High Dynamic Range Images
Programming Assignment 1
Vítor Hugo Magnus Oliveira - 00341650

Task 1:

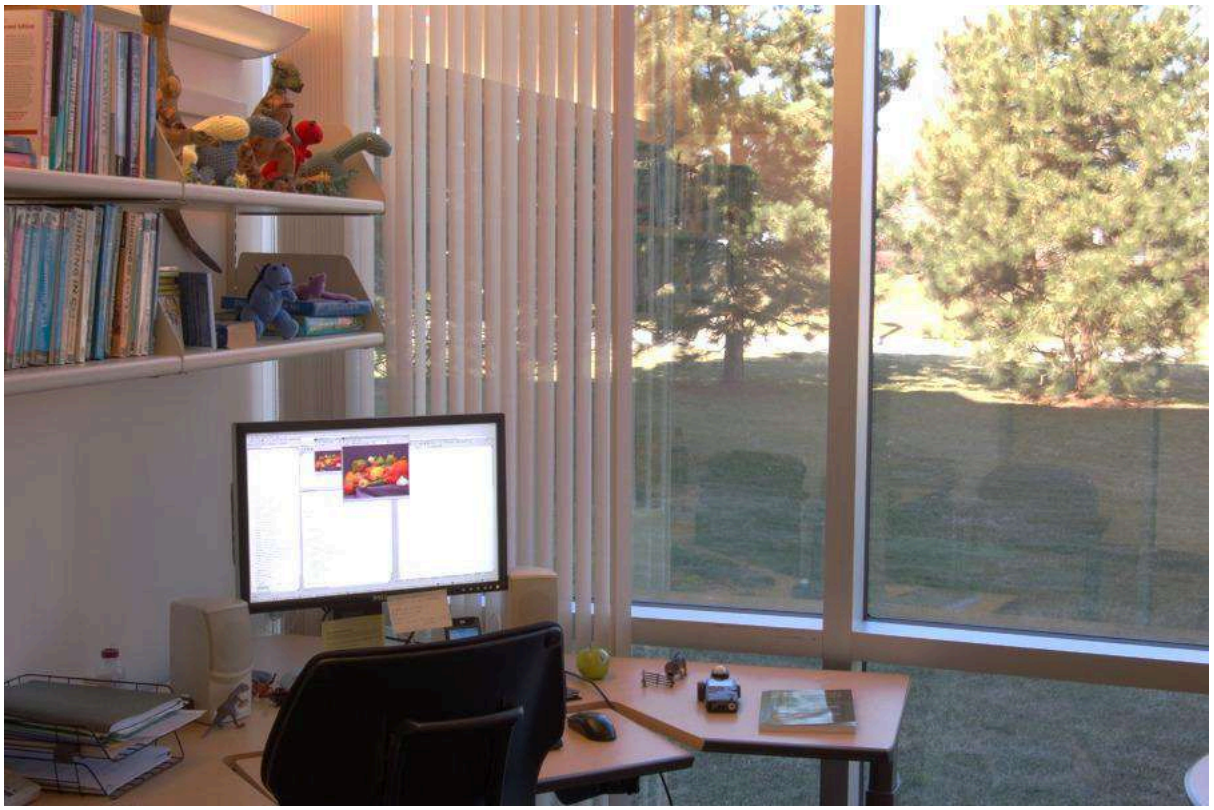
office_1.bmp : 0.0333s
office_2.bmp : 0.1s
office_3.bmp : 0.33s
office_4.bmp : 0.63s
office_5.bmp : 1.3s
office_6.bmp : 4s

Task 2:

Ok.

Task 3:

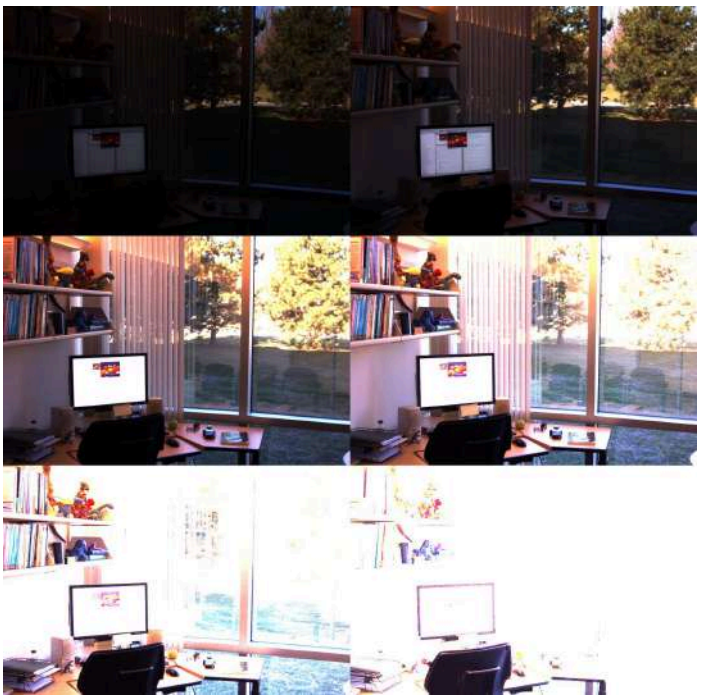
Imagem HDR gerada no HDR Shop:



Task 4:

