CS461/CS561: Mixed Reality
Short Project 3: Acquiring Illumination and Insertion of 3D Content in 3D (Augmented Reality 3)
20 points

In this short project, you will use the discussion in Lecture 06: Acquiring Illumination to obtain an environment map corresponding to the scene you reconstructed in Short Project 2. You will also use the discussion in Lecture 05: Inserting Objects into Photographs in 3D to insert the 3D models of an object into your photographs from Short Project 2 and re-light them with the environment map such that it casts shadows on the floor plane.

Since you are working in teams of 2 and each student has one photograph, you need to have two distinct results of two separate scenes. Which means you need to do all steps in this project twice.

You should put the calls for all steps below in a single script.

This is a long and considerably challenging assignment. Yes, the write-up is 10 pages long, and yes, you need to read all 10 pages. However, it is set up to guide you folks through the process of completing Short Project 3.

A successful run through this project is extremely rewarding, and will prepare you folks to be ready for taking on challenges in industry related to augmented reality content.

Part 0: Variables needed from Short Project 2

You will need to have the following variables from Short Project 2 in your workspace:

expandedim (expanded image)
floorx (x coordinates in 2D of floor plane)
floory (y coordinates in 2D of floor plane)
f (focal length)
vx (x coordinate of vanishing point)
vy (y coordinate of vanishing point)
floorplane (3D coordinates of floor plane)
ceilplane (3D coordinates of ceiling plane)
leftplane (3D coordinates of left plane)
rightplane (3D coordinates of right plane)
backplane 3D coordinates of back plane)

Part 1: Acquire the Scene Illumination

Sub-part 1.1 [1.5 points]: Take multiple exposure photographs of a mirrored sphere placed in your scene.

Place your mirrored sphere in the center of your scene. The height of the sphere should be approximately the same as the height of your camera. Your best bet is to place the sphere on an open empty narrow-mouthed bottle (e.g., an Aquafina bottle), and take your photograph from a camera placed on a low coffee table located at the same position and viewpoint as where you photographed your scene for Short Project 2.

Your camera location does not have to exactly match Short Project 2, but it should be in the same general location. The viewpoint should match as best as possible. Also, you do not need to use the same camera as in Short Project 2, but you do need to have multiple exposure control over your camera. Most modern smartphones with the latest versions of Android and iOS have multiple exposure control.

Important: you must stabilize your camera! If your camera is not stable, your images will have slight offsets, and you will see a ghosted result. Easiest way to do this is to tape your camera to a heavy box (you can put a soup can in the box containing the mirrored sphere, and use the weighted box to anchor the camera).

Make sure to have the highest possible resolution selected on your camera for image capture. Once you have stabilized your camera, go to the multiple exposure settings of your camera, and take photographs from the lowest exposure to the highest exposure. You should take at least 5-7 photographs (use an odd number). Upload these photographs to your computer, and **save them in a folder titled ‘originalSpherePhotographs’**. The images should be labeled ‘im1.jpg’ to ‘imN.jpg’, where ‘im1.jpg’ is the image of lowest exposure (darkest image), ‘imN.jpg’ is the image of highest exposure (brightest image), and N is the total number of multiple exposure photographs.

Sub-part 1.2 [1.5 points]: Crop the photographs

Use the built-in function ‘imcrop’ in MATLAB to crop each sphere photograph so that all you see is the sphere. You need to make sure your crops are square. The easiest way to do this is to use the first version of imcrop in the slides to perform a manual crop the first photograph, i.e.

```
[im1cropped,rectangle ] = imcrop( im1 );   % make sure to read im1 in first
```

‘rectangle’ is a 1x4 vector given as [starting_x, starting_y, width, height] for a rectangle containing the cropped pixels. In ‘rectangle’, ‘width’ and ‘height’ need not be the same, as you may have a harder time figuring out the height due to the occlusion from your support. So set ‘rectangle’ to be [starting_x, starting_y, width, width] (i.e., set the height to be equal to the width).

Then for every image from im1 to imN, do

```
[im*cropped,rectangle] = imcrop( im* ); % replace * with 1 to N
```

Your crops should be in the vicinity of around 200x200 pixels (between 170x170 and 220x220 is okay). If they are any smaller or bigger, you may need to re-do Subparts 1.1 and 1.2 (an environment map that is too small will not capture enough detail, and one that is too big will decelerate your render run time in Part 2).

