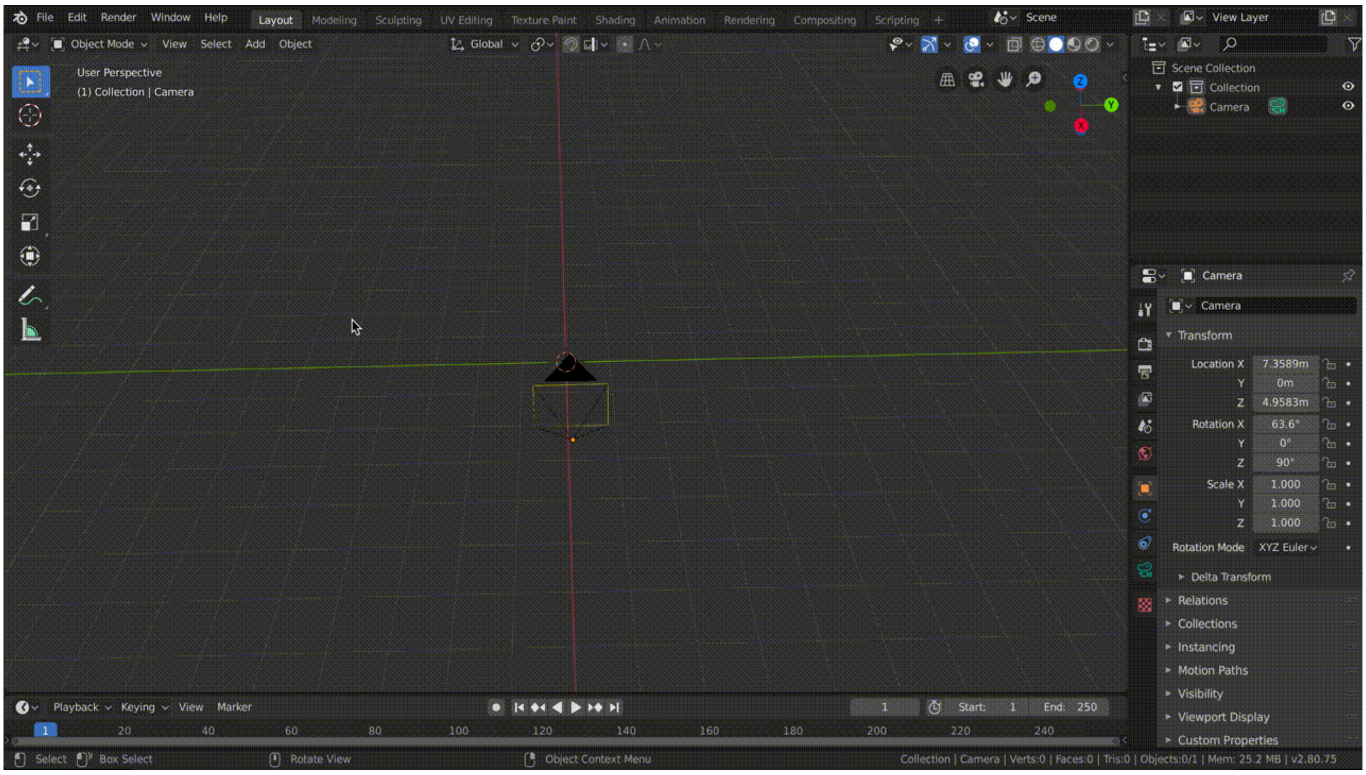
First clear out initial scene including sample mesh and lighting.


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Object Mode View Select Add Object
Global
User Perspective
(1) Collection | Cube
Scene Collection
Collection
Camera
Cube
Light
Transform
Location X 0m
Y 0m
Z 0m
Rotation X 0°
Y 0°
Z 0°
Scale X 1.000
Y 1.000
Z 1.000
Rotation Mode XYZ Euler
Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display
Custom Properties
Playback Keying View Marker
Start: 1 End: 250
Change Frame Box Select Pan View Dope Sheet Context Menu
Collection | Cube | Verts:8 | Faces:6 | Tris:12 | Objects:1/3 | Mem: 23.3 MB | v2.80.75

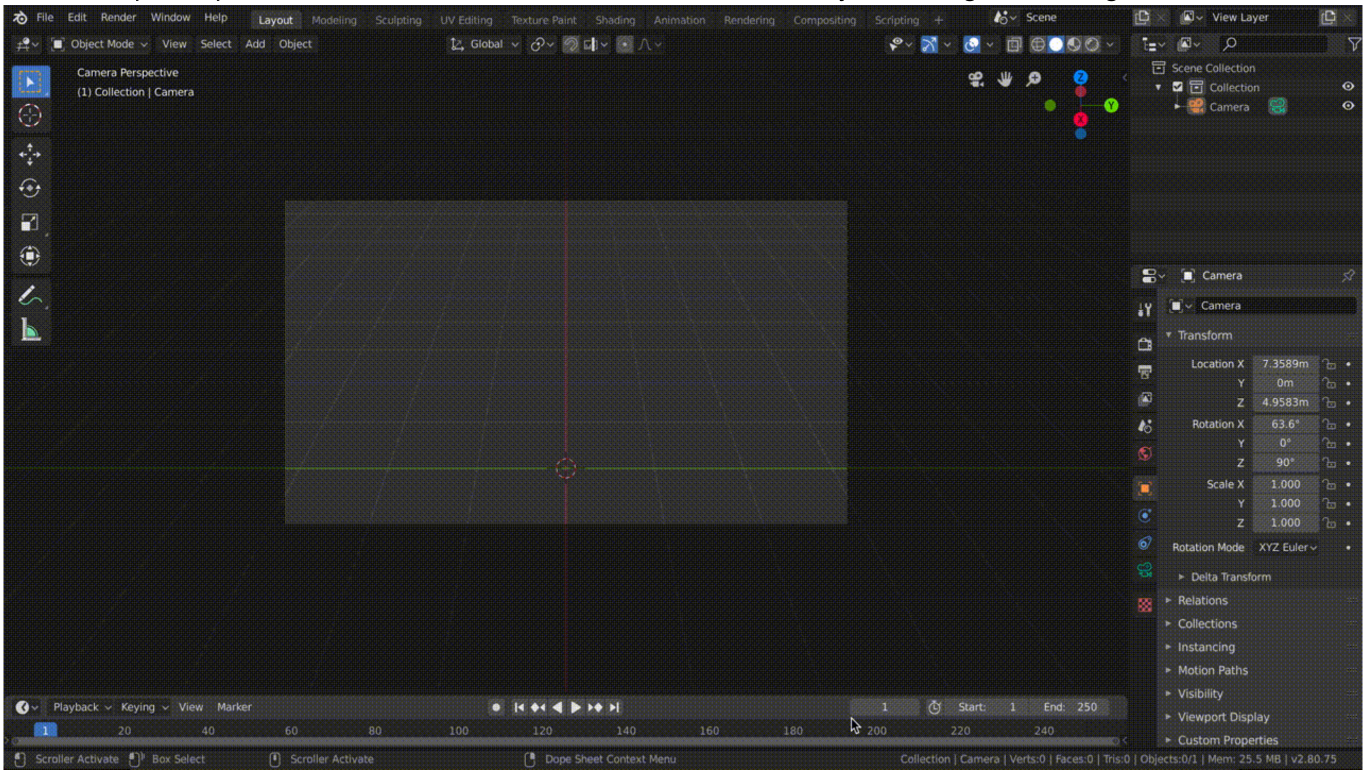
Next, set camera to correct position


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Object Mode View Select Add Object
User Perspective
(1) Collection
Scene Collection
Collection
Camera
Scene View Layer
Camera Camera
Background Sce..
Active Movie Clip
Units
Gravity
Keying Sets
Audio
Rigid Body World
Custom Properties
Select Box Select Rotate View Object Context Menu
Collection | Verts:0 | Faces:0 | Tris:0 | Objects:0/1 | Mem: 24.5 MB | v2.80.75