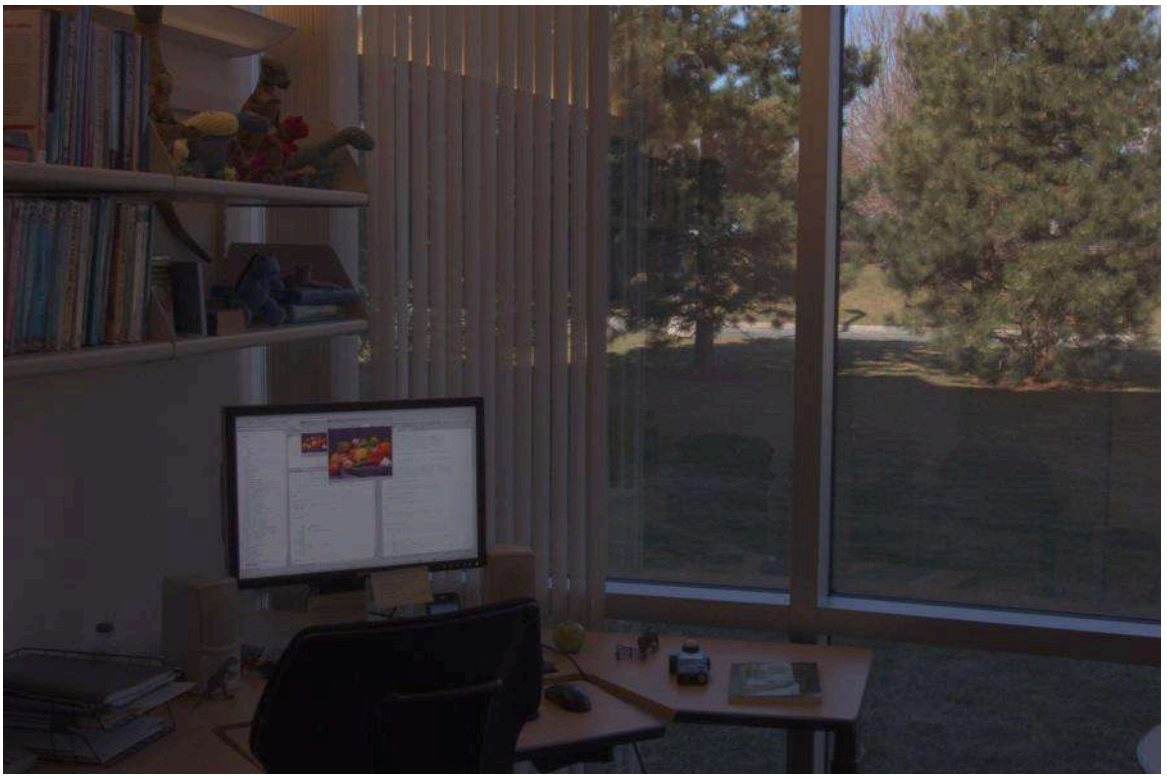
Foi criado um programa em Python com auxílio das bibliotecas cv2, numpy, math e csv. Primeiro é feita a leitura das 6 imagens LDR para uma lista. Logo em seguida outra leitura é feita, mas dessa vez é do arquivo txt que armazena os valores da curva de resposta da câmera que foram gerados no HDR Shop. Ainda nesse processo de inicialização, é definida mais uma lista que contém os tempos de exposição de cada imagem. Depois disso, é feito o cálculo da irradiância, que foi implementado na função 'get_irradiance'. Também foi desenvolvida a função 'get_irradiance_opt', que usa ferramentas de NumPy para realizar a mesma operação de uma maneira otimizada.

Resultados da multiplicação da irradiância pelos tempos de exposição (equivalente ao imtool do MATLAB) da **Task 1**:

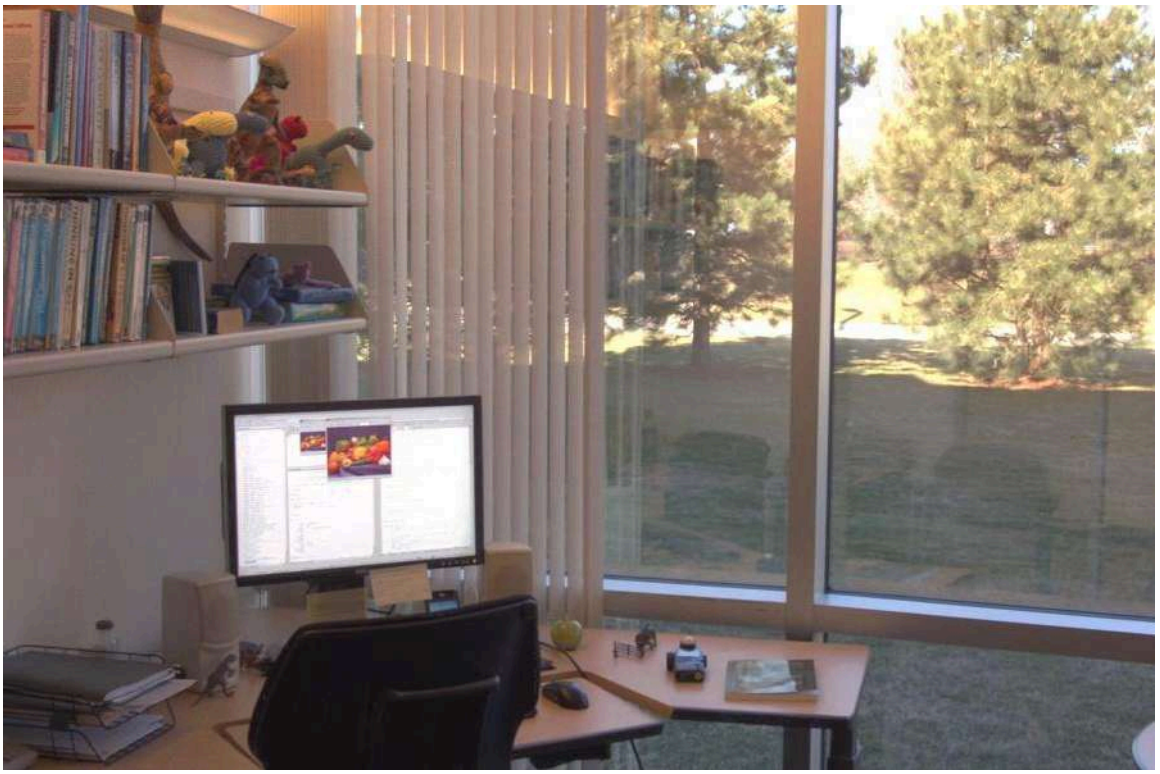


A função 'cv.createTonemap()' (equivalente ao tonemap do MATLAB) foi usada para criar um mapeamento linear. Ao processarmos esse tonemap na imagem HDR, temos:

Se aplicarmos gamma ao criar o tonemap, executando 'cv.createTonemap(gamma=2.2)' temos uma imagem com correção gamma:



Ao aplicar a operação de white balance que foi desenvolvida no último trabalho de implementação na imagem com correção gamma, o resultado é extremamente semelhante a imagem gerada na **Task 3** (Task 4 / Task 3):

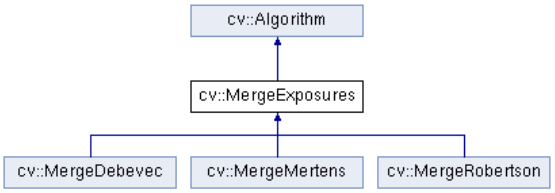


Task 5:

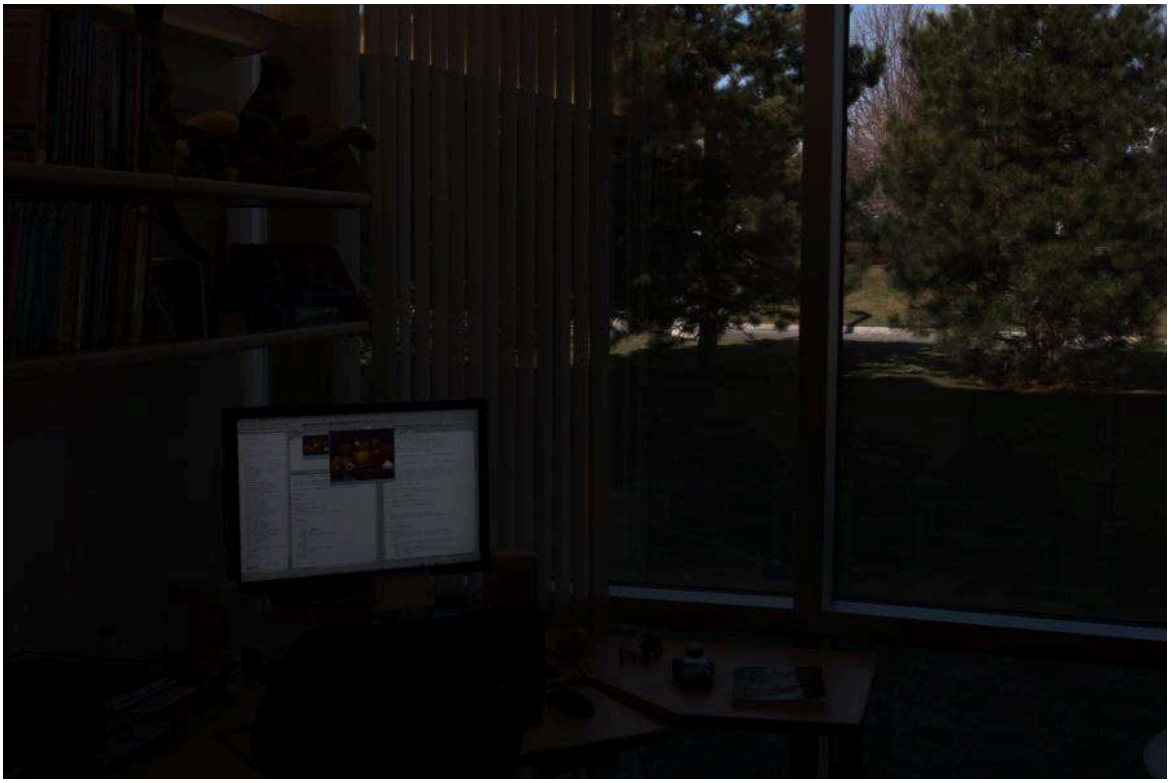
Para simular o efeito do makehdr do MATLAB, foi desenvolvida a seguinte função:

```
def makehdr(img_list, exposure_times, gamma=1.0):
    merge_debvec = cv.createMergeDebevec()
    hdr = merge_debvec.process(img_list, times=exposure_times)

    tonemap = cv.createTonemap(gamma=gamma)
    ldr = tonemap.process(hdr)

    ldr_8bit = np.clip(ldr * 255, 0, 255).astype(np.uint8)

    return ldr_8bit
```

