COMP 70001 Spring 2021 Assignment 2

George Castle Zhiyuan Zhang
Email:{george.castle20, zhiyuan.zhang17}(@imperial.ac.uk)

1. Part 1: Generate plots of Fresnel reflectance

George Castle did the first half of this coursework, which includes part 1 and part 2.

Part 1 discusses the Fresnel reflectance of dielectric materials and compares the unpolarised Fresnel reflectance against Schlick's approximation.

1.1. Fresnel Parallel and Perpendicular Polarisation

This implementation uses Python and matplotlib to generate the plots. The process for creating these plots is described in the following steps:

1. For loop iterates through 0-90 degrees of incidence, converting each measurement to radians as python trigonometric functions use radians by default. The following calculations are then conducted for each integer degree of incidence.

2. Snell's Law rearranged to calculate the angle of transmission using the angle of incidence, the refractive index of the air ($\eta_i = 1.0$) and the dielectric material ($\eta_t = 1.45$)

$$\arcsin(\frac{\eta_i \sin\theta_i}{\eta_t}) = \theta_t$$

3. The polarised light reflectance was then calculated and stored in an array for the parallel ($r_\parallel = R_p$) and perpendicular ($r_\perp = R_s$) components respectively using the following Fresnel equations:

$$r_\parallel = \left|\frac{n_i\cos\theta_t - n_t\cos\theta_i}{n_i\cos\theta_t + n_t\cos\theta_i}\right|^2 \quad r_\perp = \left|\frac{n_i\cos\theta_i - n_t\cos\theta_t}{n_i\cos\theta_i + n_t\cos\theta_t}\right|^2$$

   Each result was then adjusted to be a percentage value as shown in the examples in the slides.

4. Outside of the for loop the Brewster's angle is calculated using the following rearranged formula:

$$\theta_B = \arctan\frac{\eta_t}{\eta_i}$$

   This is the incidence angle at which the parallel polarised light ($r_\parallel = R_p$) goes to zero and the refracted light is perfectly polarised.

5. The reflectance of both components at normal incidence is then found by looking at the first entry in each array. These measurements are then added to a plot using matplotlib producing the following result:

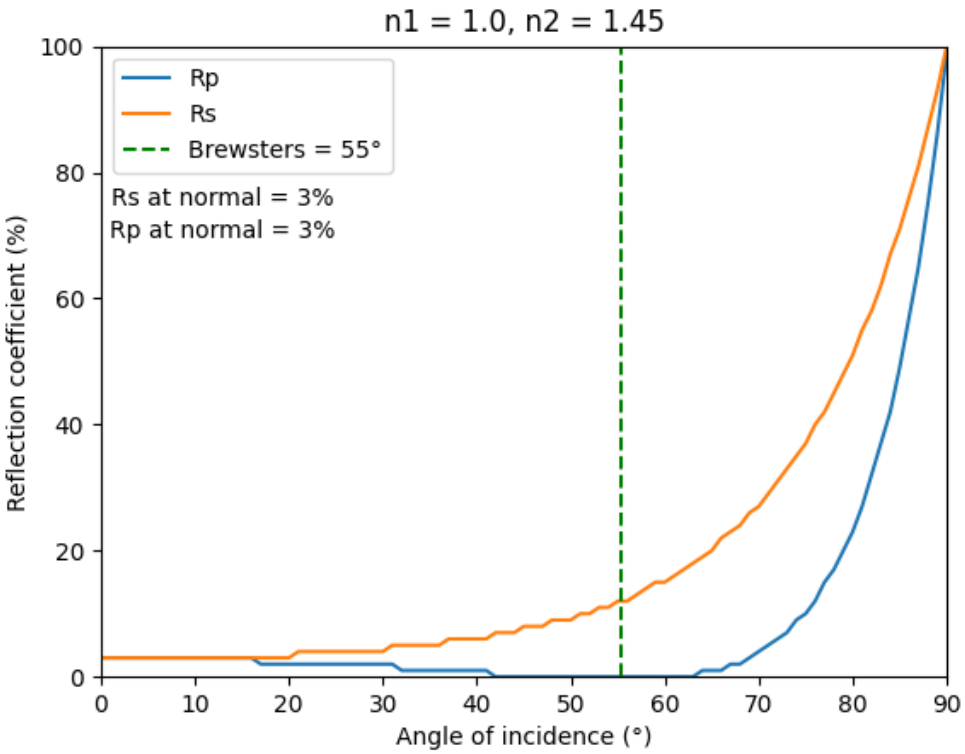


Figure 1: Fresnel reflectance from air ($\eta_i = 1.0$) to material ($\eta_t = 1.45$)