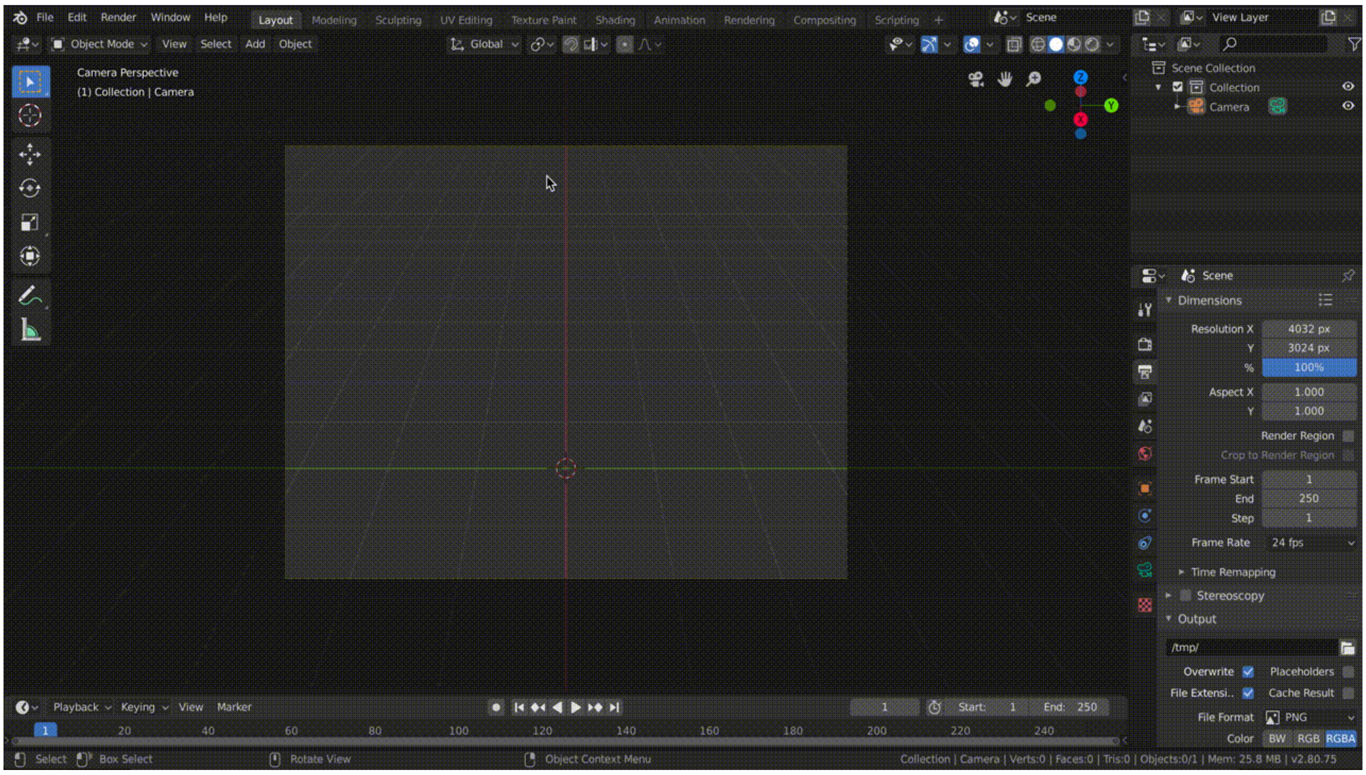
Next, go into perspective camera mode.



Then, setup viewport dimension. You want the same dimension as your background image for this.



We then want to load background image onto this viewport. Click on 'g' to move background image around, and 's' to resize it.



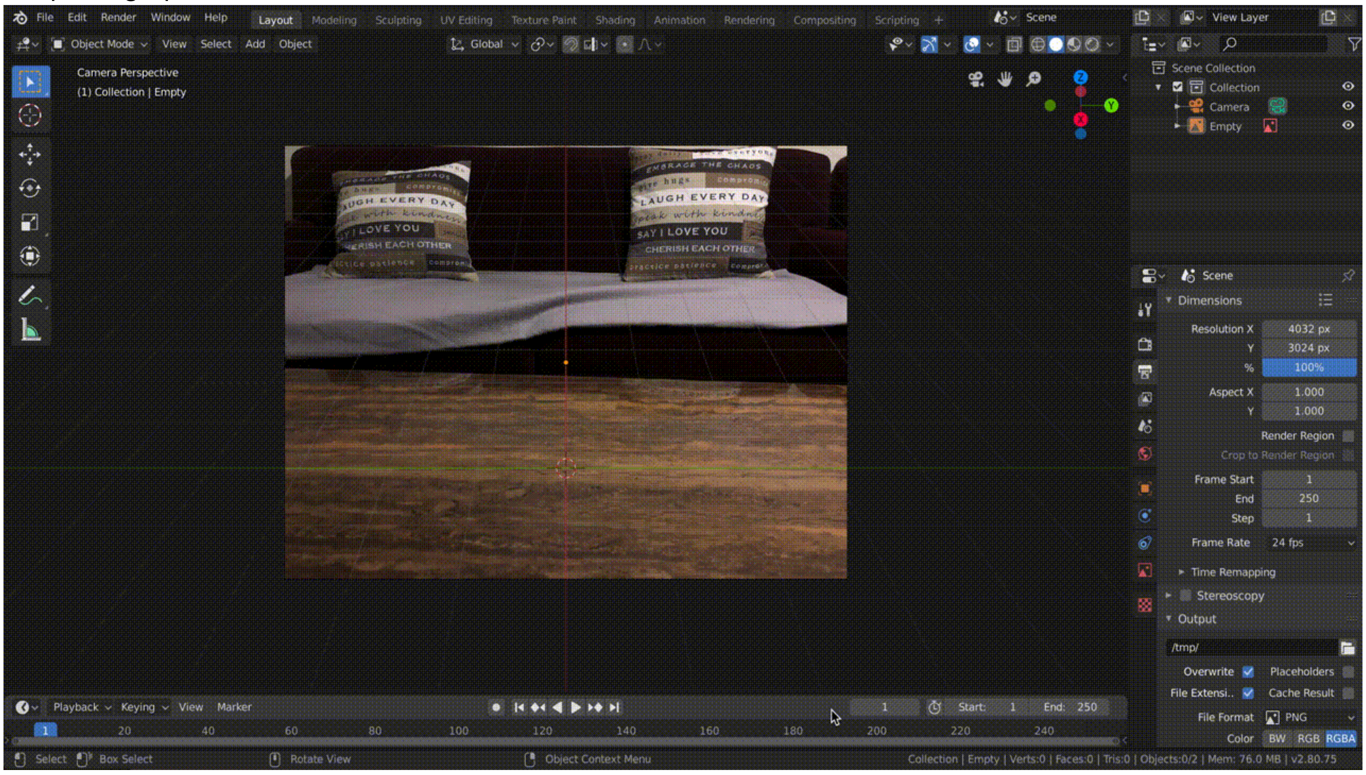
Camera Perspective
(1) Collection | Camera

Resolution X 4032 px
Y 3024 px
% 100%
Aspect X 1.000
Y 1.000
Render Region
Crop to Render Region
Frame Start 1
End 250
Step 1
Frame Rate 24 fps
/tmp/
Overwrite ✓ Placeholders
File Extensi.. ✓ Cache Result
File Format PNG
Color BW RGB RGBA

1 Start: 1 End: 250

Collection | Camera | Verts:0 | Faces:0 | Tris:0 | Objects:0/1 | Mem: 25.8 MB | v2.80.75

Then, we want to add local scene. That is, add simple geometry (usually planes suffice) to recreate the geometry in the scene near where you'd like to insert objects. For best results, this should be close to where you placed the spherical mirror. Feel free to use the sample scene provided and move the vertices of the plane to match the surface you'd like to recreate (ignore the inserted bunny/teapot/etc for now). Once you're happy with the placement, add materials to the local scene: select a piece of local scene geometry, go to Properties->Materials, add a Diffuse BSDF material, and change the "Color" to roughly match the color from the photograph.