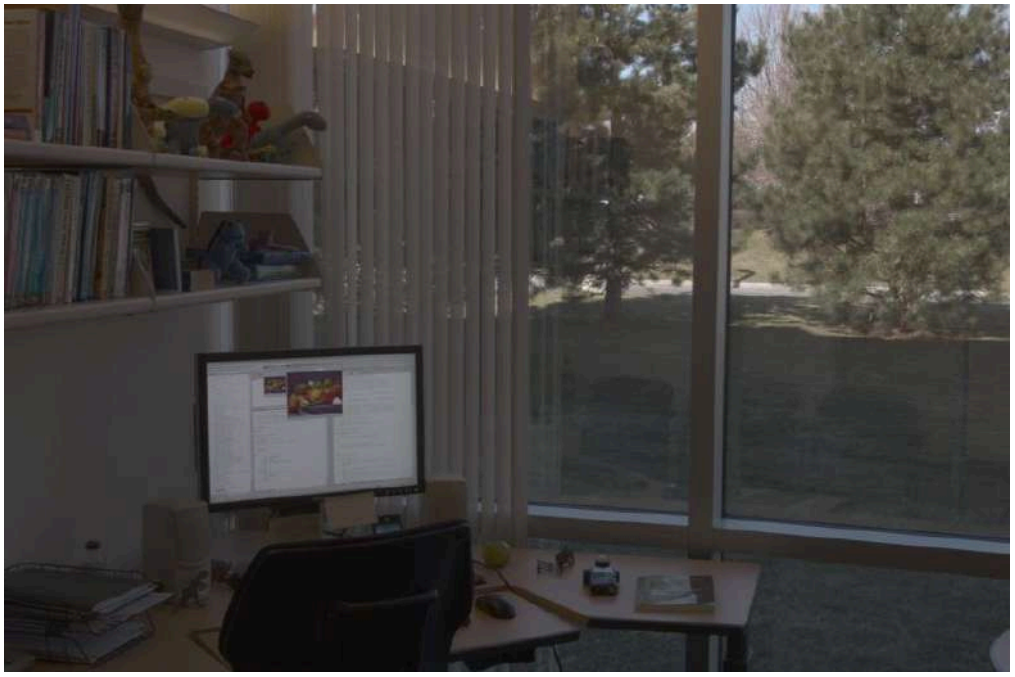
É usado o algoritmo de Debevec para calcular a curva da resposta da câmera e o método MergeExposures para realizar o cálculo da irradiância.



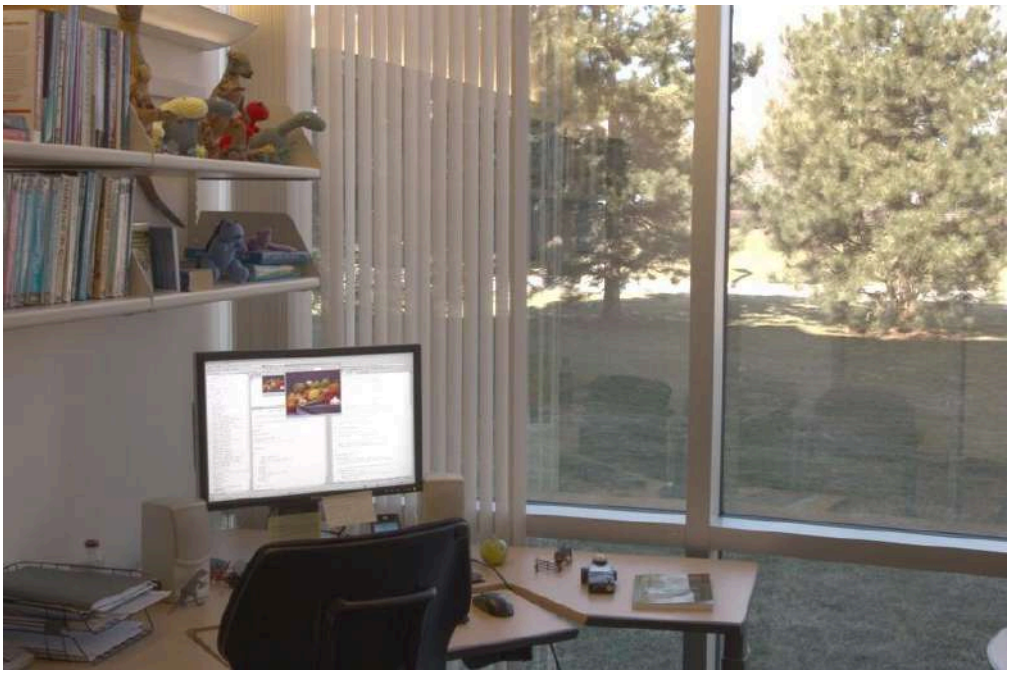
Depois disso, é aplicado o algoritmo de Tone Mapping Linear para simular o comando tonemap do MATLAB.A imagem resultante dessa função com gamma=1.0 é:



Com gamma=2.2 temos:



E ao aplicar white balance na imagem com gamma=2.2:



As imagens agora exibem uma saturação reduzida em comparação com os resultados da **Task 4**. A diferença entre os métodos de geração de imagens está na geração da curva de resposta da câmara e no cálculo da irradiância. Embora ambas as implementações compartilhem os mesmos algoritmos como base, suas abordagens permanecem distintas.