1.2. Fresnel reflectance for ray exiting the material into air ($\eta_t = 1.0$)

The same process was used as in the previous section (1.1) with some small adjustments:

1. The critical angle is calculated first using the following formula:

$$\theta_C = \arcsin \frac{\eta_t}{\eta_i}$$

   This is the angle at which all light is reflected. $(R_s = R_p = 1)$

2. The critical angle is then used to index the arrays and for loop as beyond the critical angle there is total internal reflection and Snell's law predicts that the sine of the angle of refraction would exceed one.

3. The parallel and perpendicular components were then calculated and stored in arrays using the same algorithm and equations as in the previous section.

4. The Brewster's angle was also calculated using the same formula as in the previous section. As an integer index was used for the arrays and for loop the critical angle and Brewster's angle were approximated to the nearest integer degree.

5. Adding these values to a plot using matplotlib produced the following result:

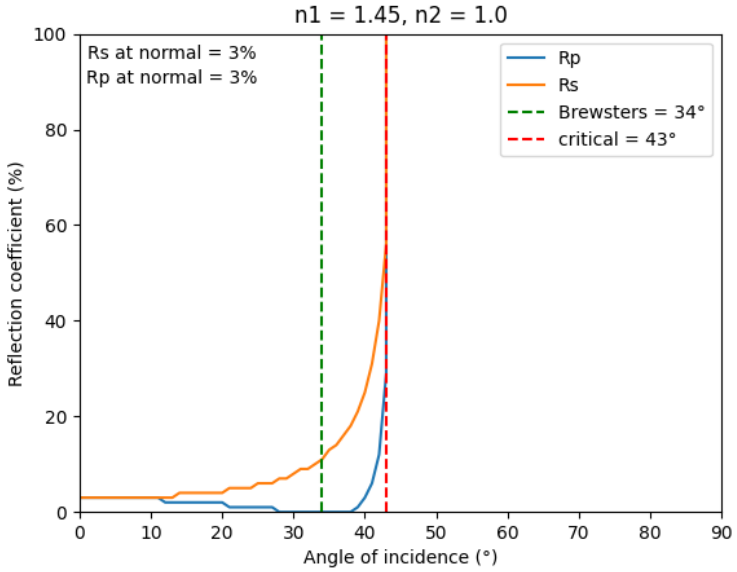


Figure 2: Fresnel reflectance from material ($\eta_i = 1.45$) to air ($\eta_t = 1.0$)

1.3. Schlick's approximation of Fresnel reflectance

1. Using the same algorithm to calculate the Fresnel reflectance from the first section (1.1) I generated the Fresnel reflectance for unpolarised light at normal incidence (R_0) by averaging the first entry of the parallel and perpendicular component arrays using the following the formula from the slides:

$$F_r = \frac{1}{2}(R_p + R_s)$$

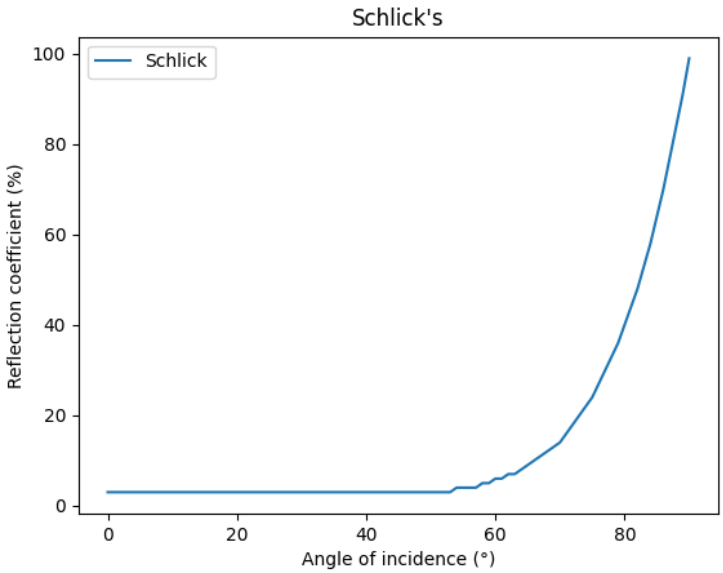
2. Using this value for reflectance at normal incidence Schlick's polynomial approximation of unpolarised Fresnel reflectance was calculated for each integer angle of incidence in the for loop and stored in an array using the following formula:

$$F_r(\cos\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$$

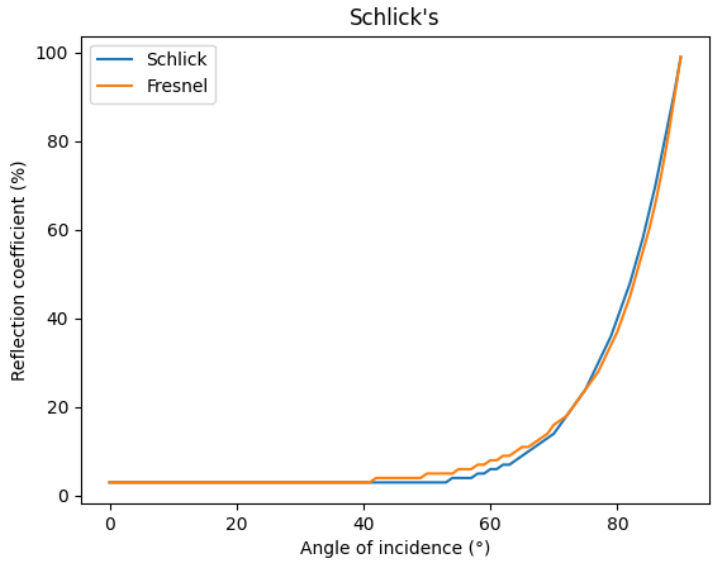
   Adjusted to be a percentage value as in the previous sections.

3. I also calculated the unpolarised light using the $F_r = \frac{1}{2}(R_p + R_s)$ equation for the rest of the incidence angles to compare to Schlick's approximation. The comparison is shown on a separate plot below

4. Adding the Schlick's approximation values to a plot using matplotlib produced the following result:



(a) Schlick's polynomial approximation of unpolarised Fresnel reflectance



(b) Schlick's approximation compared with unpolarised Fresnel reflectance

Figure 3: Plots of Schlick's approximation

2. Part2: Generate MC samples according to an Environment Map

Part 2 discusses the generation of Monte Carlo samples according to the probability density function and cumulative distribution function of the intensity values of the Grace Cathedral Environment Map pixels.

2.1. Generate samples according to the 2D PDF and CDF of the EM

To load the Grace Cathedral EM I used a function provided for the first coursework $(loadPFM())$ to store the image as a numpy array. I then wrote a function to clamp the PFM RGB array values that were above 1 to a value of 1 to prevent pixel intensities greater than 1. After which I used two nested for loops to iterate through every (x, y) pixel position in the array. For each pixel the intensity (luminance) is calculated using the provided formula:

$$I = \frac{(R + G + B)}{3}$$

This intensity is stored in a new array to be used to create the CDF for individual pixels later in the algorithm. The intensity for every pixel in each row is accumulated within the inner for loop that iterates through each x value. The accumulated value is then stored in an array to create a 1D vertical PDF of every scan line (row) intensity total. This PDF array can then be normalised to values between 0 and 1 using the following formula:

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

This normalised PDF can be displayed using matplotlib:

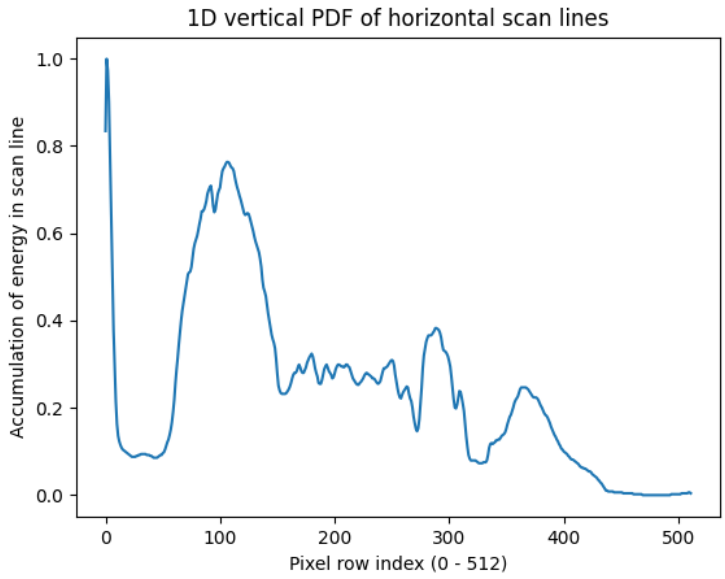


Figure 4: 1D vertical PDF across all scan lines with no sine modulation

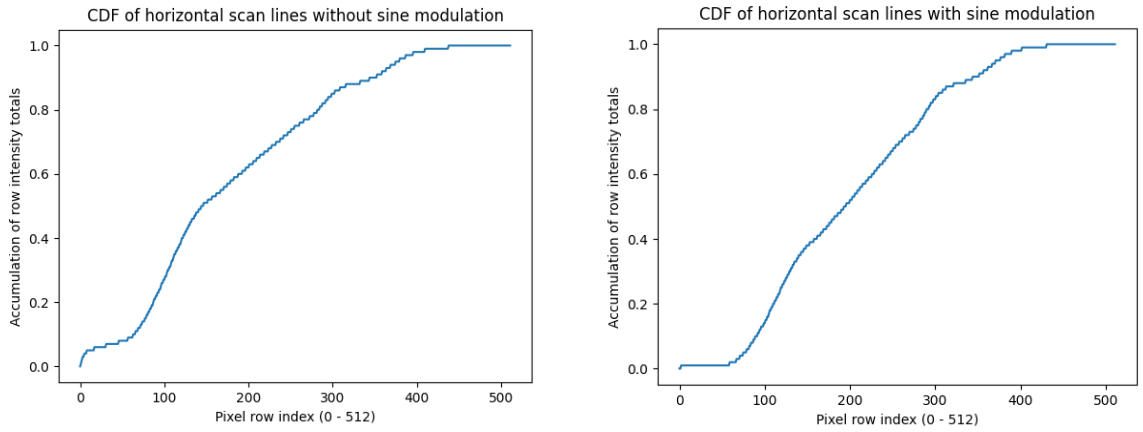
I then used another for loop to iterate through the every vertical y value in the PDF array accumulating a summation of each value in the PDF to create a cumulative distribution function of the 1D PDF. However

as the Grace Cathedral EM is a latitude-longitude Environment Map we must include $\sin\theta$ modulation to prevent oversampling of the poles of the image due to the poles being stretched across the whole of the top and bottom rows of the lat-long map. I implemented this in my algorithm by converting the for loop index (y coordinate) to its spherical coordinate θ using the following formula:

$$\theta = \frac{y}{y_{max}}\pi$$

This formula converts the y coordinate to a $0 - \pi$ range.

This θ value could then be used to calculate the $\sin\theta$ modulation by multiplying each PDF array value at index y by its respective $\sin\theta$ at index y before it is accumulated in the CDF. Below is a figure showing the CDF with and without $\sin\theta$ modulation.



(a) CDF of all rows without $\sin\theta$ modulation

(b) CDF of all rows with $\sin\theta$ modulation

Figure 5: Plots of vertical CDF of scan lines

As the figure shows there is less importance given to the poles (values at rows near 0 and near 512) when using $\sin\theta$ modulation. Despite these figures being normalised to a 0 to 1 range I normalised the CDF to a 0 to 100 range to make the random number generation and inverse CDF indexing simpler to implement using Python and to avoid the usage of small floating point numbers.

To generate samples from the CDF array I created random numbers using Python's random library. As the required sampling rates all had integer square roots I used the square root for the amount of samples to take from the vertical CDF. For example if the sample rate is 64 the algorithm generates 8 random numbers to sample the first CDF.

The $n = \sqrt{samples}$ random numbers generated were between 0 and 100 to match the normalisation of the CDF. I then created a function to find the nearest value to the random number stored in the CDF array and return the index position of that number. These returned numbers are the selected rows for the second PDF generation. The Algorithm then generates n new PDF arrays using the pixel intensity value numpy array generated in the first for loop. These n arrays are the horizontal PDFs for each of the selected rows.

The normalised CDF for each row is then calculated in the $CreateCDFofRow()$ function, by summing all the previous values of the PDF for each CDF entry. Another n random numbers are then generated for each of the n rows totaling the desired number of samples: $\sqrt{samples} * \sqrt{samples} = samples$. These samples are then inversely indexed in to the CDF they were generated for using the same function as before returning the index of the sampled pixels.

These values are stored in a numpy array with both the $x(\phi)$ and $y(\theta)$ index of the pixels to be sampled. The algorithm then creates a blue $5x5$ square around each of the pixels in a copy of the original array and creates a $5x5$ pixel window with the original PFM values in a numpy array of all black pixels for the result of part 2.3. Each resulting numpy array is then written to their respective out paths.

2.2. Write out the EM (.pfm) with the pixels corresponding to the chosen sample directions set to Blue

(a) 256 samples PFM (b) 256 samples PPM with gamma value 2.1

Figure 6: Sampled environment map

Please see Appendices Figure 10 for different sampling rates and Appendices Figure 11 for different gamma values. I chose gamma value 2.1 as it produced the closest result to the output PFM.

2.3. 256 MC samples RGB values

(a) 256 RGB samples PFM (b) 256 RGB samples PPM with gamma value 2.1

Figure 7: Sampled environment map RGB samples

Above is the RGB samples for the 256 MC PDF and CDF algorithm. Once again I found a gamma value of 2.1 to be the best match for the PFM output image with no scaling necessary.

3. Part3: Median Cut

The second half includes part 3 and part 4 is done by Zhiyuan Zhang.

The part 3 of this coursework is to implement the median cut algorithm proposed in [1]. The functions are included in 3 different files-preprocessing.py, utilities.py and image_processing.py.

Median Cut algorithm is a method to sample the environment map to reduce the computational cost. The main idea is, same as stated both in lecture and Debevec's paper, cropping the image with equal total energy and representing the light sources at its radiance centroid.

3.1. Preprocessing

Utilities.py are the load and save image functions. It included two versions, one using OpenCV library, and the second version uses the C library used from assignment1. Since this coursework's primary concern is not bit-wise manipulation of images, these functions are not discussed in detail in this report.

The preprocessing.py broadcasts the image's dimension from 512*1024*3 to 512*1024*6 and save the NumPy array as img.npy. This process takes 25 seconds which is quite long. Saving this preprocessed image could accelerate future reuse by directly loading it to memory. The additional dimensions saved the x, y coordinates from the uncut image, and the last channel saved that pixel's energy, which is the scaled average of the RGB value. Since the environment map is mapped from rectangle to sphere, the energy need to be scaled by $\sin(\theta)$, where θ is from 0 to π from bottom to top. The implementation generates a sequence from 0 to π and maps each pixel by its y coordinate.

3.2. Median Cut Algorithm

The computational trade-off made for this implementation is to sacrifice space for time. However, since this is not relevant to the course's topic, a brute force algorithm is still used, and no sophisticated optimisation is performed to implement this coursework.

The main idea of the implementation is to cut images into smaller pieces recursively. For example, after 6 cuts, the resulted 64 images will be stored in a list. The cutting is performed along the image's longer edge, where the cropped images should contain equal total energy. The implementation is straightforward. Iterate through the longer edge and compare the total energy of two regions. Finally, return positions of the two sub-images with the closest total energy. Then perform the cut and return the two

sub-images. The pseudo code for search algorithm is shown in Algorithm 1.

Algorithm 1: find_corp_position

```
init MinDifference = infinite
init CutPosition = -1
CheckLongerEdge(image)
if y is longer then
    TotalEnergyAlongY = SumAlongColumn(images)
    for each split of TotalEnergyAlongY do
        if differnce of two sub array < MinDifference then
            Update(MinDifference, CutPosition)
        end
    end
end
else
    TotalEnergyAlongY = SumAlongRow(images)
    for each split of TotalEnergyAlongY do
        if differnce of two sub array < MinDifference then
            Update(MinDifference, CutPosition)
        end
    end
end
return CutPosition
```

The goal is to find the centroids of the sub-images. The method is to find the cut positions along both edges, in which the four regions divided by the centroid has similar total energy. The centroids of subimages need to added by the x, y coordinates of the first pixel in the subimage to map it to the correct location in EM.

The final part is to draw the centroid and edges on the environment map. With the help of the original x, y coordinates, it is easy to find the edges. The pseudo code is very clear about the process in Algorithm 2.

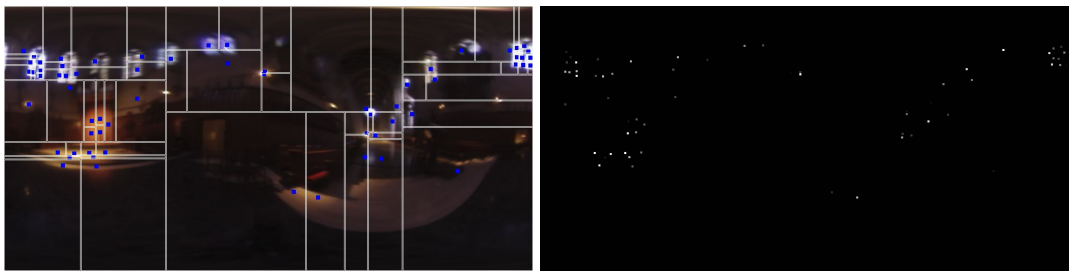
Algorithm 2: Colour for Illustration

```
for each subimage do
    PositionX, PositionY = find_corp_position(subimage)
    Centroid = Positions + FirstPixelofSubimage.xy #xy is the coordinates from original image
    DrawCenter(EM, Centroid)
    DrawEdge(EM, EdgesOfSubimage.xy)
end
```

For the Sampled 64 regions of environment map, it is just draw the centroids on a black image. For each centroid, its colour is the total energy of each region. The total energy is 178091. Since the EM is divided into 64 partitions, for each partition, it should contain approximately 2782. To print the energy, we see that all the total energy is around 2800, which is what we expected.

The result of 64 regions and the sampling EM is also shown in Figure 8b. Other images are included in the Appendices Figure 12. The calculation is used the provided HDR pfm file. However, use one stop scaling and gamma 3 to generate an LDR ppm file for illustration.

Pseudo code for main body of median cut can also be found in Appendices 3.



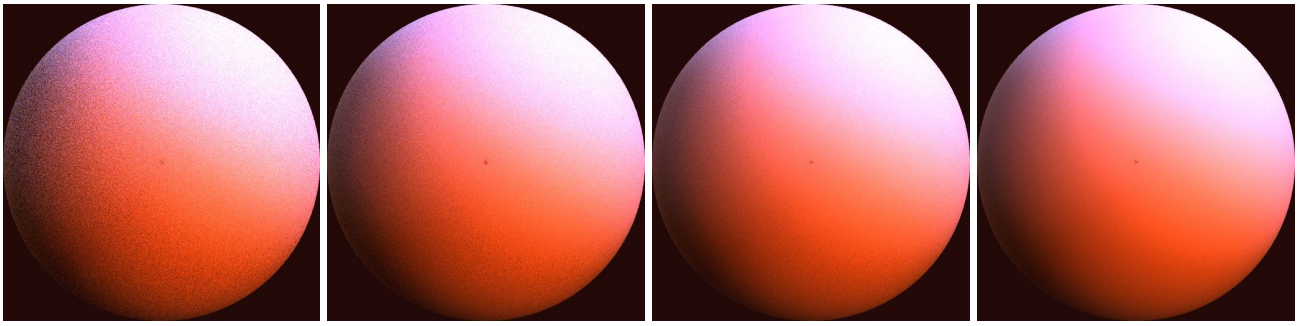
(a) median cut (b) 64 samples

Figure 8: Sampled environment map

4. Part4: PBRT sampling

Pbrt is a rendering engine designed for *Physically Based Rendering: From Theory to Implementation* [2]. The requirement in this coursework is simply to change the sampling rate of EM and see the results.

Part 4 uses the pbrt framework to obtain the rendered sphere using an environment map of sampling rates from 8 to 64. Visualising the HDR image using HDRShop, the default setting is overexposed. So, the F stop setting is changed to -2.900 and gamma is set to 1 for a better result. The original pfm and ppm files are also saved. Furthermore, the result is shown below. The conclusion is that a larger sampling rate could provide a more smooth surface while increasing the computational time.



(a) 8 samples per pixel (b) 16 samples per pixel (c) 32 samples per pixel (d) 64 samples per pixel

Figure 9: Rendered images by Grace Cathedral environment map

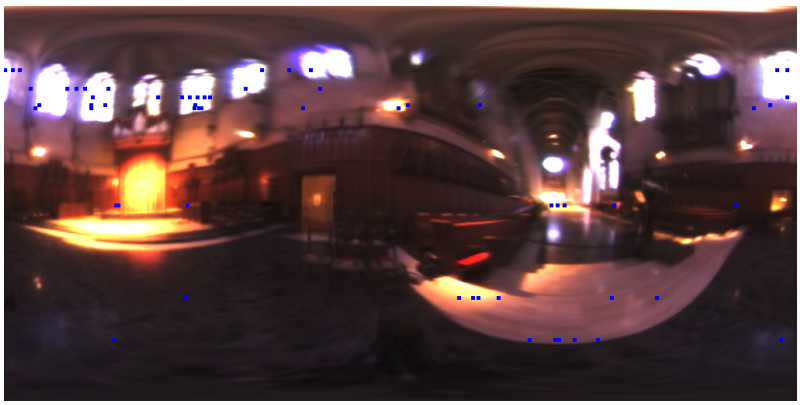
References

[1] P. Debevec. 2005. ”A median cut algorithm for light probe sampling”. In ACM SIGGRAPH 2005 Posters (SIGGRAPH ’05). Association for Computing Machinery, New York, NY, USA, 66–es.

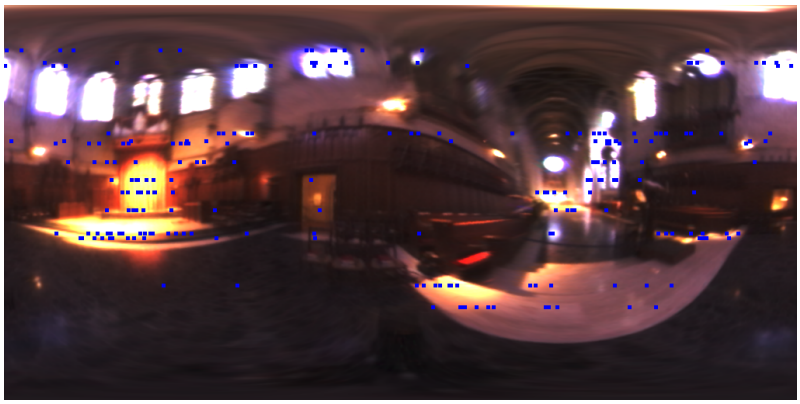
[2] M. Pharr, W. Jakob, and G. Humphreys. ”Physically Based Rendering: From Theory to Implementation”

Appendices

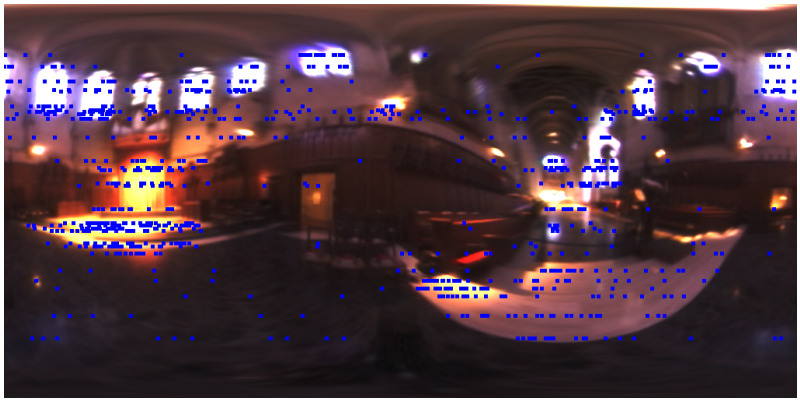
A. MC EM samples in blue



(a) 64 samples



(b) 256 samples

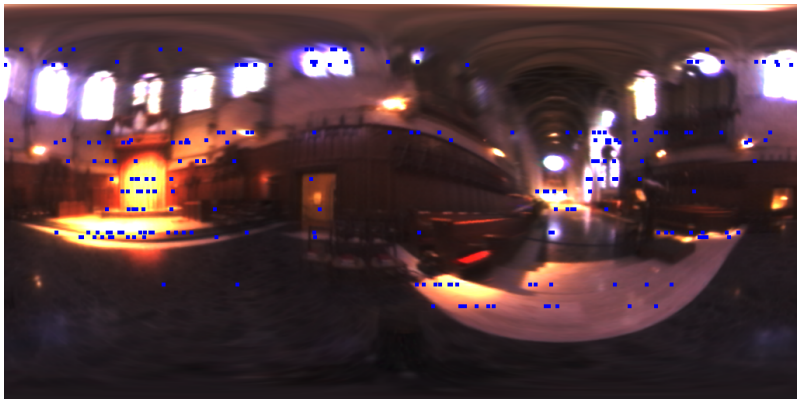


(c) 1024 samples

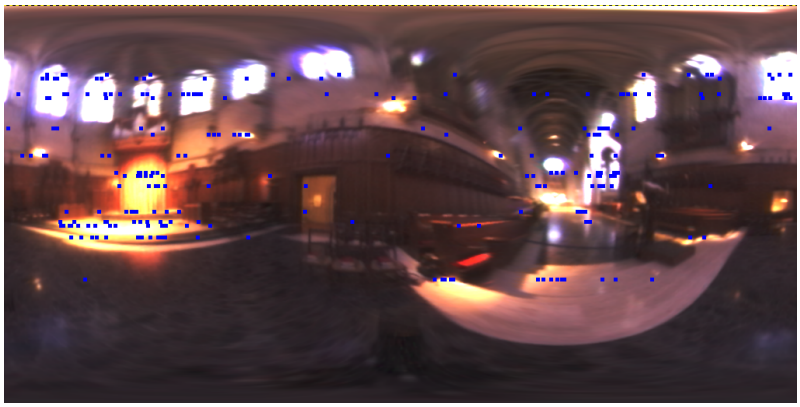
Figure 10: MC samples with different sampling rates

B. PPM gamma values

(a) gamma value 1 (no gamma correction)



(b) gamma value 2.1

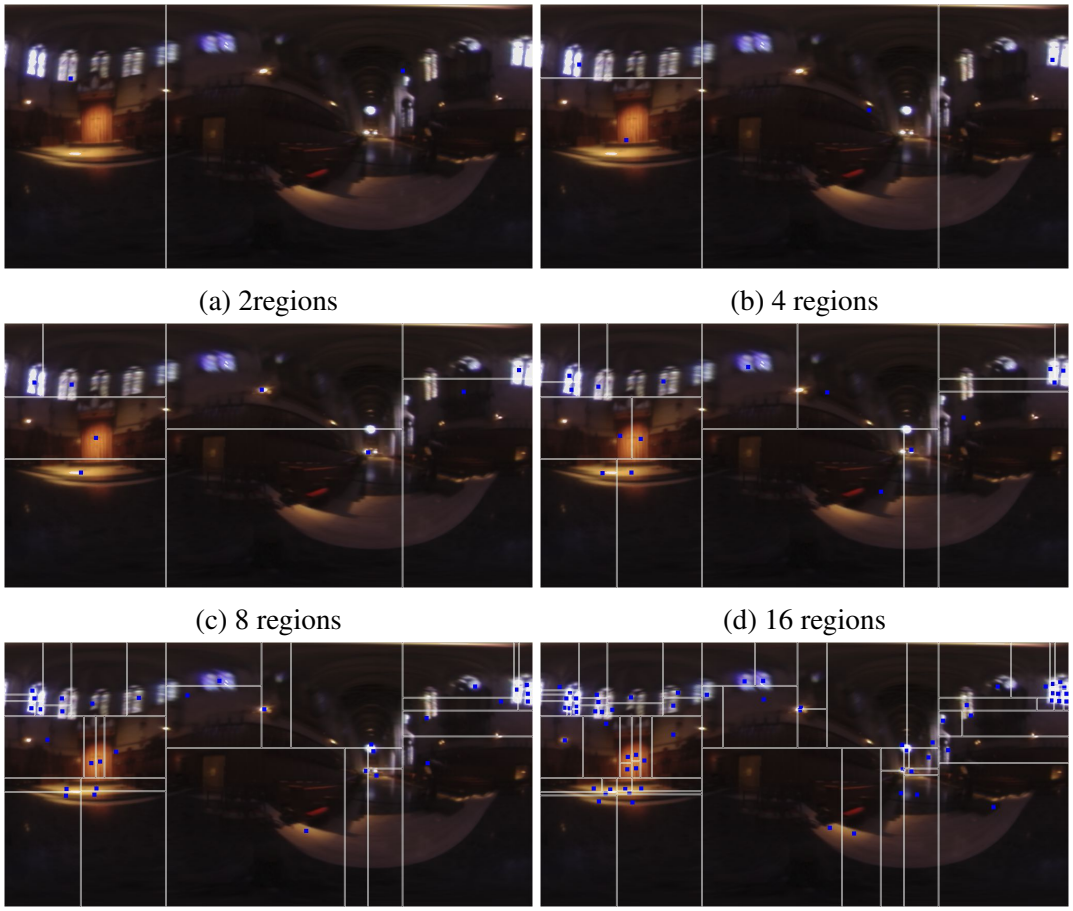


(c) gamma value 2.5

Figure 11: Different Gamma values for PPM

C. Median Cut

C.1. Images



(a) 2regions (b) 4 regions

(c) 8 regions (d) 16 regions

(e) 32 regions (f) 64 regions

Figure 12: Median cuts

C.2. Pseudo Code

Algorithm 3: Median Cut algorithm

```
for number of cuts do
    for ecah images do
        x,y = find_corp_position(image)
        subimages.add(corp(image,x,y))
    end
    output_plots(subimages)
    #reset for recursive
    images = subimages
    subimages = []
end
```