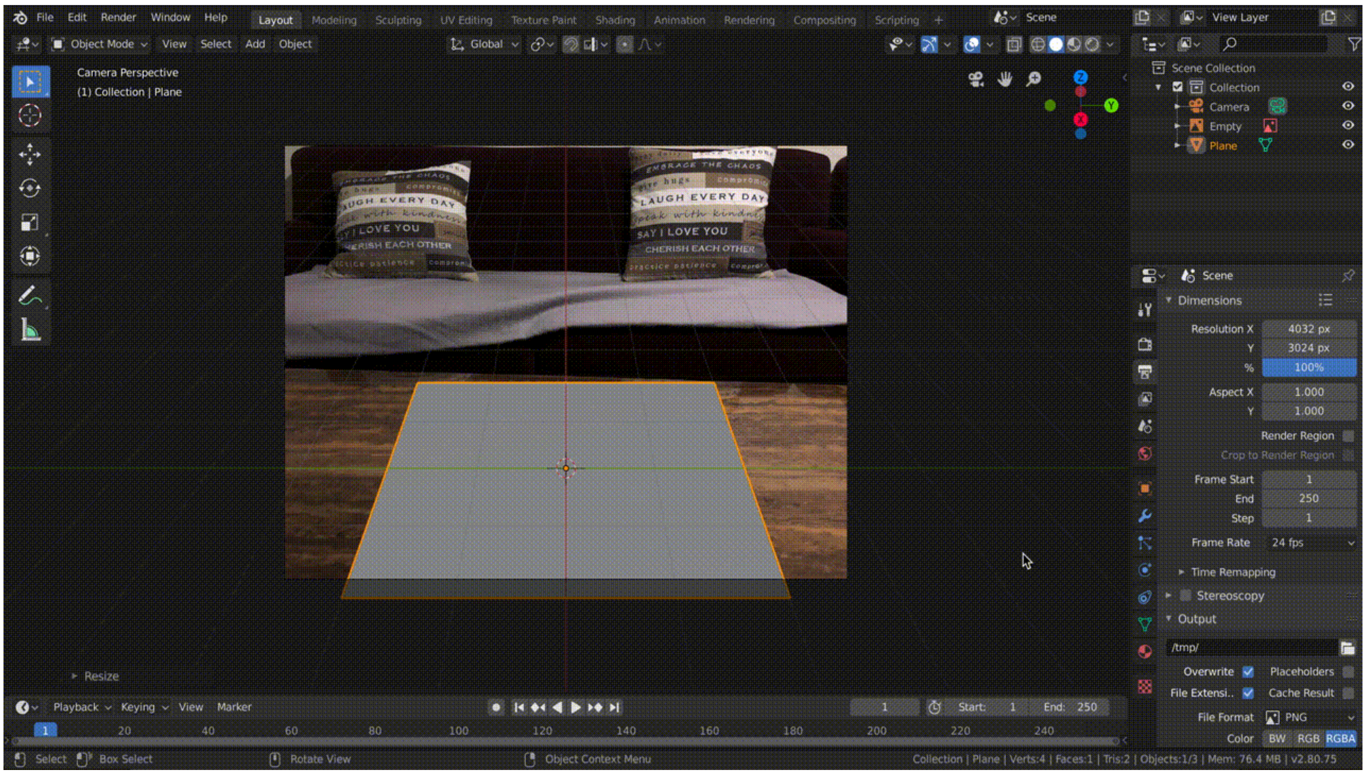


Camera Perspective
(1) Collection | Empty

Resolution X 4032 px
Y 3024 px
% 100%
Aspect X 1.000
Y 1.000
Render Region
Crop to Render Region
Frame Start 1
End 250
Step 1
Frame Rate 24 fps
/tmp/
Overwrite ✓ Placeholders
File Extensi.. ✓ Cache Result
File Format PNG
Color BW RGB RGBA

1 Start: 1 End: 250

Collection | Empty | Verts:0 | Faces:0 | Tris:0 | Objects:0/2 | Mem: 76.0 MB | v2.80.75

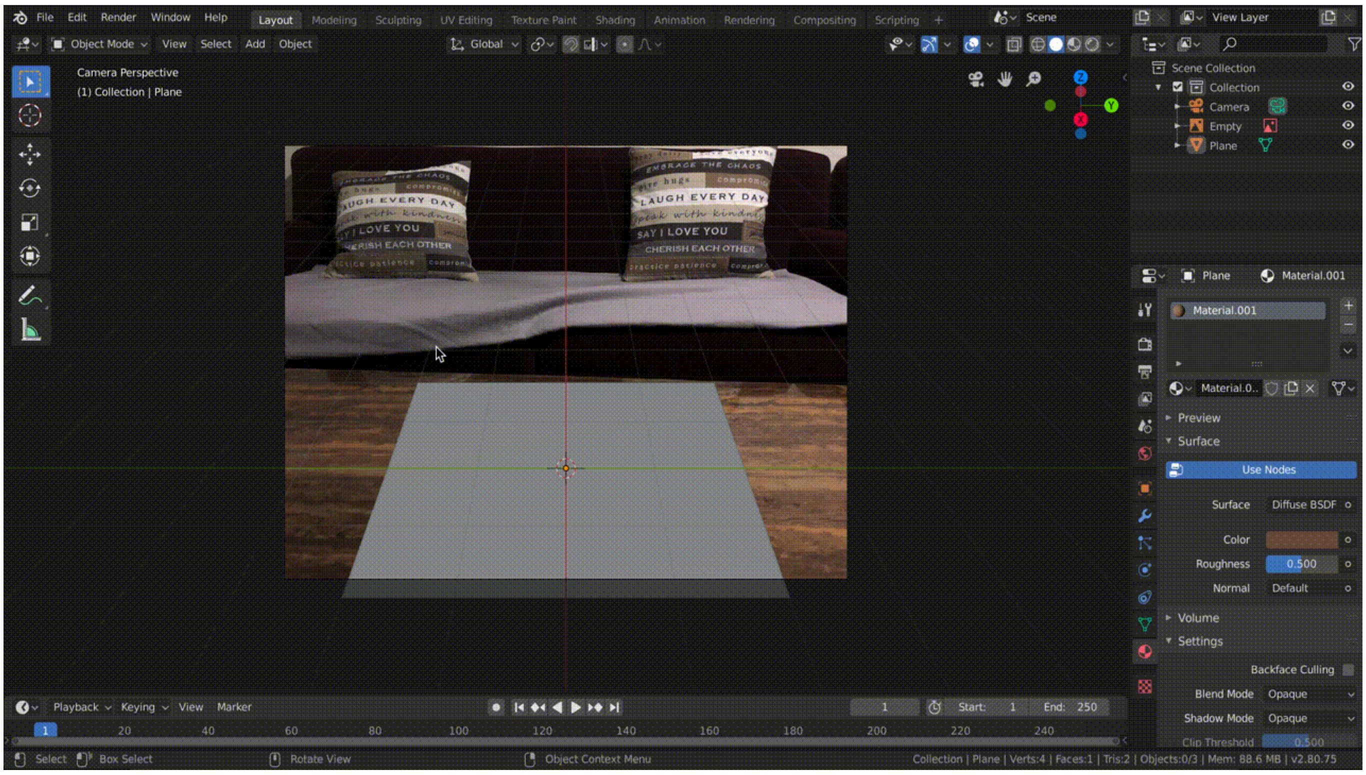

File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Object Mode View Select Add Object
Global
Camera Perspective
(1) Collection | Plane
Scene Collection
Collection
Camera
Empty
Plane
Resize
Playback Keying View Marker

Scene
Dimensions
Resolution X 4032 px
Y 3024 px
% 100%
Aspect X 1.000
Y 1.000
Render Region
Crop to Render Region
Frame Start 1
End 250
Step 1
Frame Rate 24 fps
Time Remapping
Stereoscopy
Output
/tmp/
Overwrite ✓ Placeholders
File Extensi.. ✓ Cache Result
File Format PNG
Color BW RGB RGBA

Start: 1 End: 250

Collection | Plane | Verts:4 | Faces:1 | Tris:2 | Objects:1/3 | Mem: 76.4 MB | v2.80.75

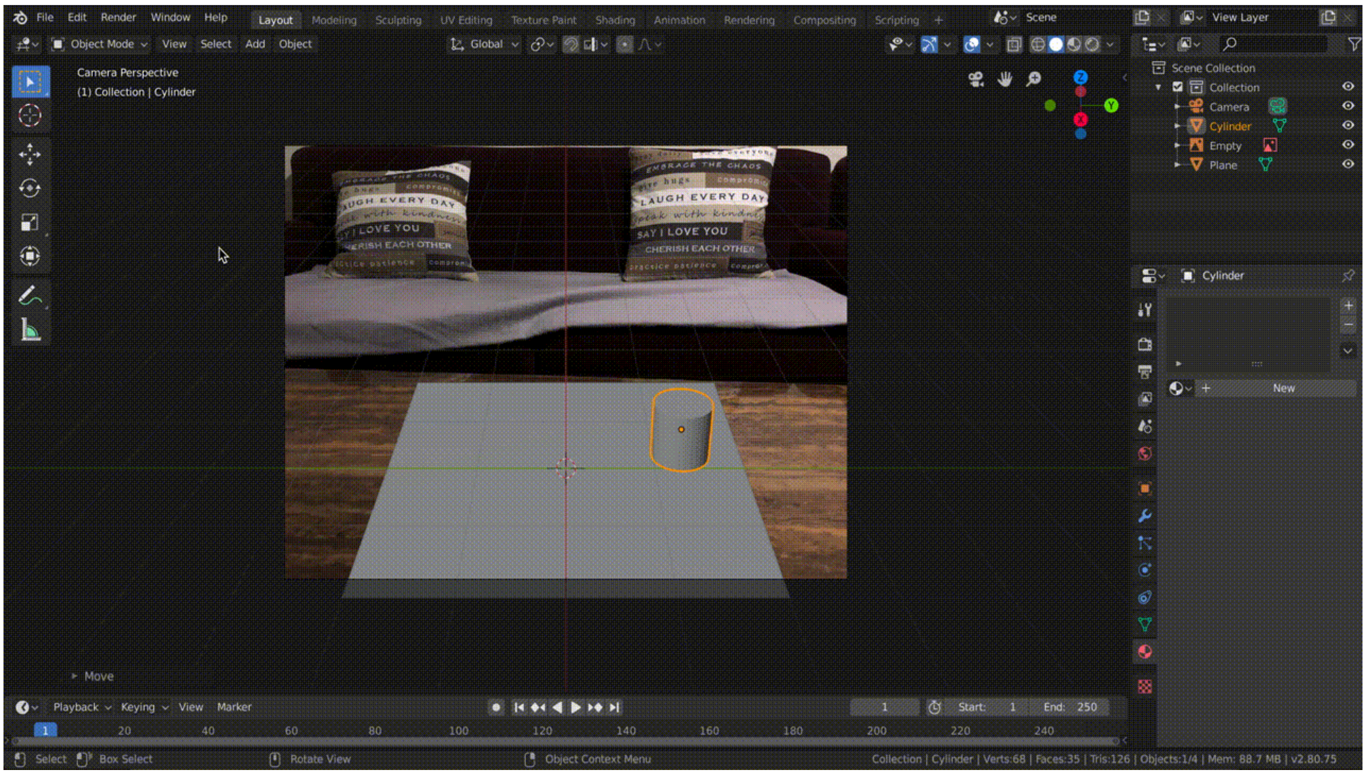
Insert synthetic objects into the scene. Feel free to use the standard models that I've included in the sample blend file, or find your own (e.g. Turbosquid, Google 3D Warehouse, DModelz, etc). Add interesting materials to your inserted objects as well. This tutorial is a great introduction to creating materials in Blender. Once finished, your scene should now look something like the right image below.


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Object Mode View Select Add Object
Global
Camera Perspective
(1) Collection | Plane
Scene Collection
Collection
Camera
Empty
Plane
Playback Keying View Marker

Plane Material.001
Material.001
Preview
Surface
Use Nodes
Surface Diffuse BSDF
Color
Roughness 0.500
Normal Default
Volume
Settings
Backface Culling
Blend Mode Opaque
Shadow Mode Opaque
Clip Threshold 0.500

Start: 1 End: 250

Collection | Plane | Verts:4 | Faces:1 | Tris:2 | Objects:0/3 | Mem: 88.6 MB | v2.80.75

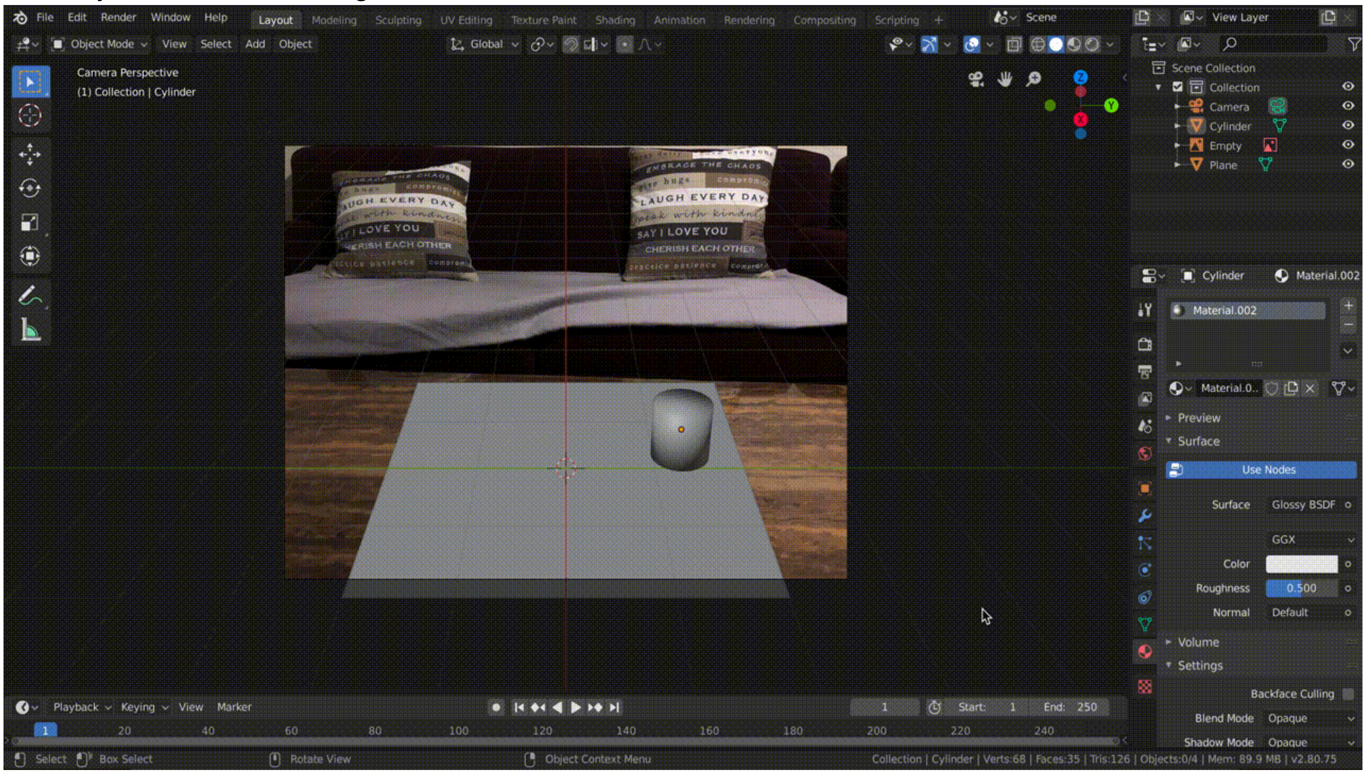

File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Scene
View Layer
Object Mode View Select Add Object
Global
Camera Perspective
(1) Collection | Cylinder
Scene Collection
Collection
Camera
Cylinder
Empty
Plane
Cylinder
New
Move
Playback Keying View Marker
Start: 1 End: 250
Select Box Select Rotate View Object Context Menu

Collection | Cylinder | Verts:68 | Faces:35 | Tris:126 | Objects:1/4 | Mem: 88.7 MB | v2.80.75

Then, add your HDR image (the equirectangular map made above) to the scene.

First, use notebook to save the HDR panorama: write_hdr_image(eq_image, 'equirectangular.hdr').

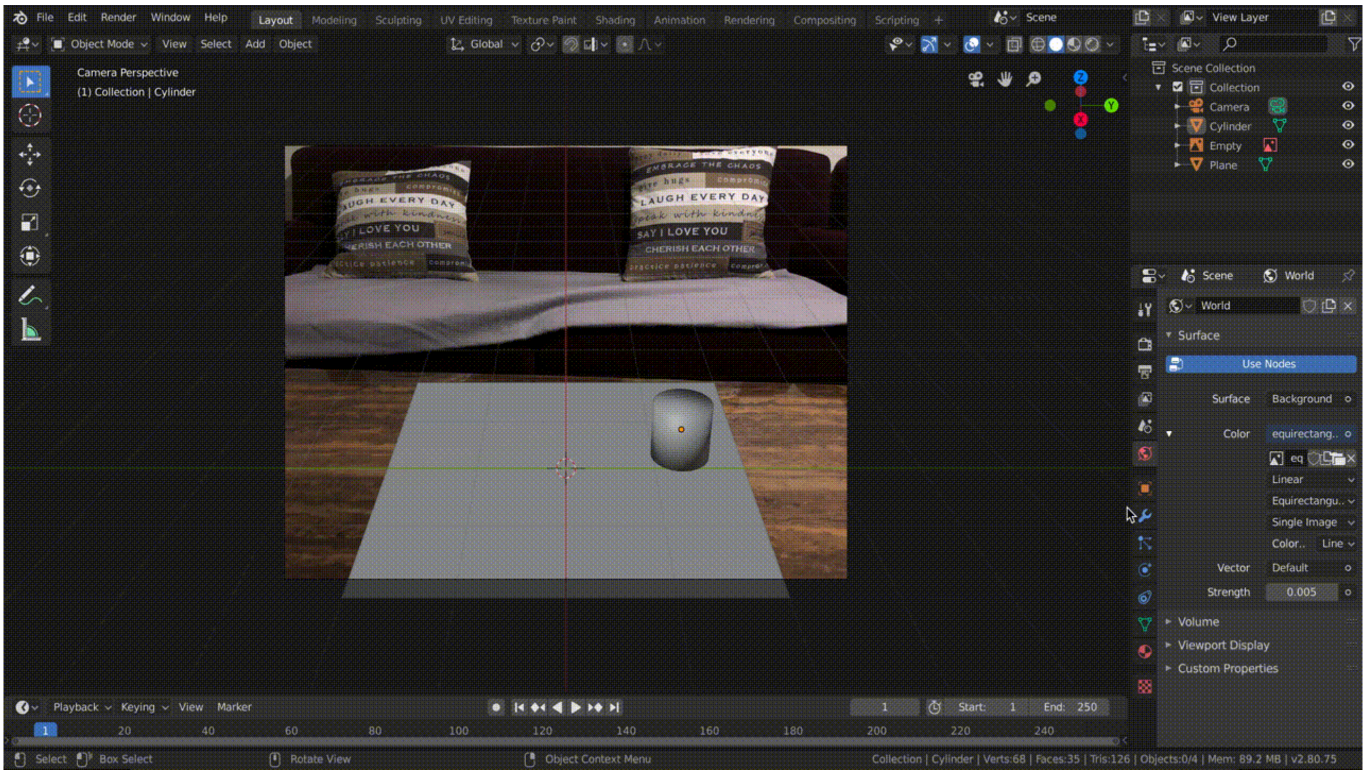
In the World tab on property panel, make sure Surface="Background" and Color="Environment Texture". Locate your saved HDR image in the filename field below "Environment Texture".


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Scene
View Layer
Object Mode View Select Add Object
Global
Camera Perspective
(1) Collection | Cylinder
Scene Collection
Collection
Camera
Cylinder
Empty
Plane
Cylinder
Material.002
Material.002
Material.0..
Preview
Surface
Use Nodes
Volume
Settings
Playback Keying View Marker
Start: 1 End: 250
Select Box Select Rotate View Object Context Menu

Surface: Glossy BSDF
GGX
Color
Roughness: 0.500
Normal: Default
Backface Culling
Blend Mode: Opaque
Shadow Mode: Opaque

Collection | Cylinder | Verts:68 | Faces:35 | Tris:126 | Objects:0/4 | Mem: 89.9 MB | v2.80.75

Finally, render scene using 'cycle' renderer. Note that this step takes a while to run.

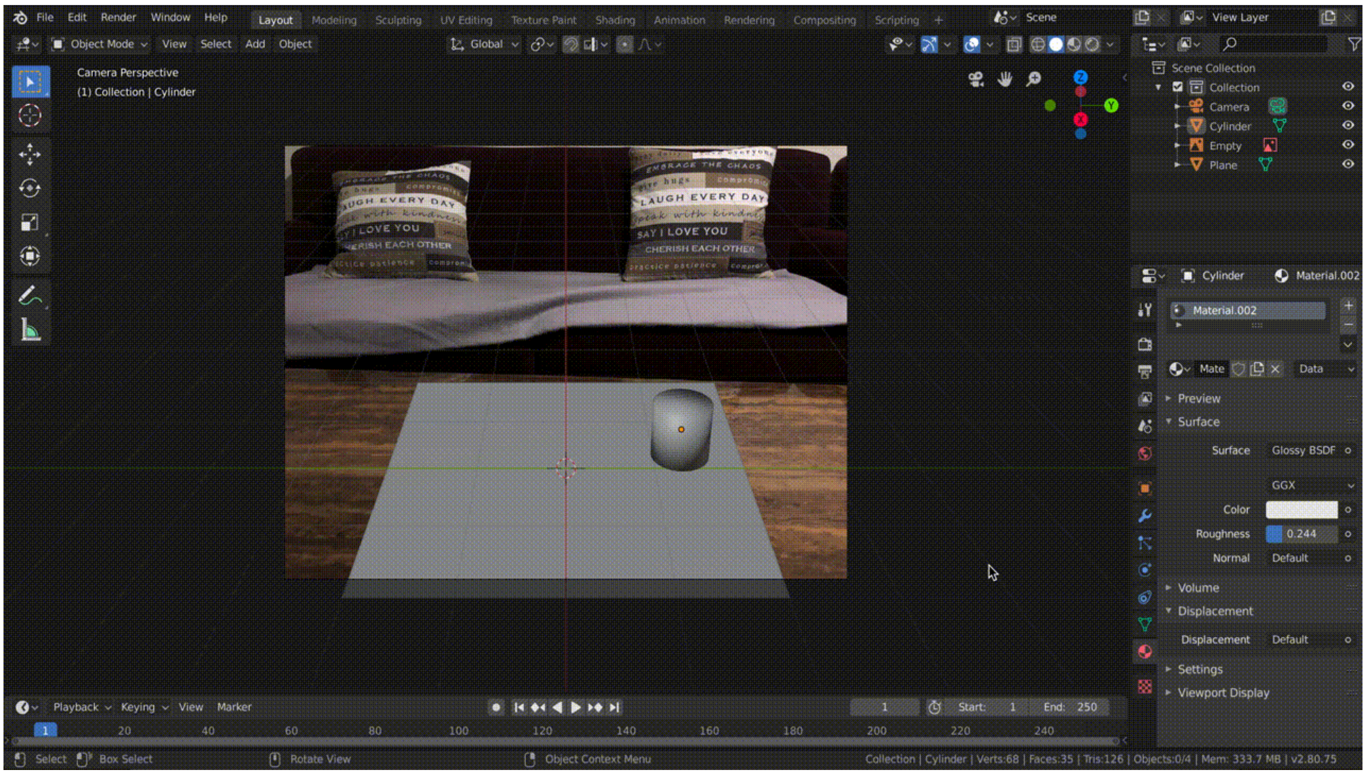


World
Surface
Use Nodes
Surface: Background
Color: equirectang..
eq
Linear
Equirectangu..
Single Image
Color.. Line
Vector: Default
Strength: 0.005

Collection | Cylinder | Verts:68 | Faces:35 | Tris:126 | Objects:0/4 | Mem: 89.2 MB | v2.80.75

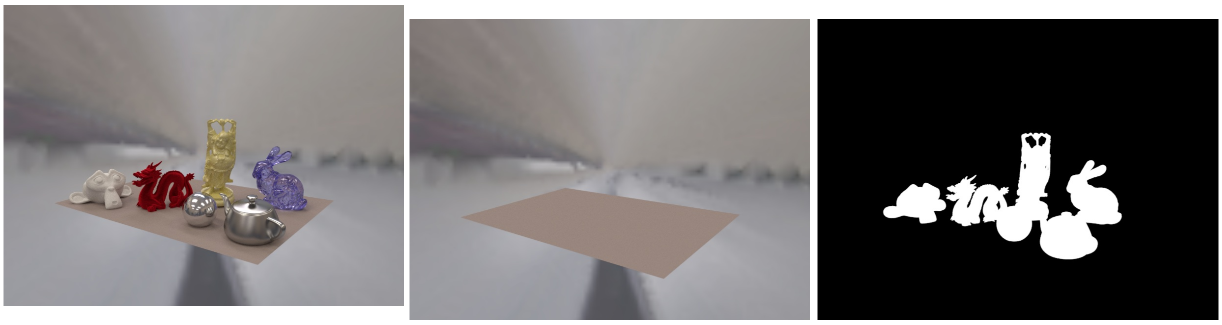
Once you've finished rendering scene with object, you should save it using image > save functionality in render tab.

You also need to create 'empty' scene without inserted objects, and mask for added objects. To render masked scene, remove equirectangular based lighting, and set all inserted objects' material to emission. You can render the mask by using 'Eevee' renderer


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Scene
View Layer
Object Mode View Select Add Object
Global
Camera Perspective
(1) Collection | Cylinder
Scene Collection
Collection
Camera
Cylinder
Empty
Plane
Cylinder
Material.002
Preview
Surface
Volume
Displacement
Settings
Viewport Display
Playback Keying View Marker
Select Box Select Rotate View Object Context Menu

Surface: Glossy BSDF
GGX
Color
Roughness: 0.244
Normal: Default
Displacement: Default
Start: 1 End: 250
Collection | Cylinder | Verts:68 | Faces:35 | Tris:126 | Objects:0/4 | Mem: 333.7 MB | v2.80.75

With all rendered images, you should have there three images, (rendered with object, without object, and mask):

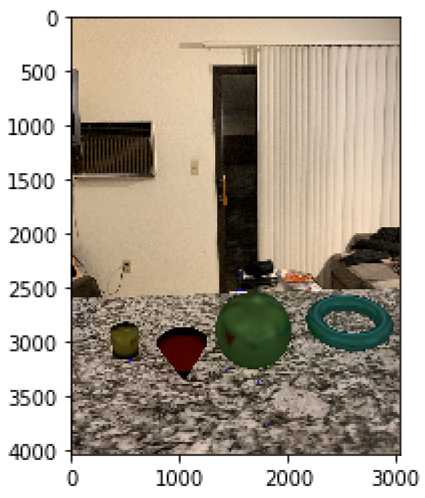


In [18]:

```
# this part assumes that you have these from Blender.
blender_output_with_object_path = "samples/r1.png"
blender_output_without_object_path = "samples/r2.png"
blender_output_mask_path = "samples/mask.png"

R = read_image(blender_output_with_object_path)
E = read_image(blender_output_without_object_path)
M = read_image(blender_output_mask_path)
M = M > 0.5
I = background_image
```

In [19]:

```
merged = M*R + (1-M)*I + (1-M)*(R-E)*0.7
plt.imshow(merged)
plt.show()
```



In [20]:

```
write_image(merged, 'images/outputs/merged.png')
```

Bells & Whistles (Extra Points)

Other panoramic transformations (20 pts)

Different software accept different spherical HDR projections. In the main project, we've converted from the mirror ball format to the equirectangular format. There are also two other common formats: angular and vertical cross (examples here). Implement these transformations for 10 extra points each (20 possible).

Photographer/tripod removal (20 pts)

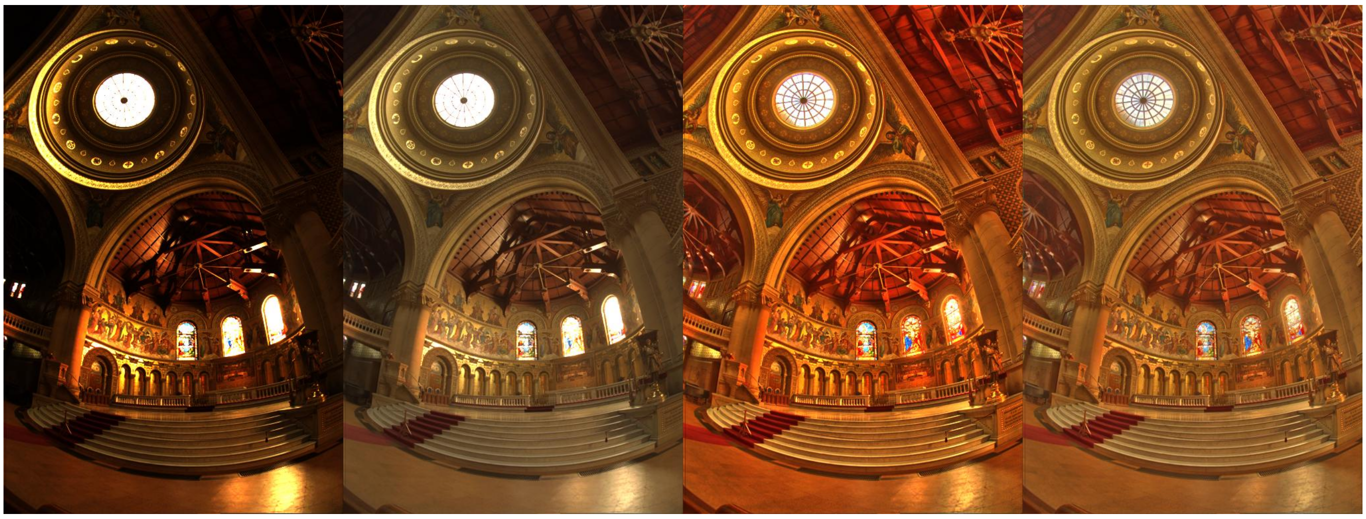
If you look closely at your mirror ball images, you'll notice that the photographer (you) and/or your tripod is visible, and probably occupies up a decent sized portion of the mirror's reflection. For 20 extra points, implement one of the following methods to remove the photographer:

1. Cut out the photographer and use in-painting/hole-filling to fill in the hole with background pixels (similar to the bells and whistles from Project 2), or
2. Use Debevec's method for removing the photographer (outlined here, steps 3-5; feel free to use Debevec's HDRShop for doing the panoramic rotations/blending).

The second option works better, but requires you to create an HDR mirror ball image from two different viewpoints, and then merge them together using blending and panoramic rotations.

Local tonemapping operator (30 pts)

HDR images can also be used to create hyper-realistic and contrast enhanced LDR images. This paper describes a simple technique for increasing the contrast of images by using a local tonemapping operator, which effectively compresses the photo's dynamic range into a displayable format while still preserving detail and contrast. For 30 extra credit points, implement the method found in the paper and compare your results to other tonemapping operations (see example below for ideas). You can use bilateral_filter code, provided by us, in your implementation, but do not use any other third party code. You can find some example HDR images here, including the memorial church image used below.


From left to right: simple rescaling, rescaling+gamma correction, local tonemapping operator, local tonemapping+gamma correction.

In [21]:

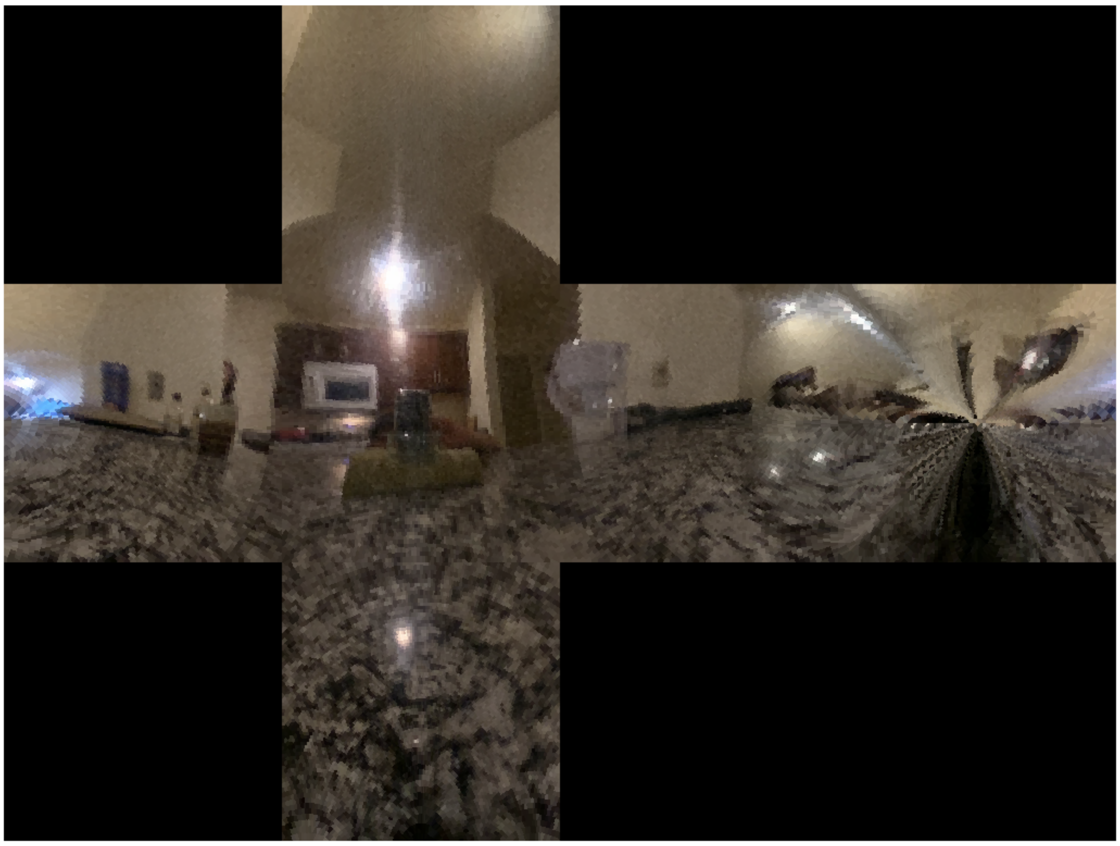
```
def convert_xyz_to_cube_uv(x, y, z):
    absX = abs(x)
    absY = abs(y)
    absZ = abs(z)

    isXPositive = 1 if x > 0 else 0;
    isYPositive = 1 if y > 0 else 0;
    isZPositive = 1 if z > 0 else 0;

    # POSITIVE X
    if isXPositive and absX >= absY and absX >= absZ:
        # u (0 to 1) goes from +z to -z
        # v (0 to 1) goes from -y to +y
        maxAxis = absX
        uc = -z
        vc = -y
        index = 0

    # NEGATIVE X
    if not isXPositive and absX >= absY and absX >= absZ:
        # u (0 to 1) goes from -z to +z
        # v (0 to 1) goes from -y to +y
        maxAxis = absX
        uc = z
        vc = -y
        index = 1

    # POSITIVE Y
    if isYPositive and absY >= absX and absY >= absZ:
        # u (0 to 1) goes from -x to +x
        # v (0 to 1) goes from +z to -z
        maxAxis = absY
        uc = x
        vc = z
        index = 2

    # NEGATIVE Y
    if not isYPositive and absY >= absX and absY >= absZ:
        # u (0 to 1) goes from -x to +x
        # v (0 to 1) goes from -z to +z
        maxAxis = absY
        uc = x
        vc = -z
        index = 3

    # POSITIVE Z
    if isZPositive and absZ >= absX and absZ >= absY:
        # u (0 to 1) goes from -x to +x
        # v (0 to 1) goes from -y to +y
        maxAxis = absZ
        uc = x
        vc = -y
        index = 4

    # NEGATIVE Z
    if not isZPositive and absZ >= absX and absZ >= absY:
        # u (0 to 1) goes from +x to -x
        # v (0 to 1) goes from -y to +y
        maxAxis = absZ
        uc = -x
        vc = -y
        index = 5

    # Convert range from -1 to 1 to 0 to 1
    u = 0.5 * (uc / maxAxis + 1.0)
    v = 0.5 * (vc / maxAxis + 1.0)

    return index, u, v
```

In [22]:

```
def interpolate(H, W, values, coords):
    h_ranges = np.arange(H)
    w_ranges = np.arange(W)
    w_grid, h_grid = np.meshgrid(w_ranges, h_ranges)

    r_pixels = np.asarray(values[:, 0], dtype=np.float64)
    g_pixels = np.asarray(values[:, 1], dtype=np.float64)
    b_pixels = np.asarray(values[:, 2], dtype=np.float64)

    r = griddata(coords, r_pixels, (w_grid, h_grid), method='nearest')
    g = griddata(coords, g_pixels, (w_grid, h_grid), method='nearest')
    b = griddata(coords, b_pixels, (w_grid, h_grid), method='nearest')
    pixels = np.rollaxis(np.asarray([r, g, b]), 0, 3)
    return pixels

def cube_transform(hdr_image):
    H, W, C = hdr_image.shape
    assert H == W
    assert H % 2 == 0
    assert C == 3
    R = H // 2

    cube_inds = [[] for _ in range(6)]
    cube_vals = [[] for _ in range(6)]

    center_h, center_w = H/2, W/2
    for h in range(H):
        for w in range(W):
            Ny = (center_h - h) / R
            Nx = (w - center_w) / R

            Nxy_sqr_sum = Ny**2 + Nx**2
            if math.sqrt(Nxy_sqr_sum) > 1: # Outside the sphere
                continue
            Nz = math.sqrt(1 - (Nxy_sqr_sum))

            Rx, Ry, Rz = get_reflection((0, 0, -1), (Nx, Ny, Nz))
            index, u, v = convert_xyz_to_cube_uv(Rx, Ry, Rz)

            u, v = int(u * H), int(v * H)
            cube_vals[index].append(hdr_image[h, w])
            cube_inds[index].append((u, v))

    lowest = np.ones(3) * 1e6
    cubes = []
    for ind in range(6):
        cube_vals[ind] = np.asarray(cube_vals[ind])
        cube_inds[ind] = np.asarray(cube_inds[ind])
        pixels = interpolate(H, W, cube_vals[ind], cube_inds[ind])
        for c in range(3):
            tmp_low = np.min(pixels[:, :, c])
            if tmp_low < lowest[c]:
                lowest[c] = tmp_low

        cubes.append(np.array(pixels))

    blank_image = np.zeros((H, W, C))
    blank_image[:, :] = lowest
    final_image = np.concatenate(
        (np.concatenate((blank_image, cubes[2], blank_image, blank_image), axis=
1),
         np.concatenate((cubes[1], cubes[4], cubes[0], cubes[5]), axis=1),
         np.concatenate((blank_image, cubes[3], blank_image, blank_image), axis=
1)),
        axis=0
    ).astype(np.float32)

    return final_image
```

In [23]:

```
img = cube_transform(naive_hdr_image)

fig = plot_no_frame((img - np.min(img)) / (np.max(img) - np.min(img)))
_ = plt.show()
fig.savefig("images/outputs/cubic.png")
```



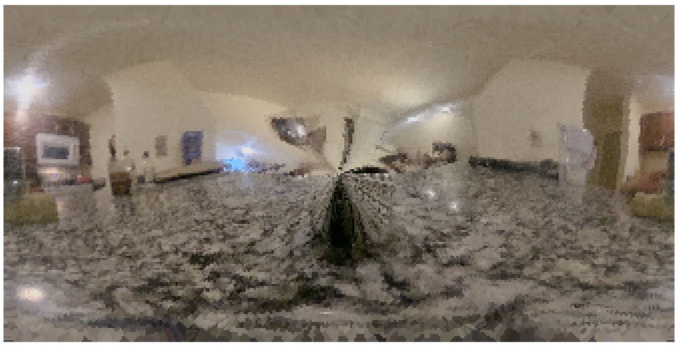
In [24]:

```
def ssd_patch(sample, mask, template, overlap,
              cor_sample=None, cor_template=None, alpha=0.5):

    ssd = np.zeros_like(sample)
    if mask is not None:
        mask = mask.astype(np.float32)
    if template is not None:
        ssd = ((mask * template) ** 2).sum() \
            - 2 * cv2.filter2D(sample, ddepth=-1, kernel=mask * template) \
            + cv2.filter2D(sample ** 2, ddepth=-1, kernel=mask)

    if cor_sample is not None:
        cor_template = cor_template.astype(np.float32)
        cor_ssd = (cor_template ** 2).sum() \
                - 2 * cv2.filter2D(cor_sample, ddepth=-1, kernel=cor_template) \
                + cv2.filter2D(cor_sample ** 2, ddepth=-1, kernel=np.ones_like(c
or_template))

        ssd = ssd * alpha + (1-alpha) * cor_ssd

    return ssd

def choose_sample(ssd, patch_shape, sample, k=5):
    h, w = patch_shape
    sh, sw = ssd.shape

    h_start = int(h/2); h_end = sh - int(h/2) + (h+1) % 2
    w_start = int(w/2); w_end = sw - int(w/2) + (w+1) % 2; w_len = w_end - w_sta
rt

    indices = np.argsort(ssd[h_start:h_end, w_start:w_end].flatten())

    ind = np.random.choice(indices[:k], 1)[0]
    h_i = int(ind / w_len); w_i = ind % w_len

    hs = h_start + h_i - int(h/2)
    he = h_start + h_i + int(h/2) + h % 2
    ws = w_start + w_i - int(w/2)
    we = w_start + w_i + int(w/2) + w % 2

    return sample[hs:he, ws:we, :], (hs, he), (ws, we)

def get_patch(img, h, w, half_h, half_w):
    if len(img.shape) == 2:
        hmax, wmax = img.shape
    else:
        hmax, wmax, _ = img.shape

    h_start = max(h - half_h, 0)
    h_end   = min(h + half_h + 1, hmax)
    w_start = max(w - half_w, 0)
    w_end   = min(w + half_w + 1, wmax)

    if len(img.shape) == 2:
        return img[h_start:h_end, w_start:w_end]
    else:
        return img[h_start:h_end, w_start:w_end, :]

def get_priority(conf_map, front_pts, cur_mask, cur_img, half_h, half_w, h, w):
    # Update confidence map
    new_conf_map = np.array(conf_map)
    for pts in front_pts:
        h_i, w_i = pts
        conf_patch = get_patch(conf_map, h_i, w_i, half_h, half_w)
        conf = conf_patch.sum() / conf_patch.size
        new_conf_map[h_i, w_i] = conf

    # Get data
    # Compute surface normal
    h_kernel = np.asarray([[-0.25, 0, 0.25], [-0.5, 0, 0.5], [-0.25, 0, 0.25]])
    v_kernel = np.asarray([[0.25, 0.5, 0.25], [-0.25, -0.5, -0.25], [0, 0, 0]])
    x_norm = scipy.signal.correlate2d(cur_mask, h_kernel, mode='same')
    y_norm = scipy.signal.correlate2d(cur_mask, v_kernel, mode='same')
    norm = np.dstack((x_norm, y_norm))

    delim = np.sqrt(x_norm ** 2 + y_norm ** 2)
    delim[delim == 0] = 1
    norm /= (delim[:, :, np.newaxis].repeat(2, axis=-1))

    # Compute gradient
    gray = cv2.cvtColor(cur_img, cv2.COLOR_RGB2GRAY).astype(np.float32)
    gray[cur_mask] = None

    gradients = np.array(np.gradient(gray))
    gradients = np.nan_to_num(gradients)
    gradients = np.rollaxis(gradients, 0, 3)
    grad_norm = np.sqrt(gradients[:, :, 0] ** 2 + gradients[:, :, 1] ** 2)
    max_grad = np.zeros_like(gradients)

    for pts in front_pts:
        h_i, w_i = pts
        norm_patch   = get_patch(grad_norm,          h_i, w_i, half_h, half_w)
        grad_x_patch = get_patch(gradients[:, :, 0], h_i, w_i, half_h, half_w)
        grad_y_patch = get_patch(gradients[:, :, 1], h_i, w_i, half_h, half_w)

        nh, nw = norm_patch.shape
        max_grad_pt = np.argmax(norm_patch)
        max_grad_h, max_grad_w = int(max_grad_pt / nw), max_grad_pt % nw

        max_grad[h_i, w_i, 0] = grad_x_patch[max_grad_h, max_grad_w]
        max_grad[h_i, w_i, 1] = grad_y_patch[max_grad_h, max_grad_w]

    data = norm * max_grad
    data = np.sqrt(data[:, :, 0] ** 2 + data[:, :, 1] ** 2) + 1e-6

    # Compute priority
    priority = new_conf_map * data * cur_mask
    return new_conf_map, priority

def inpaint(img, mask, patch_size):
    from skimage.filters import laplace

    img = img.astype(np.float32)
    img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    h, w, c = img.shape
    half_h = half_w = int(patch_size/2)
    conf_map = (1 - mask).astype(np.float32)
    data = np.zeros((h, w), dtype=np.float32)
    cur_img = np.array(img).astype(np.float32)
    cur_img[mask] = 0
    cur_img_gray = cv2.cvtColor(cur_img, cv2.COLOR_RGB2GRAY)
    cur_mask = np.array(mask)

    total_painted = 0
    while cur_mask.sum() > 0:
        # Finding the front
        front_pts = np.argwhere(laplace(cur_mask) > 0)

        conf_map, priority = get_priority(conf_map, front_pts, cur_mask, cur_img
, half_h, half_w, h, w)
        pt_to_patch = np.argmax(priority)
        h_to_patch, w_to_patch = int(pt_to_patch / w), pt_to_patch % w

        # Search for the most suitable patch
        mask_patch = get_patch(cur_mask,     h_to_patch, w_to_patch, half_h, hal
f_w)
        img_patch  = get_patch(cur_img_gray, h_to_patch, w_to_patch, half_h, hal
f_w)
        mh, mw = mask_patch.shape

        # ssd_patch(sample, mask, template, overlap)
        ssd = ssd_patch(img_gray, 1 - mask_patch, img_patch, 0)
        max_cost = np.max(ssd)
        valid = cv2.filter2D(1 - cur_mask, ddepth=-1, kernel=np.ones((mh, mw)))
        ssd[valid != mh * mw] = max_cost
        ssd[cur_mask] = max_cost

        # def choose_sample(ssd, patch_shape, sample, k=5)
        patch, (hs, he), (ws, we) = choose_sample(ssd, (mh, mw), cur_img, k=1)

        # Inpaint the area
        paste_hs = max(h_to_patch - half_h, 0)
        paste_he = min(h_to_patch + half_h + 1, h)
        paste_ws = max(w_to_patch - half_w, 0)
        paste_we = min(w_to_patch + half_w + 1, w)
        mask_patch_rgb = mask_patch[:, :, np.newaxis].repeat(3, axis=-1)
        cur_img[paste_hs:paste_he, paste_ws:paste_we] = \
                (1 - mask_patch_rgb) * cur_img[paste_hs:paste_he, paste_ws:paste
_we] + \
                mask_patch_rgb       * cur_img[hs:he, ws:we]

        cur_img_gray = cv2.cvtColor(cur_img, cv2.COLOR_RGB2GRAY)

        # Update conf_map using the confidence of the center point
        next_mask = np.array(cur_mask)
        next_mask[paste_hs:paste_he, paste_ws:paste_we] = 0
        new_pad = np.logical_and(next_mask == 0, cur_mask == 1)
        conf_map[new_pad] = conf_map[h_to_patch, w_to_patch]
        cur_mask = next_mask

        if new_pad.sum() == 0:
            break

        total_painted += new_pad.sum()
        # print(f'Total painted: {total_painted}, Painted: {new_pad.sum()}, {cur
_mask.sum()} more to paint')

    return cur_img
```

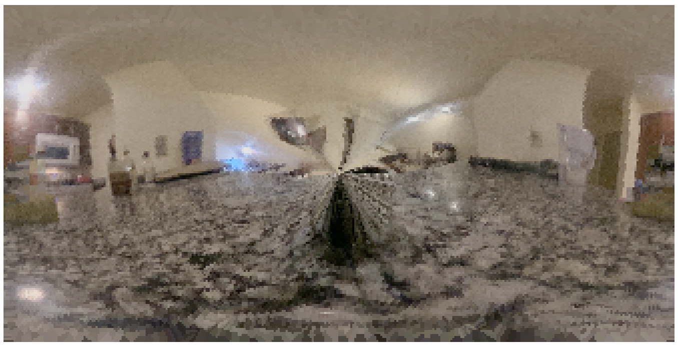
In [25]:

```
import scipy
display_hdr_image(eq_image.astype(np.float32))
plt.show()

mask = np.zeros(eq_image.shape[:2]).astype(np.bool)
mask[150:210, :45] = 1
mask[150:210, 659:] = 1

inpainted = inpaint(eq_image, mask, patch_size=15)
display_hdr_image(inpainted.astype(np.float32))
plt.show()
```

```
Warning: Negative / Inf values found in hdr image. Clamping to neare
st valid value
```



```
Warning: Negative / Inf values found in hdr image. Clamping to neare
st valid value
```



In [26]:

```
inpainted_copy = (inpainted - np.min(inpainted)) / (np.max(inpainted) - np.min(i
npainted))
plt.imshow(inpainted_copy)
plt.show()

fig = plot_no_frame(inpainted_copy)
fig.savefig("images/outputs/inpainted.png")
plt.close(fig)
```