Task 6:

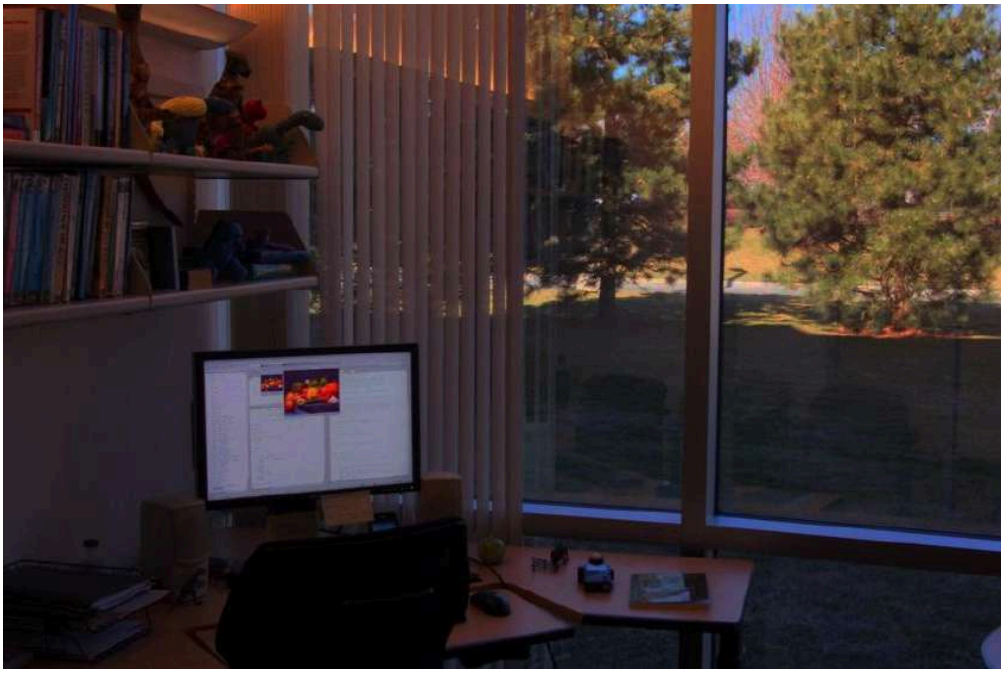
A implementação do algoritmo de Tone Mapping de Reinhard é, relativamente, simples. Entretanto, tive dificuldades em aplicar a nova luminância na imagem. Tentei duas abordagens diferentes:

- Converter para o espaço de cores Yxy e trocar o valor de Y.
- Converter para o espaço de cores Lab e trocar o valor de L.

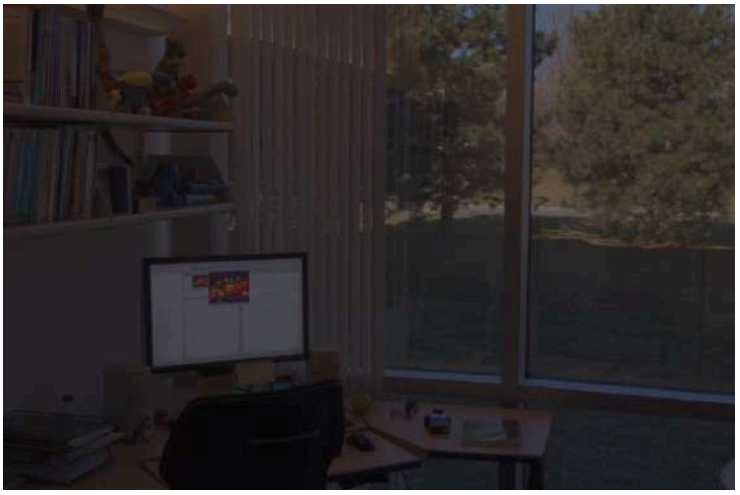
Falhei nas duas tentativas, por algum motivo as cores, ao voltar para o espaço sRGB, estavam completamente alteradas. A alternativa que encontrei foi multiplicar os três canais RGB pela razão da iluminância calculada pelo algoritmo de Reinhard pela luminância original da imagem:

```python
def tonemap_reinhard(img, gamma=1.0):
    img_processed = (img/255.0)**(1/gamma)
    luminance = (0.299*img_processed[...,0] + 0.587*img_processed[...,1] + 0.114*img_processed[...,2])

    n = luminance.size
    delta = 0.00001
    alpha = 0.18

    # L~ = exp((1/N) * sum(log(L[x,y]) + delta))
    avg_log_luminance = np.exp((1/n) * np.sum(np.log(luminance + delta)))

    # Ls[x,y] = (alpha/L~) * L[x,y]
    scaled_luminance = (alpha/avg_log_luminance) * luminance

    # Lg[x,y] = Ls[x,y] * (1 + Ls[x,y] / Lw[x,y]^2) / (1 + Ls[x,y])
    #global_luminance = scaled_luminance * (1 + scaled_luminance / scaled_luminance.max()**2) / (1 + scaled_luminance)

    # Lg[x,y] = Ls[x,y] / (1 + Ls[x,y])
    global_luminance = scaled_luminance  / (1 + scaled_luminance)

    global_luminance_ratio = global_luminance / luminance

    img_processed[:,:,0] = img_processed[:,:,0] * global_luminance_ratio
    img_processed[:,:,1] = img_processed[:,:,1] * global_luminance_ratio
    img_processed[:,:,2] = img_processed[:,:,2] * global_luminance_ratio

    return (img_processed*255).astype(np.uint8)
```