Save each cropped image under a directory ‘croppedSphereImages’, in the same fashion, where im1.jpg is the lowest exposure cropped image and imN.jpg is the highest exposure cropped image. (Note: we call them images as opposed to photographs as photographs are shot by a camera. Once you modify a photograph it loses its ‘photograph’ feature).

Sub-part 1.3 [3 points]: Convert the cropped photographs to an HDR image

Use EXIF tags for the photographs in the folder ‘originalSpherePhotographs’ to obtain the shutter times for all multiple exposure images. Create a vector B with the shutter times.

For this part, you will need to adapt the ‘gsolve’ function (included in this project) to perform sparse matrix computations, instead of the full matrix computations it currently does. **Important: the full matrix computations can take a huge amount of memory and will either not run, or will freeze your computer.** Use Lecture 03: Gradient Domain Composition to guide you in going from full to sparse matrix setup.

Create a matrix Z of size numpixels*3 x numimages. Each column in Z should be a vectorized version of the corresponding cropped image in the ‘croppedSphereImages’ folder. Each vector should contain double values from 0 to 255.

Run the gsolve function with sparsified matrix computations as:

```
      [g,lE] = gsolve( Z,B,1 );
```

Here IE is the logarithm of the irradiance. Get the original HDR image, i.e., the irradiance E as the exponential of IE.

Save the imaging response function g and the irradiance E to a .mat file ‘envmap2D_HDR.mat’.

Note: the irradiance E obtained in the above steps is correct up to a multiplicative factor ‘alpha’. Typically, this multiplicative factor is determined by assuming that the central image in your set of multiple exposure images is at an exposure of 1. However, since not all smartphone cameras have control over the absolute exposure settings, you will take care of ‘alpha’ in Part 2, where you will re-scale the output illumination by a manually chosen scale factor that allows you to see the render without over-saturation or under-saturation.

Sub-part 1.4 [4 points]: Convert the 2D HDR spherical map image to 3D world coordinates

Write a function ‘envmap2Dto3D.m’ that has the following form:

```
    function [e_xyz, e_rgb] = envmap2Dto3D( E )
```

Here, the input E is the irradiance map from Subpart 1.3. It will be of size m x m, where m should be close to 200 (i.e., somewhere between 170 and 220).

The output e_xyz is an n x 3 matrix of 3D world coordinates for all valid points in the environment map (i.e., points inside the circle containing the mirrored sphere in E). Since not all points are valid, n should be less than m*m.

The output e_rgb is an n x 3 matrix of r, g, and b values from E for all valid points in the environment map.

To write this function, have two nested for loops, one over the columns of E (parametrized by u), and one over the rows of E (parameterized by v). Within the nested for loop, use the slides titled ‘3D World Coordinates from 2D Spherical Map’ in Lecture 06 to compute u’ and v’, and r. If the value of r is valid, compute theta, phi, x, y, and z, push a 1x3 vector into e_xyz with x, y, and z, and push a 1x3 vector into e_rgb containing the r, g, and b values obtained from E at row v and column u. Yyou can push a 1x3 vector v into an increasing matrix M by saying M = [M;v], and you can initialize M to be the empty matrix [].

Use ‘envmap2Dto3D’ to obtain e_xyz and e_rgb from E that you obtain in Sub-part 1.3. Create a struct ‘envmap’ with fields ‘vertices’ and ‘colors’ as

```
            envmap.vertices = e_xyz;
            envmap.colors = e_rgb;
```

Save the struct envmap to a .mat file ‘envmap3D.mat’.

Part 2 [10 points total]: Insert the 3D model of an object into your photograph

3D models for several objects are provided to you as various .mat file. You can load a .mat file into MATLAB (e.g., chair.mat) as

```
load chair.mat
```

Loading the .mat file will generate a struct with the same name as the object. In the case of chair.mat, a struct called chair is created in your workspace, with fields:

chair.vertices: an Nverts x 3 matrix of x, y, z locations for vertices
chair.faces: an Nfaces x 3 matrix of triangular faces, where each row points to three vertices for one triangle
chair.colors: an Nfaces x 3 matrix of colors, with one color per face. Each color represents a diffuse reflectance.

You will use this struct in the rest of this section.