Os resultados foram satisfatórios, mas não foram alcançados pelos métodos que estava originalmente esperando. Imagem com gamma=1.0

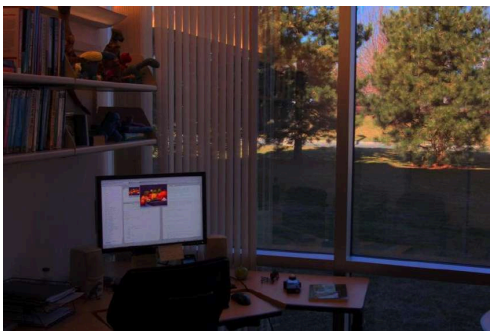


Resultado com gamma=2.2:



Aplicando white balance na imagem com gamma=2.2:

A imagem que mais se destacou em termos de diferença em relação à sua equivalente na Task 4 foi aquela com gamma=1.0. É evidente que essa imagem possui consideravelmente mais brilho e saturação (Task 6 / Task 4):



Por outro lado, as outras imagens, com gamma=2.2 e gamma=2.2 + balanço de branco, são extremamente semelhantes com as da Task 4.

Task 7:

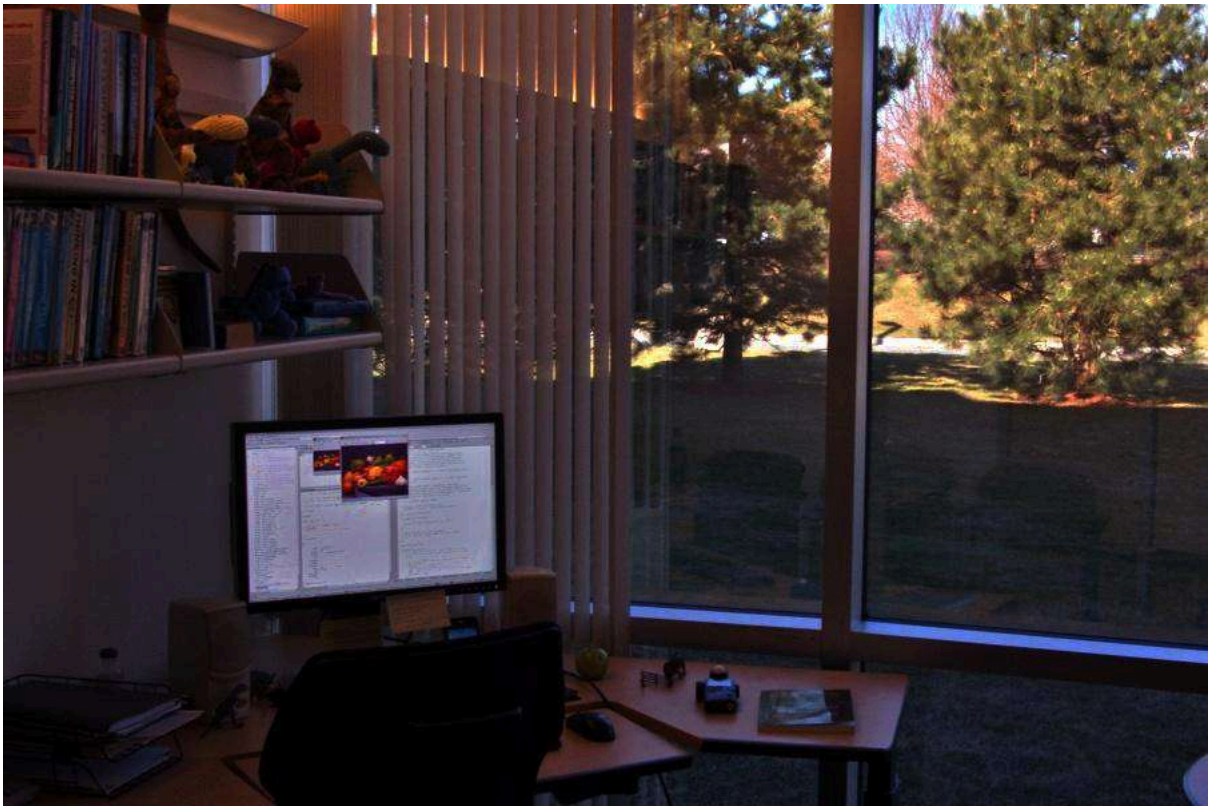
Para a implementação da versão local do algoritmo de Reinhard, foi preciso de somente uma pequena alteração no cálculo da iluminância global:

```
185     loop = True
186     while loop:
187         k_size += 2
188         weight = center_surround(scaled_luminance, k_size, key=alpha)
189         print(f'abs(max(weight)) = {np.max(np.abs(weight))}\nthreshold: {threshold}')
190         loop = np.any(np.abs(weight) > threshold)
191
192     # V[x,y,s(max)] = Ls[x,y] (x) W[x,y,s(max)]
193     gaussian = cv.GaussianBlur(scaled_luminance, (k_size, k_size), 0)
194
195     # Lg[x,y] = Ls[x,y] / (1 + V[x,y,s(max)])
196     global_luminance = scaled_luminance / (1 + gaussian)
```

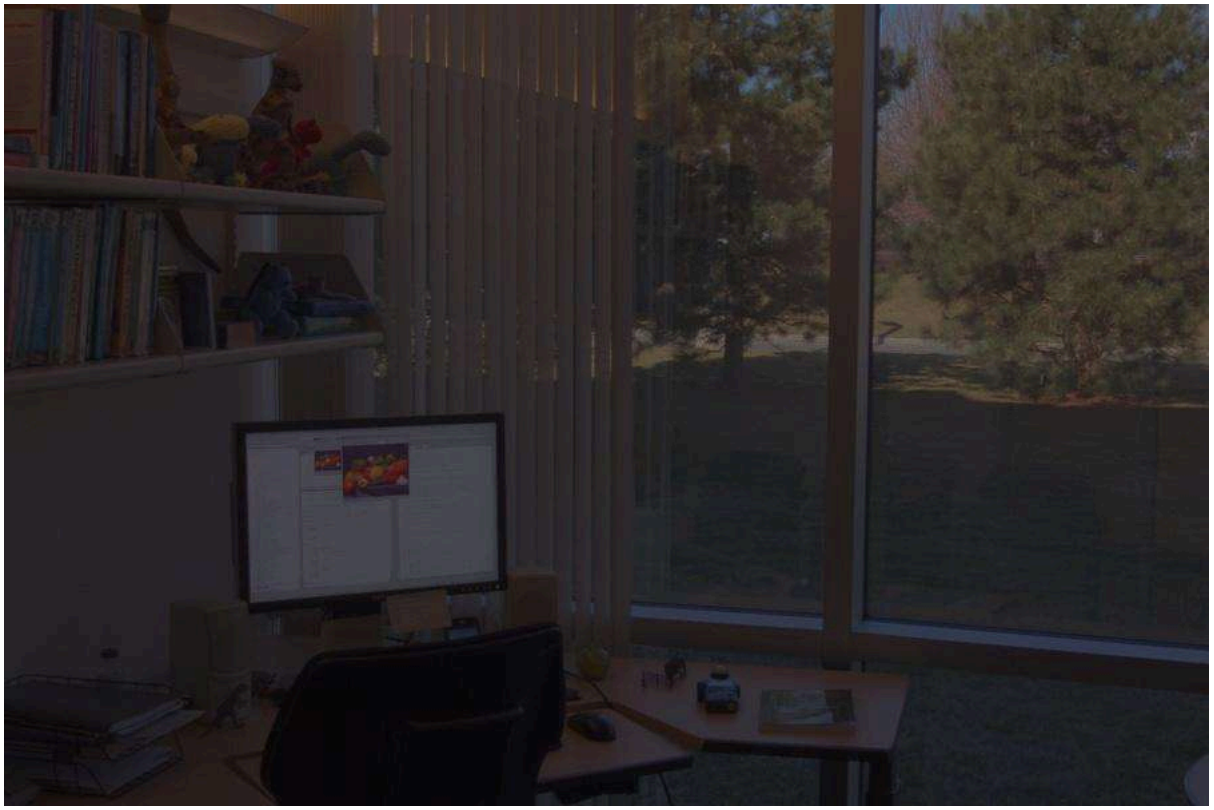
Em que a função center_surround é:

```
206 def center_surround(img, k_size, sharp=8.0, key=0.18):
207     gaussian_1 = cv.GaussianBlur(img, (k_size, k_size), 0)
208     gaussian_2 = cv.GaussianBlur(img, (k_size + 2, k_size + 2), 0)
209     normalizer = ((2**sharp) * key / (k_size**2) + gaussian_1)
210
211     # W[x,y,s(i)] = (V[x,y,s(i)] - V[x,y,s(i+1)]) / ((2^phi)*alpha*(s(i)^2) + V[x,y,s(i)])
212     center = (gaussian_1 - gaussian_2) / normalizer
213
214     return center
```