Sub-part 2.1: Make the scene from Short Project 2 right-handed, and set up a struct for the floor plane

The scene that you reconstructed in Short Project 2 is in a left-handed coordinate system. However, to perform lighting computations using normals, we need to work with a right-handed coordinate system. The easiest way to do this is to negate the y coordinate for all planes in Short Project 2. E.g., for the floor plane, you would say

```
floorplane(:,2) = -floorplane(:,2);
```

To create shadows on the floor, you will need to have your floor plane set up as a struct containing the floor plane vertices and an average color for the floor plane. You can get the average color by first getting a mask image for the floor using the built-in function ‘roipoly’, by saying

```
R = roipoly( expanded_im, floorx, floory );
```

then using R to mask out non-zero pixels in expanded_im as

```
vals = reshape( expanded_im,[],3 );
vals = vals( R,: );
```

and finally computing the average color ‘floorplaneColor’ as the mean of vals (do a help on ‘mean’).

Set up a struct for the floor plane as

```
floorplaneStruct.vertices = floorplane;
floorplaneStruct.color = floorplaneColor;
```

Sub-part 2.2 [3 points]: Position the 3D model in the context of the 3D scene

To insert the 3D model into the image, we need to first position the 3D model in the context of the 3D scene behind the image. For this you should use the functions ‘scale3’, ‘rotate3’, and ‘translate3’ given to you as part of this assignment to move the vertices of the object. E.g.

```
      chair.vertices = scale3( chair.vertices, [1,.5, 2] );
```

scales the chair’s vertices by .5 in y, by 2 in z, and keeps them unchanged in x.

```
      chair.vertices = rotate3( chair.vertices, ‘y’, pi/6 );
```

rotates the chair by pi/6 radians or 30 degrees around the y axis.

At the very least, you will have to translate the chair in z, i.e., you will have to do

```
      chair.vertices = translate3( chair.vertices, [0, 0, tz] );
```

where tz is a large enough value to put the chair far from the camera so that you can see the chair. However, you will likely need to use all functions to set up your object correctly.

To visualize your 3D model in the context of your scene, you can use the ‘plotPlanes’ function provided in this code to plot your planes as

```
      plotPlanes( leftplane, ‘r-’, rightplane, ‘g-’,...
            ceilplane, ‘b-’, floorplane, ‘c-’, backplane, ‘y-’ );
```

You can then visualize the 3D model by using the built-in function ‘patch’ as

```
patch( ‘vertices’, chair.vertices, ‘faces’, chair.faces,...
      ‘facecolor’, ‘flat’, ‘facevertexcdata’, object.colors );
```

Save the struct containing the floor plane and the struct containing the transformed object vertices in a .mat file titled ‘geometry.mat’.

Sub-part 2.3 [3 points]: Perform ray tracing

To perform ray tracing within a reasonable amount of time, this assignment contains C++ code with the ray tracing computations (if we were to use pure old MATLAB, the code would take days and days to run). The code does diffuse or Lambertian illumination. The amazing thing about MATLAB is that it allows you to interface with C++ code using special files called MATLAB Executable (or MEX) files.

The main files containing code that can be converted into a MATLAB Executable are RayTracerMEX.cpp and RayTracerPlanelyMEX.cpp. These files need to be compiled. On Windows, MATLAB uses Visual Studio to compile code, while in a Unix based environment you can use gcc or Clang (I suspect you can use Clang for Windows as well).

If you have never used MEX before, you may first need to do

```
mex -setup
```

and walk through any instructions that come up.

To compile RayTracerMEX.cpp and RayTracerPlaneOnlyMEX.cpp, do:

```
mex RayTracerMEX.cpp BBox.cpp BVH.cpp
mex RayTracerPlaneOnlyMEX.cpp BBox.cpp BVH.cpp
```

The files use .cpp files BBox.cpp and BVH.cpp and the .h files Object.h, Ray.h, Vector3.h, Triangle.h, and IntersectionInfo.h.

The above function being in C++ could cause issues, that may cause MATLAB to crash. Before running the call below, please save all variables in your workspace by doing

```
save vars.mat
```

You can now use the structs ‘envmap’, ‘chair’, and ‘floorplaneStruct’ in the RayTracer.m wrapper function around the RayTracerMEX file, together with an additional struct ‘imageParameters’. Set up imageParameters as a struct as follows:

```
imageParameters.focalLength = f;
imageParameters.vanishingPoint = [vx, vy];
imageParameters.size = [height_of_expandedim, width_of_expandedim];
```

Here, f is the focal length from Short Project 2, vx and vy are the coordinates of the vanishing point, and height_of_expandedim and width_of_expandedim are the height and width of the expanded image from Short Project 2.

One issue that you will run into is that your image from Short Project 2 may be too large to do ray tracing in time. You may be better off resizing expandedim to be no more than 800-1000 pixels in the longest direction. If you resize expandedim, you must resize the parameters in imageParameters by the same amount. For instance, for a resize by 1/4, you would have

```
        expandedim = imresize( expandedim,1/4 );
        imageParameters.focalLength = f/4;
        imageParameters.vanishingPoint = [vx/4, vy/4];
        imageParameters.size = [height_of_expandedim, width_of_expandedim];
```

Here height_of_expandedim and width_of_expandedim are the height and width of the resized expanded image. **Write imageParameters to a file imageParameters.mat.**

Call the wrapper function RayTracer.m as

```
[IllumImage, ReflectImage, ObjectMask] = ...
        RayTracer( chair, floorplaneStruct, envmap, imageParameters );
```

IllumImage and ReflectImage refer respectively to the illumination image and reflectance image. ObjectMask refers to a mask with ones for pixel locations in the resized expanded image that contain the object.

You can obtain the shading (i.e., the render) by taking the element-wise product of the two images. However, to compensate for the fact that E is correct up to a multiplicative factor ‘alpha’, you will first need to scale the values in IllumImage down. Choose a scale factor ‘alpha’ that allows you to see the illumination well (including the shading and the shadows). For example, you may even need to scale by ‘alpha’ of 100 to 1000 before you see the image.

Similarly, call the wrapper function RayTracerPlaneOnly.m to get the background without the object for differential rendering as

```
[IllumImagePlaneOnly, ReflectImagePlaneOnly] = ...
        RayTracerPlaneOnly( floorplaneStruct, envmap, imageParameters );
```

and scale IllumImagePlaneOnly by the same value ‘alpha’.

Compute the render with the object (stored in a variable RenderWithObject) by performing element-wise multiplication on the scaled IllumImage and ReflectImage. Similarly, compute the render without the object (stored in a variable RenderWithoutObject) by performing element-wise multiplication on the scaled IllumImagePlaneOnly and ReflectImagePlaneOnly.

Write IllumImage, ReflectImage, IllumImagePlaneOnly, ReflectImagePlaneOnly, RenderWithObject, and RenderWithoutObject to png files with the same name as the corresponding variable. Write the expanded image to a file ‘Input.png’.

Sub-part 2.4 [4 points]: Implement and perform differential rendering

Implement a function to perform differential rendering with the following signature:

```
function Output = differentialRender( InputPhotograph, RenderWithObject, ...
      RenderWithoutObject, ObjectMask )
```

Use the function to obtain the final output by appropriately compositing the renders with and without the object into the input photograph using the object mask. The input photograph in this case should be the resized expanded image.

Write the final output to a file 'Output.png'.

Deliverables for Part 1 and Part 2
A .zip file containing the following:

- Function gsolve.m re-written to have sparse matrix computations (from Part 1, Sub-part 1.3)
- Function envmap2Dto3D.m (from Part 1, Sub-part 1.4)
- Function differentialRender (from Part 2, Sub-part 2.4)
- Script containing all calls in this assignment
- **Two separate folders (one per student) containing the following done for each student's photograph. You can work together for both folders. Name each folder by the name of the student.**
    - The folder 'originalSpherePhotographs' with all N original multiple exposure photos (from Part 1, Sub-part 1.1)
    - The folder 'croppedSphereImages' with all N cropped multiple exposure photos (from Part 1, Sub-part 1.2)
    - envmap2D_HDR.mat (from Part 1, Sub-part 1.3)
    - envmap3D.mat (from Part 1, Sub-part 1.4)
    - geometry.mat (from Part 2, Sub-part 2.2)
    - imageParameters.mat (from Part 2, Sub-part 2.3)
    - IllumImage.png, ReflectImage.png, IllumImagePlaneOnly.png, ReflectImagePlaneOnly.png, RenderWithObject.png, RenderWithoutObject.png (from Part 2, Sub-part 2.3)
    - Output.png (from Part 2, Sub-part 2.4)