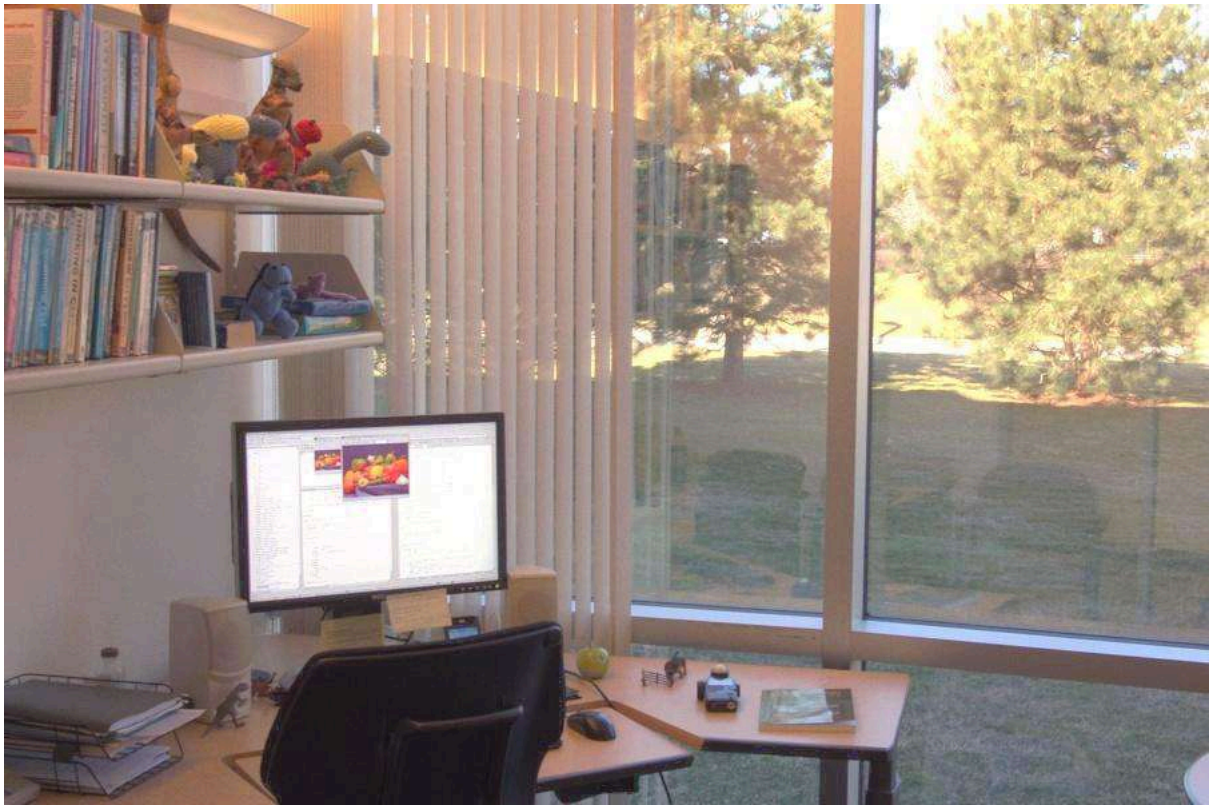
Resultado com gamma=1.0:



Resultado com gamma=2.2:

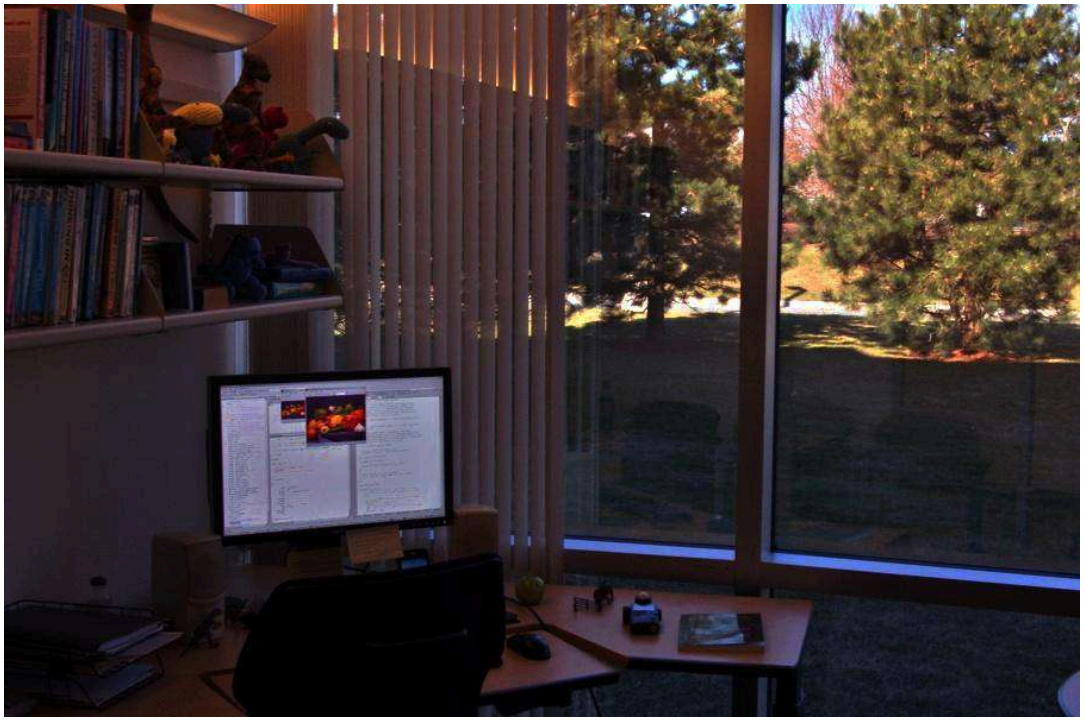


Resultado com gamma=2.2 + white balance:



Ao comparar as imagens do algoritmo local com o global, percebemos que, principalmente na região do céu, temos uma propagação diferente da luz. Na imagem do algoritmo local, a nitidez é maior nessas áreas luminosas.

Local gamma=1.0:



Global gamma=1.0: