ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

SEMESTER PROJECT REPORT

Dataset and algorithm for scene relighting

Johan BARTHAS

Image and Visual Representation Lab

Supervisors :
Majed El Helou, Ruofan Zhou

Professor : Sabine Süsstrunk

Fall Semester 2019

February 15, 2020

Table of contents

Section 1

Introduction

Scene relighting is a valuable problem. Indeed, in professional photography, complex lighting setups are necessary for a photographer to get the best out of a model whereas consumer photographers are constrained to environment illumination. But the use cases are not limited to photography, we can expect a big use in augmented reality if ever we were able to do realistic real-time object relighting to have a perfect integration of objects on the screen. Last but not least, there is a big potential for e-commerce to provide customers a more realistic look for the clothes that they want to buy.

Nonetheless, datasets are always a bottleneck for scene relighting. Real scenes datasets require complex lighting setups using several illuminants and a lot of preparation [18] which can require photography experts, increase the cost of production and also need to spend a lot of time to take the scenes. Besides, it has been shown in [13] that it is possible to reduce the complexity of lighting setups by using the reflections of flashlights as virtual light sources by monitoring the direction of directional flashlights, but still the limitations of this approach are the need to take only indoor scenes. The former limitations could be overcome by relying on synthetic approaches that could produce any realistic scene without being limited by any complex lighting setup.

This paper [22] showed a very interesting synthetic approach to this problem by generating procedural shapes. However, despite showing really good results on relatively small objects, it was not shown to be a viable solution for real-life applications for entire environments.

Nonetheless, game engines are close to reality enough to build datasets with and they are cheap ways to generate data as mentioned in this paper [15]. Moreover, the increase of computing power makes possible to have the real-time realistic rendering and therefore fast access to realistic scenes. In this paper, we describe a game engine approach using Unreal Engine 4 that allows us to fastly and cheaply generate linear images of realistic scenes. Unreal Engine had already been used for Computer Vision research [14] but is not anymore supported in newer Unreal Engine 4 versions.

Our approach allows us to generate 15840 different 32-bit linear images: 396 scenes with 40 illuminants per scene. We spend 9 minutes to generate each scene. The illuminants that we use are synthetic light bulbs from Unreal Engine 4 that we move in each cardinal direction around each center of scenes. We also allow fast creation of new datasets using a gaming-like experience to capture new scenes. Finally, we show in section 4 the possibilities using a proof-of-concept dataset to handle color temperature relighting using CycleGAN [23].

Section 2

Literature

Obtaining outdoor scenes with various lighting conditions can be easily solved since the sky and the sun is varying over time. In fact, timelapse datasets have been created using web cameras [21, 19], video collections [16] or using controlled camera setups [17, 8, 7]. Nonetheless, the use of these approaches does not allow the creation of a multi-illumination image in all the cardinal directions around a given object or scenery since we're depending on the sunlight.

Besides, indoor scenes with different sources of illumination are often harder to obtain and require complex hardware setups and multiple light sources [1, 2]. Also, some approaches use a stationary motor-controlled light source and indirect bounce illumination [13, 12] to generate HDR images with several illuminations. These methods remain restricted to indoor capture and unfortunately, we did not find any unified approach as we do to deal with both landscape-size scenes and tiny objects.

To assess the quality of our dataset, we will perform image-to-image translation using CycleGAN [23]. This model was initially developed to learn how to translate images from a source domain X to a target domain Y without having paired examples. Our dataset presents aligned paired examples but for color temperature, for example, this model can allow us to go unaligned without any problem.

Section 3

Implementation

3.1 Unreal Engine 4

A gaming engine is something that allows game developers to build fastly on top of its different games. It allows both the real-time rendering part and the programming part. The advantage of Unreal Engine 4 is that it is a fully open source for research projects and therefore is very versatile. it is also perfect for realistic computer vision.

This gaming engine gives easy access to cutting-edge realistic gaming technologies such as real-time ray tracing, moveable lights and a huge amount of parameters for lighting setup. Moreover, the camera is fully customizable with exposure, lens, white balance control, etc... And finally, it allows the rendering of HDR 32bit/channel linear images, image normals, and scene depth.

Furthermore, Epic Games which develops Unreal Engine often publish free content with very realistic map levels. And last but not least, even though it came at the end of this project, the release of version 4.24 of Unreal Engine 4 integrates for now on Quixel Megascan objects which are real-life scans of a lot of landscapes and objects. The figure 1 is an example of what can produce Quixel without a lot of design required.

Figure 1: Example of realistic scene that can be produced by Unreal Engine 4 using the Quixel add-on.

Despite being a very versatile gaming engine, some drawbacks must be presented. The first one is that since the gaming engine must produce high FPS during runtime, it requires the compilation of shaders which can be very long with a huge map level. For example, an environment with about 100,000 objects can take up to 3 hours to compile the shaders. Shaders are necessary to reduce the computing required to generate realistic images during runtime by optimizing the pipeline for Graphics Processing Units (GPUs) since shaders are usually developed for parallel computing, the figure 2 presents the usual distribution of work of CPU and GPU during game runtimes.

Another drawback is that we can not give a spectral profile as a parameter to build a light and therefore, we

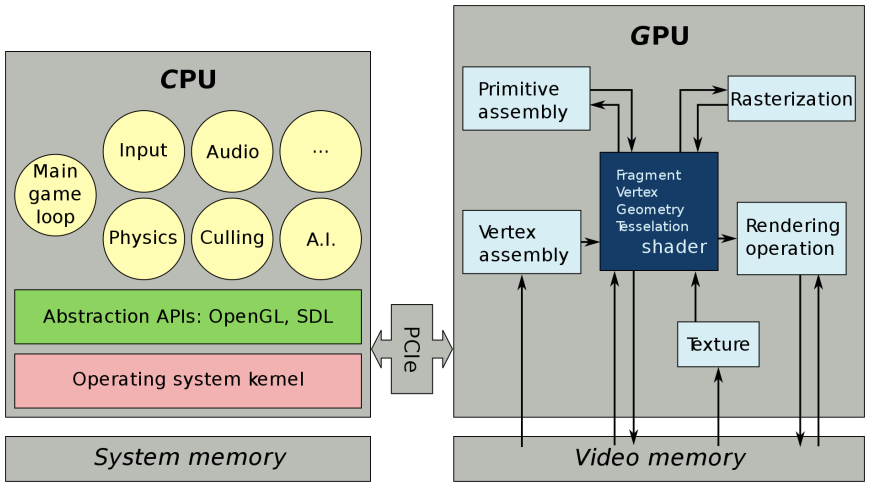

Figure 2: Usual CPU/GPU work distribution in video games.

have to trust the developers for the physics involved in the gaming engine. We still have a wide range of parameters and especially color temperature.

3.1.1 Picking the scenes in the environments

For our task, we collected both free and paid environments to generate the dataset. Unfortunately, not all environments will work with our approach: environments showing a poor collision design will not allow us to have realistic lighting results for two reasons, our method to decide what will be the center of the scene is done by sending a ray trace from the camera actor (us) in the direction pointed out by a crosshair. But if ever the collision is not enabled on the object that we want to focus on, then the ray trace will not stop where we want it to stop and therefore this will be an outlier in our dataset. Therefore, we have to be really careful about the scenes that we pick.

To help us to pick the scenes, we create an actor in Unreal Engines which is composed of 9 light bulbs: 8 in each cardinal directions at an equidistance to the center where we also place a light bulb. To this actor, we draw a cuboid collision component so that there will not be any obstacle between us and any of the light bulb; this is done in order to avoid black images. Then we follow the flow 3 to pick the scenes.

Pressing P will place the actor that we created above the hit point of the ray trace that we shot from where we are. In order to have always the same luminosity, we introduce a scale factor, this factor is computed such as: $\textbf{scale} = \textbf{ray trace distance}/\textbf{ref distance}$. The figure 4 describes the actor that we mentioned precedently and introduce hyper-parameters that we have to choose.

The hyper-parameters for this algorithm are the following:

- **ref distance**: the reference distance that allows us to compute the scale of the scene from it
- **ref radius**: the reference radius between the center and the peripheral light bulbs
- **ref luminosity**: the reference luminosity, to be adjusted visually
- **ref attenuation radius**: the reference attenuation radius of the light bulbs, to be adjusted visually
- **ref offset**: the reference vertical offset between the ray trace hit point and the plane including the light bulbs

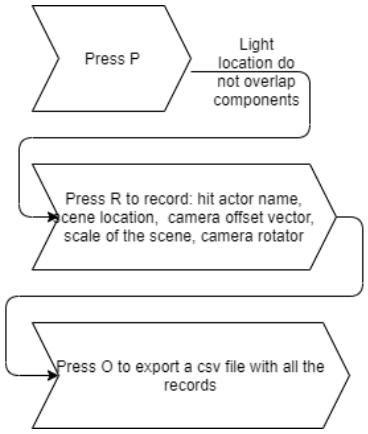

Figure 3: Flow diagram showing the algorithm that we follow to pick the scenes in an Unreal Engine 4 environment using our algorithm.

As described in figure 3, for each scene such that the light actor collision component is not overlapping any other component, we can press R to store the following data: (hit actor name, scene location, camera offset vector, the scale of the scene, camera rotator). This will help us to place ourselves in the same conditions during the recording step that we will describe afterward.

Once we have repeated the process of pushing P and R, we can export a csv file with all the scenes that we decided to choose. This promotes reproducibility because once we have this file, we just have to put the right csv file with the right environment to generate the dataset again.

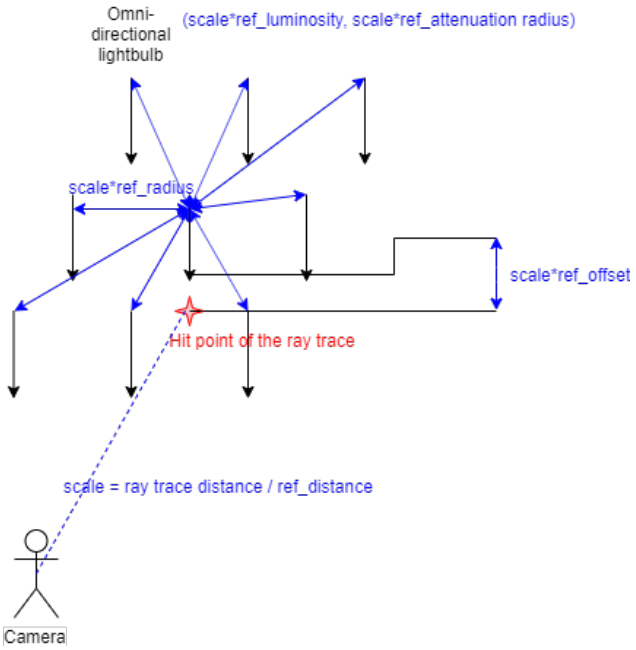

Figure 4: Light bulbs actor that we place above the scene that we want to capture. The goal is to obtain a visual of what it will looks like when we will generate a dataset.

3.1.2 Generating the scenes

Once we have a csv file describing all the scenes that we want to record, we can start to generate the scenes. This looping algorithm is basically going through all the scenes with all the illuminations that we want.

In our algorithm, we allow the possibility to generate images using several light bulbs on cardinal directions. For each scene, we choose the light to appear in a csv where:

- The rows are binary 8-tuples (ex: 0,0,1,0,1,0,1,0)
- The columns are the light location around the scene (NW, N, NE, E, SE, S, SW, W)

This light dataset describes the different lighting setup that we have to go through. And also we have to precise as hyper-parameter what are the different color temperature that we want the light bulbs to take. Unfortunately, the tinting parameter is not available for the light bulb actors and therefore we can only move the color of the lighting through the line described in figure 5.

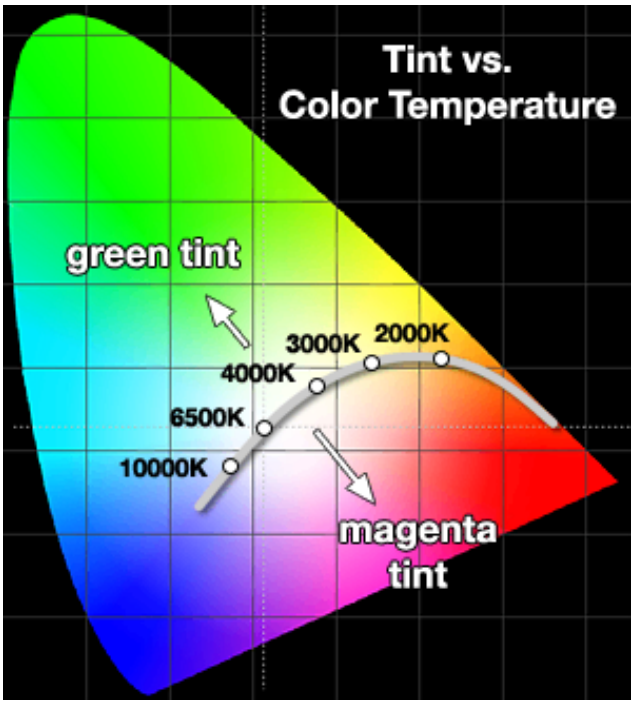

Figure 5: CIE XYZ color space diagram explaining the difference between tint and color temperature.

We also have additional hyper-parameters:

- Upscale parameter: default is 2 which leads to 1024x1024 resolution
- Image initial index: useful to merge datasets
- Time dilation: default is 0.0001, it is used to fix the moving particles, but it has to be set at 1 when you pick the scenes, else you will not be able to move the camera

And more technical hyper-parameters: time dilation, loop delay for synchronization. The need for the two last parameters is due to the fact that we used Unreal Engine as a designer would do: doing very high-level

Object-Oriented Programming on top of the C++ libraries developed by Epic Games and contributors using blueprints (example of such a blueprint: figure 6). This is a key advantage of Unreal Engine since it allows us to do complex and fast computing tasks but the main drawback is that we are forced to do asynchronous programming for a task that we launch in the gaming engine terminal. For example, taking High-Resolution Screenshot have to be launch in a terminal and is required to have access to the best quality of screenshots.

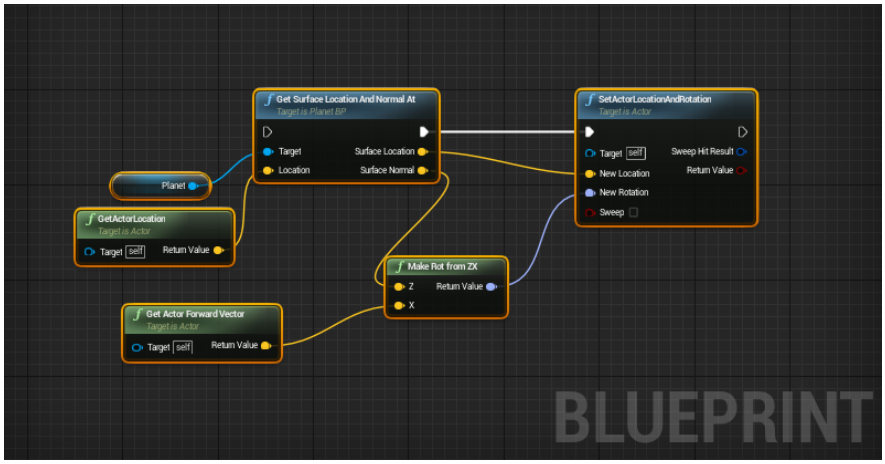

Figure 6: Example of a blueprint in Unreal Engine 4. Blueprints are a way to do very high-level Object-Oriented Programming on top of C++ libraries in Unreal Engine 4.

3.1.3 Including a new environment in our project

Including a new environment is a required step to increase the number of scenes that we have but it is not that simple.

Firstly, we have to open the map level that is given with the environment that we want to include and delete all the unnecessary lighting such as directional lights, light bulbs, sky spheres, etc... We also have to remove unnecessary things such as camera actors, player starters, etc...

Then, we have to copy everything using the top view and paste it into a clean map level that has our code inside of it. To do it, we have to duplicate a previously working scene map level which had our code and remove the environment but not what we do want to keep. We want to keep the light bulb that we defined and are in a folder named *light locations*, a directional light that we use to have light while we pick the scenes and has to be set to hide in-game when we generate the datasets.

Finally, we hopefully managed to copy it without bypassing the RAM and we can now start to pick the scenes and generate a new dataset.

3.1.4 Automatic segmentation

We tried to do automatic segmentation with a unique color for each actor in the scene but it did not work. The procedure was the following:

1. Change the view from lit to unlit in order to have true colors of all components
2. Generate an equivalence dataset between the name of actors and unique color

3. Generate a dynamic material for each unique color
4. Change each actor with its corresponding dynamic material
5. Record a dataset with only 1 illuminant condition since lighting has no importance in unlit mode

Nevertheless, it did not work because all the components in our scene are not from the class Actor and therefore we had objects that ended up not having any texture.

3.2 Color temperature relighting

In this subsection, we first introduce the approach that we will follow to change the white balance using linear operations then we will present the network architecture that we pick for CycleGAN and the training details.

3.2.1 Changing color temperature with linear operations

Before proceeding to deep learning, we assess the performance that we can get by changing white balance with linear operations. Color balancing is performed on three-component images using a 3x3 matrix when the image was captured using the wrong white balance setting on a digital camera. And this is the case for us since the synthetic camera that we use is always set up to 6500K and therefore, for most cases, it is not appropriate.

To scale monitor RGB, the principle is the following. We want to scale all the colors relatively to a color that is supposed to be neutral. For example, if an area of an image with $(R = 200, G = 150, B = 177)$ is supposed to be a white object and if a white object is described with $(R = 255, G = 255, B = 255)$, we must multiply all red values by $255/200$, green values by $255/150$ and blues values by $255/177$. In theory, we should obtain a color-balanced image. The computation can be resumed by the matrix multiplication in equation 1.

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 255/R'_w & 0 & 0 \\ 0 & 255/G'_w & 0 \\ 0 & 0 & 255/B'_w \end{pmatrix} \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} \tag{1}$$

Where (R, G, B) is the color balanced tuple of red, green and blue components of a pixel in the image, (R', G', B') the tuple of the image before color balancing and (R'_w, G'_w, B'_w) the tuple of a pixel believed to be a white surface in the image.

In our case, we know the exact temperature of the lighting and therefore we can use black body color temperature scaling to adapt the color of our images in the sense that we know the tuple (R'_w, G'_w, B'_w) for all our images. We use the Python package **color-temp**[1] to compute the white references for given color temperatures. Thus the color balanced tuples are obtained with equation 2.

$$\begin{pmatrix} R2 \\ G2 \\ B2 \end{pmatrix} = \begin{pmatrix} R2_w/R1_w & 0 & 0 \\ 0 & G2_w/G1_w & 0 \\ 0 & 0 & B2_w/B1_w \end{pmatrix} \begin{pmatrix} R1 \\ G1 \\ B1 \end{pmatrix} \tag{2}$$

Where $(R2, G2, B2)$ is the color balanced tuple of a pixel in the image in the new temperature, $(R1, G1, B1)$ the tuple of a pixel in the image in the original temperature, $(R2_w, G2_w, B2_w)$ the white color in the new temperature, and $(R1_w, G2_w, B2_w)$ the white color in original temperature.

[1] https://github.com/narfman0/color-temp

3.2.2 CycleGAN

We assess now color temperature relighting using our dataset. We used CycleGAN to assess the performance as it is the state-of-the-art in image-to-image translation.

3.2.2.1 Formulation

CycleGAN was an algorithm introduced in [23] as a new approach to image-to-image translation. The goal of image-to-image translation is usually a task where you want an algorithm to map image input of domain X to image output of domain Y using pairs of images. Nonetheless, learning with constrained datasets or where the pairs just do not exist makes the task difficult. CycleGAN deal with this issue by learning in cycle in the sense that the algorithm must be able to map $G : X \longrightarrow Y$ input of domain X to output of domain Y while also being able to map $F : Y \longrightarrow X$ back the output of domain Y to a fake image of domain X so that $F(G(X)) \approx X$. The idea of the model is presented in figure 7.

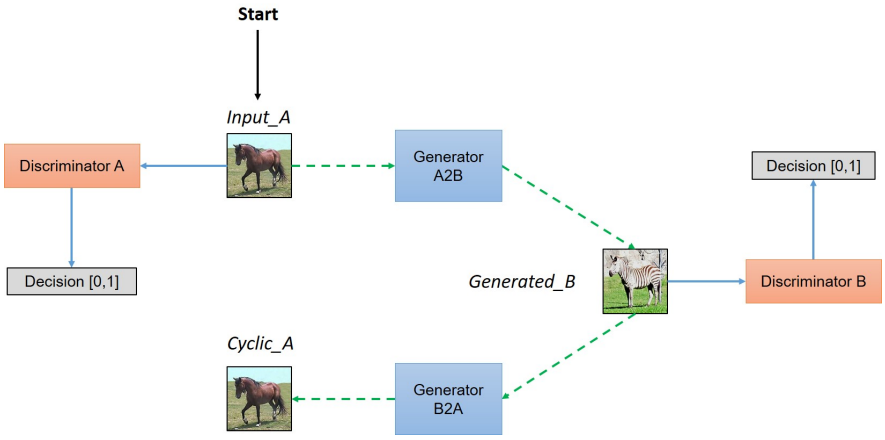

Figure 7: A graph to describe the CycleGAN model.

This model uses two adversarial discriminators D_X and D_Y such that D_X aims to distinguish between images from X and images from $F(Y)$, and D_Y from Y and $G(X)$. To that is added two adversarial losses for matching images generated to the distribution to the target domain on both sides. And also, a cycle consistency loss is added to keep consistency between G and F. Those losses are expressed such that:

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{data}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{data}(x)}[\log(1 - D_Y(G(x))] \tag{3}$$

$$\mathcal{L}_{GAN}(F, D_X, Y, X) = \mathbb{E}_{x \sim p_{data}(x)}[\log D_X(x)] + \mathbb{E}_{y \sim p_{data}(y)}[\log(1 - D_X(F(y))] \tag{4}$$

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{data}(y)}[\|G(F(y)) - y\|_1] \tag{5}$$

Therefore the total loss is:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(F, D_X, Y, X) + \mathcal{L}_{GAN}(G, D_Y, X, Y) + \lambda \mathcal{L}_{cyc}(G, F) \tag{6}$$

Where λ is a regularization factor used to control the relative importance between the GAN objective and the cycle consistency objective.

3.2.2.2 Network architecture

The architecture for our generative networks is based on [5]. This network contains two stride-2 convolutions for both down-sampling and up-sampling parts with instance normalization [20] and 9 residual blocks. The number of residual blocks for 256x256 images was explored in [23].

For the discriminator networks, we use 70x70 PatchGANs [4, 10, 9]. This approach aims to classify whether 70x70 overlapping image patches are real or fake.

3.2.2.3 Training details

For all the training, we used subsets of the dataset that we generated with our script in Unreal Engine 4.

We stabilize the GAN losses using least-squares GAN losses instead of negative log likelihood objectives as described in [11]. Practically, for a GAN loss $\mathcal{L}_{GAN}(G, D, X, Y)$, we train G to minimize $\mathbb{E}_{x \sim p_{data}(x)}[(D(G(x)) - 1)^2]$ and D to minimize $\mathbb{E}_{y \sim p_{data}(y)}[(D(y) - 1)^2] + \mathbb{E}_{x \sim p_{data}(x)}[(D(G(x)))^2]$ since we consider label 1 to be a real image and 0 to be a fake image.

For all experiments, we set $\lambda = 10$ in equation 5 and we add $1/2$ in front of the identity loss ($\mathbb{E}_{y \sim p_{data}(y)}[\|G(F(y)) - y\|_1]$ in equation 5) because we prioritize cycle loss before identity loss. We use Adam solver [6] with regularization factors (0.5, 0.999) and with a learning rate of 0.0002 with a linearly decaying of the learning rate starting after epoch 100. We use a batch size of 8 using two GTX Titan X.

Section 4

Results

4.1 Current dataset

The current dataset was generated using the environments listed in section 6.1.5 and currently have the following content:

- 8 single cardinal direction lights per scene (NW,N,NE,E,SE,S,SW,W)
- 5 color temperatures per scene (2500K, 3500K, 4500K, 5500K, 6500K)
- 396 scenes with indoor and outdoor scenes with different scales split into train/test/validate with 303/46/47 repartition (there is no environment overlapping for each category)
- Total of 15840 1024x1024 linear images (see 6.1.4), normal images, depth images and png sRGB[2] images.
- Total time taken to generate the dataset without taking in account the human time spent to prepare the parameters: 2 days and 18 hours : approx. 10 minutes per scene (40 lighting conditions/scene)
- Dataset size: 115 Go
- Dataset url: https://drive.google.com/open?id=1FdQCDQNjmB4YNV8yZZZgsnidOumfCNfI

The time taken to generate the dataset we came up with was way longer due to technical problems such as hardware problem (not enough RAM, storage full, ...), or connection problem since we were using a Shadow cloud computer that required to have the desktop application open unless it would hibernate.

4.2 Illuminant relighting

To evaluate our models we combine both quantitative and qualitative analysis. For our quantitative analysis, we use the metrics Peak Signal to Noise Ration (PSNR) and Structural SIMilarity (SSIM), a comparison between the two metrics was done in this paper [3].

4.2.1 Linear approach

4.2.1.1 Qualitative analysis

We see in figure 8 that there is an obvious difference between real images and linear transformed ones. We can see that there is a green tint on all these images compared to the real ones. In figure 9, the linear approach has the benefit to save the intensity of the colors but there is a bias compared to the real images.

[2] https://www.w3.org/Graphics/Color/sRGB.html

Figure 8: 1st row: 2500K real images, 2nd row: images generated using a CycleGAN model to transfer from 2500K real images to 4500K, 3rd row: 4500K real images, 4th row: images generated using a linear model to transfer from 2500K real images to 4500K, 5th row: images generated using a CycleGAN model to transfer from 2500K real images to 6500K, 6th row: 6500K real images, 7th row: images generated using a linear model to transfer from 2500K real images to 6500K.

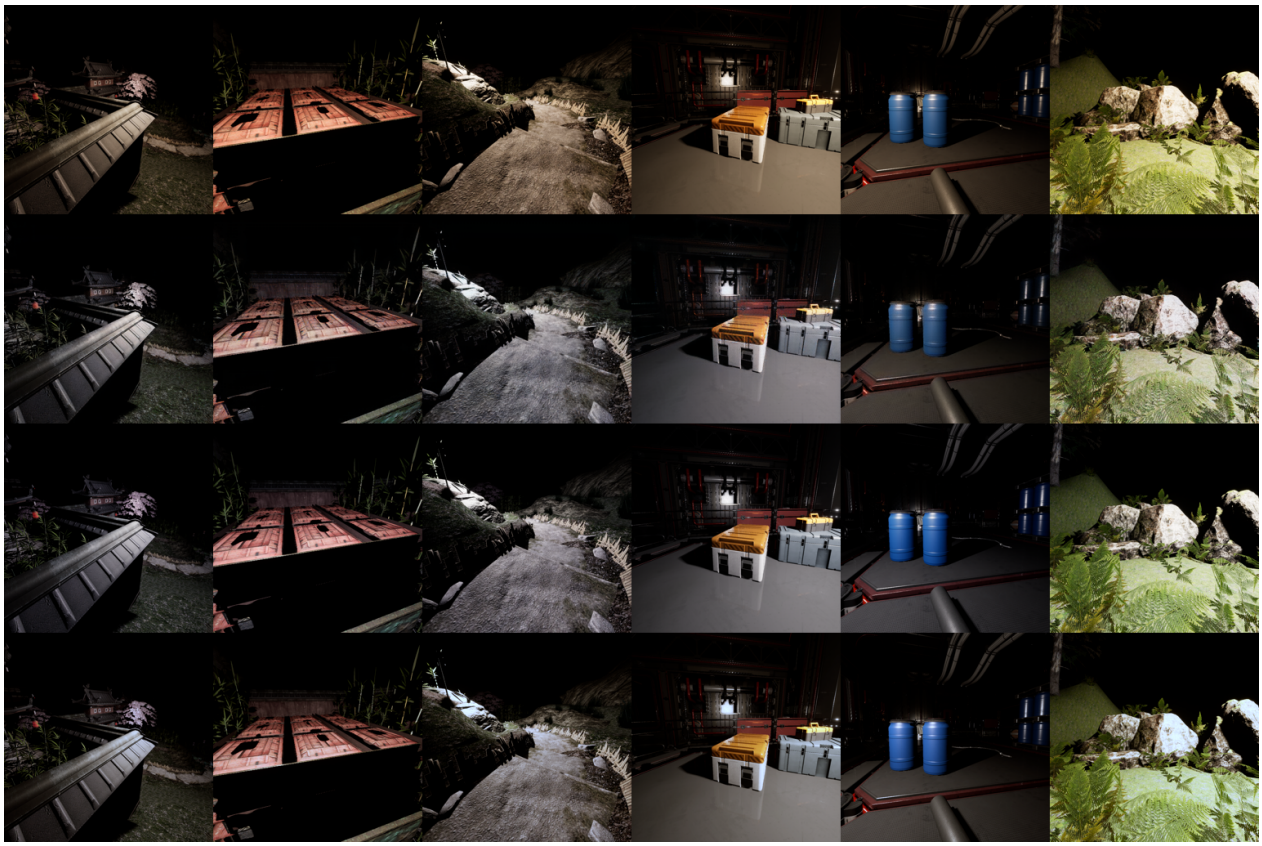

Figure 9: 1st row: 4500K real images, 2nd row: images generated using a CycleGAN model to transfer from 4500K real images to 6500K, 3rd row: 6500K real images, 4th row: images generated using a linear model to transfer from 4500K real images to 6500K.

4.2.1.2 Quantitative analysis

We run experiments with linear transformation using the following test samples:

- Color temperature relighting 2500K to 4500K
- Color temperature relighting 2500K to 6500K
- Color temperature relighting 4500K to 6500K

We obtain the results table 4.2.1.2. We see that overall, we obtain a good PSNR in comparison to the SSIM results. This can be explained by the fact that PSNR is more sensitive to noise as mentioned in [3] but SSIM explains better the similarity for an human eye. We see that the best linear model is to go from 4500K to 6500K. We see that overall there is a green tint deviation that can explain the good PSNR results although we have average SSIM results.

4.2.2 CycleGAN approach

We trained CycleGAN for different tasks:

Linear relighting	2500_4500	2500_6500	4500_6500
SSIM A2B	0.70	0.59	0.76
SSIM B2A	0.71	0.61	0.76
PSNR A2B	21.8 dB	18.0 dB	27.4 dB
PSNR B2A	23.6 dB	19.4 dB	27.0 dB

Table 1: We assess quantitatively the performance of our linear models using SSIM and PSNR metrics. The A2B contraction means the transfer between one domain A to a domain B, for example from 2500K illuminant to 4500K illuminant. A close to 1 SSIM value means that the transfer was a success while 0 means that there is no similarity between two images. The higher the PSNR is, the better it is and PSNR is more sensitive to noise.

- Color temperature relighting 2500K to 4500K
- Color temperature relighting 2500K to 4500K with unaligned pairs
- Color temperature relighting 2500K to 6500K
- Color temperature relighting 6500K to 2500K
- Color temperature relighting 4500K to 6500K
- Cardinal direction relighting SW to SE at 4500K

4.2.2.1 Qualitative analysis

We can see on figure 8 that it is very hard to distinguish the fake images from the real ones but if we give attention we see that there are some inconsistencies on large areas with a uniform color, but overall it does the work very well and way better than the linear approach since the colors are closer to the real images. Nonetheless, we see that we lose the intensity of the colors when transferring from 2500K to 6500K: the pink color of *sakura* is less intense than on the real image at 6500K. On the figure 10 we see that the transfer from South West lighting to South East lighting did not work at all and the map functions $F : A \longrightarrow B$ and $G : B \longrightarrow A$, where A is the domain with SW illuminant and B with SE illuminant, are just identity functions. We, therefore, did not manage to transfer the shadow styles from one domain to another. In figure 9, we see that we lose intensity in the colors but still, we are very close to the real images.

4.2.2.2 Quantitative analysis

To run the experiment, we go evaluate PSNR and SSIM on the whole test dataset using our pretrained weights that are available here. We obtain the results in table 4.2.2.2.

We see that we get very good results for the following relighting: 2500K to 4500K, 2500K to 4500K unaligned and 4500K to 6500K. Indeed, the SSIM metric is close to 1 when there is high similarity between the images and we get an average score above 0.75 for these 3 models. The results for color relighting from 6500K to 2500K and 2500K to 6500K are the same since CycleGAN is a symmetric problem and fortunately, it seems to converge quite well for color temperature relighting. From these 5 models, we can say that CycleGAN works best when the shift of color temperature is small like 2000K but when we do a jump of 4000K, we get worse results due to color imprecisions as we noticed in the qualitative analysis.

Figure 10: 1st row: South West illuminant real images, 2nd row: South West illuminant fake images, 3rd row: South East illuminant real images, 4th row: South East illuminant fake images.

We noted in the qualitative analysis that CycleGAN did not manage to capture the cardinal direction change and did not convert image from South West illuminant to South East illuminant. And the metrics correctly identify it by giving poor results.

Finally, even though we did train only one model both with aligned pairs and unaligned pairs of data, we can see that the results are quantitatively the same. This is recomforting since CycleGAN was designed to work with unpaired datasets.

CycleGAN	2500_4500	2500_4500_unaligned	2500_6500	6500_2500	4500_6500	SW_SE
SSIM A2B	0.74	0.74	0.65	0.69	0.76	0.49
SSIM B2A	0.77	0.77	0.69	0.65	0.77	0.49
PSNR A2B	18.0 dB	18.0 dB	13.9 dB	13.9 dB	18.8 dB	9.07 dB
PSNR B2A	18.0 dB	17.8 dB	13.9 dB	13.9 dB	18.9 dB	9.11 dB

Table 2: We assess quantitatively the performance of our CycleGAN models using SSIM and PSNR metrics. The A2B contraction means the transfer between one domain A to a domain B, for example from 2500K illuminant to 4500K illuminant.

Section 5

Conclusion

We propose a framework based on Unreal Engine 4 to generate realistic dataset for illuminant related works, we generated a dataset using several paid/free environments with 396 scenes and 40 illuminants per scene (5 color temperature, 8 lighting position around the scene) with a time to generate the scenes of 10 min/scene.

Then we assessed our dataset by trying two simple tasks: color temperature relighting and cardinal direction relighting. We obtained good results with CycleGAN for color temperature relighting but terrible results for cardinal direction relighting. We saw that CycleGAN can create some distortion for large areas of uniform colors when the gap between the original color temperature and the final color temperature is too large (4000K) and involves a loss of color intensity but overall the results are very good. Besides, we conducted the color temperature experiment with a linear approach and saw that the intensity of the colors is preserved but there is always a green shift that could allow a human to recognize that it is not a real image or at least not a correctly color balanced image.

It will be important that future research investigate all the possibilities of our dataset and especially working on transferring from one cardinal direction to another.

Section 6

Appendix

6.1 Unreal Engine 4

6.1.1 Setting up Unreal Engine 4 for our project

To be able to run our project, please follow these instructions:

1. Install Unreal Engine 4 in version 4.23.1 (Our project is not working with UE4 4.24 for some reasons)
2. Install Visual Studio (Community) 2019 with Unreal developing support
3. Download our project here RelightingGenerator.zip and unzip it
4. Open *RelightingGenerator.uproject*, you may have to edit this file with Notepad, for example, to remove the plugin LowEntry if you do not have it, and also if you have an error with VictoryBPLibrary, please download the 4.23 version here and replace the content in the folder Plugins/VictoryPlugin23.
5. Once the project is up, wait until the shaders are compiled, it can take several hours
6. Now you have to check that the viewport that is displayed when you click on play is a 1024x1024 window, to do this, you have to go to Edit $\longrightarrow$ Editor preferences $\longrightarrow$ Level Editor $\longrightarrow$ Play $\longrightarrow$ Game Viewport Settings and check that the values of Width and Height are 1024

Unreal Engine 4 is now correctly set up!

6.1.2 Preparing the generation of a new dataset

We will now see how to prepare for the generation of a new dataset. Please follow these instructions:

1. The first step is to make visible a directional light that is already placed in the Scene1 map level which should be already loaded (if not on your content Browser select it, figure 11), and type light in the upper-right search bar. Then double-click on DirectionalLight (figure 12 and go to Details $\longrightarrow$ Rendering and uncheck the box Actor Hidden In Game.

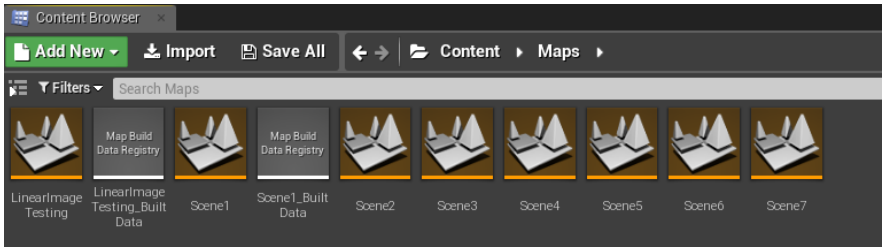

Figure 11: Map levels: different environments that are already set up for dataset generation in our Unreal Engine 4 project.

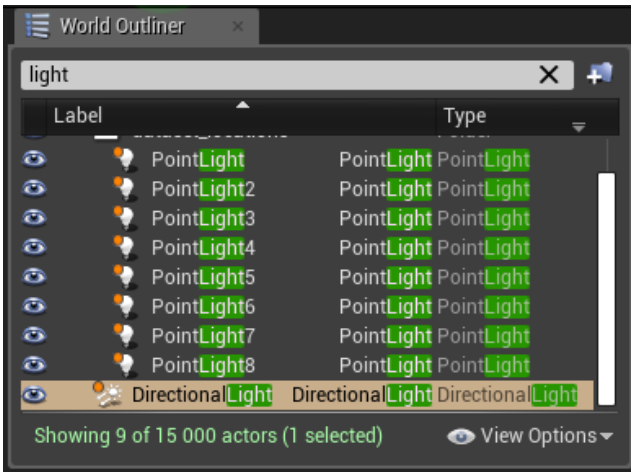

Figure 12: DirectionalLight: Actor that must be hidden during the data generation and visible during the scene selection. The purpose of this illuminant is to avoid moving into the dark while picking scenes.

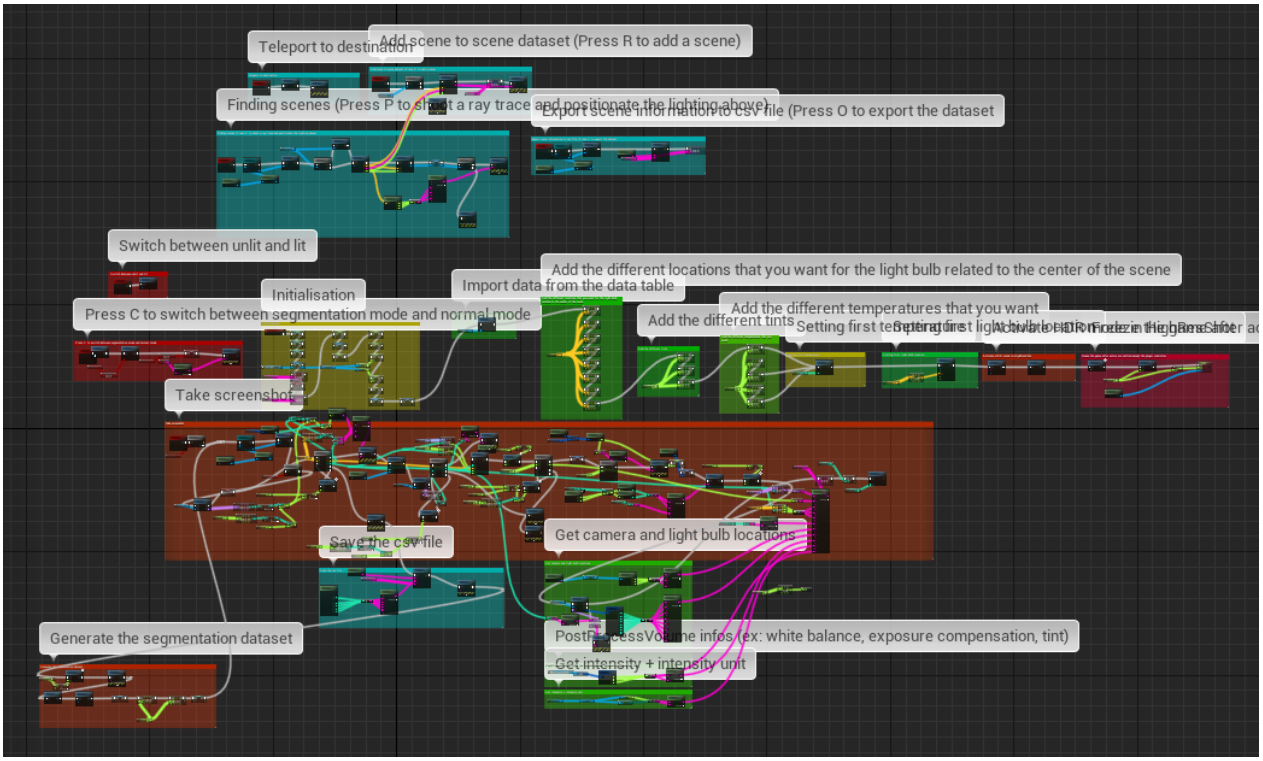

Figure 13: An overview of the main of our algorithm that we developed in Unreal Engine 4.

2. Then go to Blueprints $\longrightarrow$ Open Level Blueprint and click on Graphs $\longrightarrow$ EventGraph (this is the main for the map level blueprint), it should look like figure 13.

3. Then zoom on the yellow box (figure 14) entitled Initialisation and set the parameter Time Dilation to 1, else you will not be able to move in the play mode.

4. Click on Play

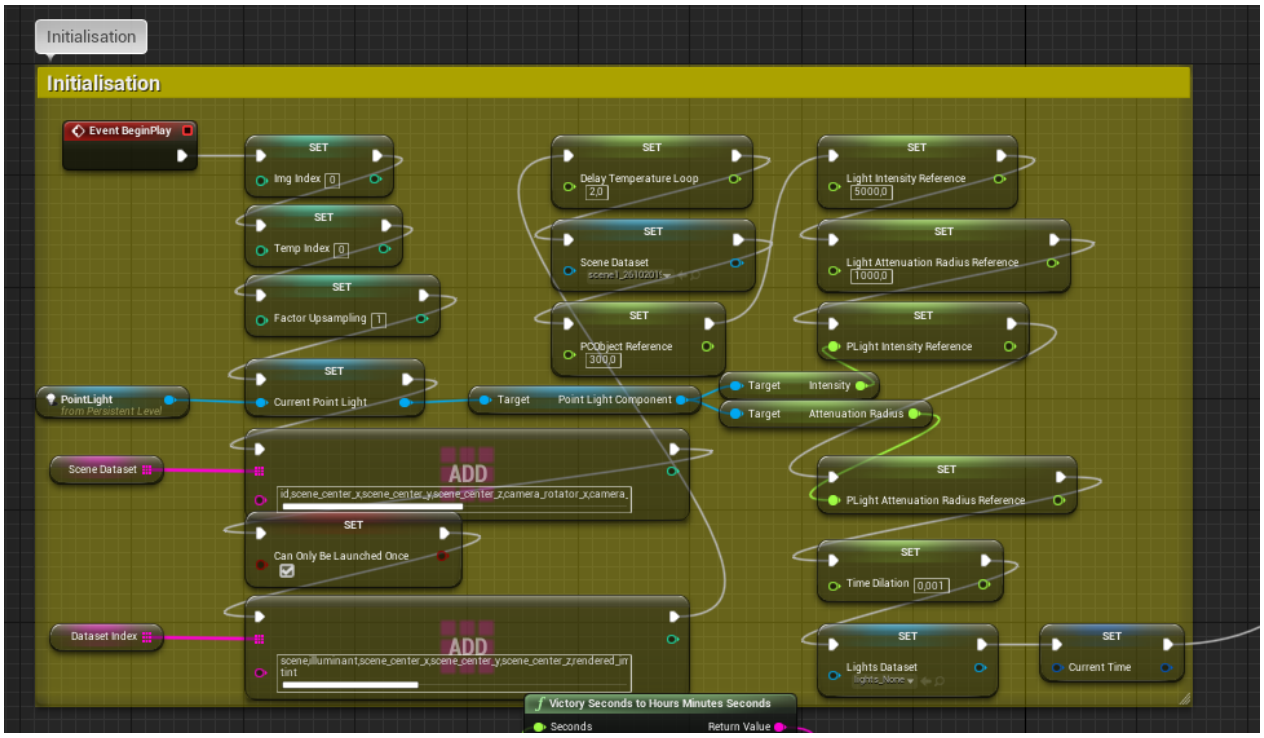

Figure 14: Initialisation box where we can find all the hyper-parameters necessary to have a good flexibility of our algorithm in Unreal Engine 4.

5. Use T to teleport to the pointed destination, use P to place the lights above your target, use R to record the scene if the lights did not overlap other components
6. Use O to export a scene.csv that you can find in the folder RelightingGenerator/Saved/Screenshots/Windows

6.1.3 Generating a dataset

Once the selection of the scenes done, we can now generate a dataset. To do so, please follow these instructions:

1. Hide the directional light following the last subsubsection fist item instruction
2. Go back to the Blueprints $\longrightarrow$ Open Level Blueprint $\longrightarrow$ EventGraph, zoom on initialisation (figure 14)
3. Set Time Dilation to 0.001 (it fixes the moving particles with an ultra-slow motion)
4. Reduce the window and go to Content Browser $\longrightarrow$ Content $\longrightarrow$ Datasets $\longrightarrow$ Scenes, and import the scene.csv that you generated by right-clicking and importing asset. Choose the DataTable Row Type: scene_dataset.
5. Go back to the minimized window and set the Scene Dataset variable to the asset that you just imported.

6. Now set the Lights Dataset variable to light_identity, it will basically move one light in all cardinal directions on a circle around the offset scene center.

For more details on the implementation, please open directly the project, we commented on all the blueprints and used appropriate names for variables. In fact, showing all the blueprints here would take a lot of pages and you'll be able to see all the function blueprints of figure 15.

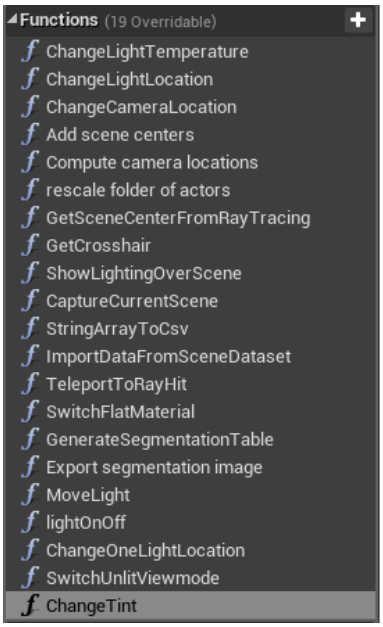

Figure 15: Functions that we use in our algorithm in Unreal Engine 4.

6.1.4 Linearity check of HDR images

An important part of our work is to generate HDR linear images (that we did not use for the experiments) and due to the lack of documentation from Epic Games regarding this linearity, we decided to conduct an experiment to assess this linearity. We took a screenshot of a sphere with a range of material colors from 0 to 1 with the shade of greys. Then we used photoshop to get the output value at the left top corner minus one pixel as in figure 16 and then we plotted the figures 17.

6.1.5 Environments used to generate our dataset

The environments from the marketplace that we used to generate our dataset are the following:

- Table chair: created using starter content
- House in middle of walls: created using starter content
- Little rocks in middle of forest: created using starter content
- Modular Scifi Season 2 Starter Bundle
- City Subway Train - Modular

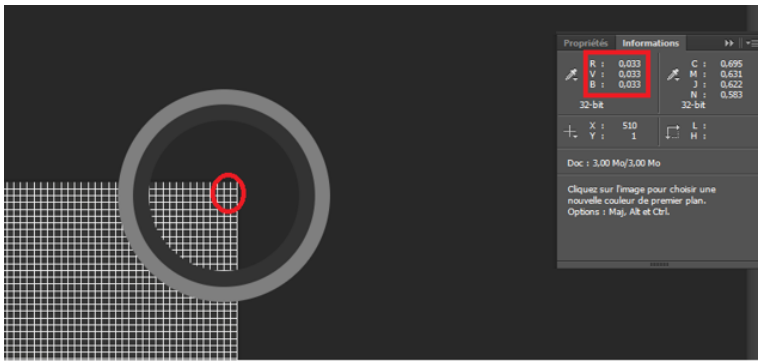

Figure 16: Measuring the left top corner minus one pixel with Photoshop to compare with the color of the object that we selected in the gaming engine. We want to assess the relation between the colors in the gaming engine and the colors of high resolution screenshots. We work with shade of greys to look for an eventual gamma filter.

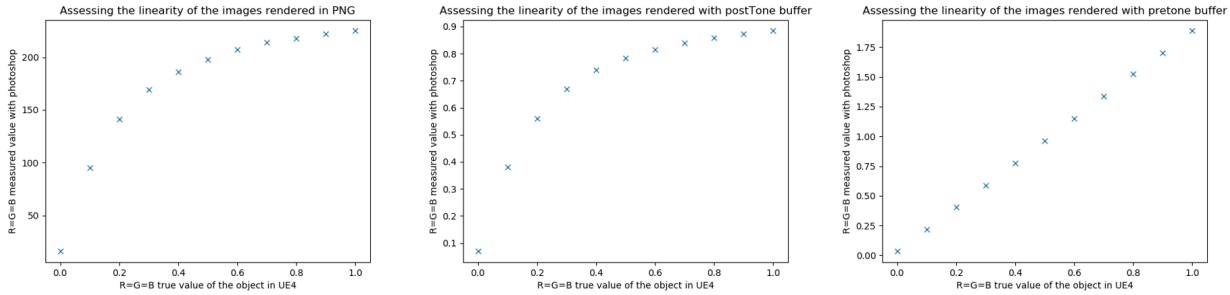

Figure 17: Assessing the linearity of PNG images, HDR images after postTone buffer and HDR images before postTone buffer. When taking a high resolution screenshot in Unreal Engine 4, several buffers are considered before getting the final PNG image. A gamma 2.2 is applied by a toner working with 8-bit per channel images before converting to 8-bit per channel PNG images.

- Advanced Village Pack
- Infinity Blade: Ice Lands
- Modular Building Set
- Infinity Blade: Grass Lands
- Fantasy and Medieval Architecture Kit
- Abandoned Full City Pack with Interiors
- Infinity Blade: Village
- Infinity Blade: Hideout
- Brushify - Arctic Pack
- Brushify - Natural Roads Pack
- [SCANS] Abandoned Factory Buildings - Day/Night Scene

6.2 Python implementation

Our implementation is available here we modify an existing CycleGAN repository to fit to our usage and we add several notebooks and python script to prepare our dataset for CycleGAN. To run our code you will need the following packages: PIL, pandas, numpy, shutil, torch, torchvision, pathlib, visdom, notebook, SSIM-PIL.

6.2.1 Dataset preparation

6.2.1.1 completeness_test.ipynb

Using the dataset that we generated, the first step is to check the completeness of the dataset, to do that, we just have to give the path of a given scene folder and it will check whether all the images are available.

6.2.1.2 dataset_tree_generation.ipynb or prepare_dataset.py

For each scene of our dataset, all the illuminants are in the same folder, therefore we create a hierarchy for each scene folder where we have the cardinal directions folders inside of the color temperature folders inside of the scene folders. We just have to give the path of a given scene folder to do it.

An example for prepare_dataset.py: python prepare_dataset.py –folder './data/train/scene_abandonned_city_54/'.

6.2.1.3 prepare_data_cyclegan.ipynb

This notebook must be used after running prepare_dataset.py on all the scenes, is used to prepare the dataset for the CycleGAN code and can be used as follows:

- Prepare the dataset for color_temperature transfer using the function *generate_dataset_color_temperature*
- Prepare the dataset for cardinal direction tranfer using the function *generate_dataset_both*

6.2.2 Training CycleGAN

Once the dataset is prepared, run the following instructions to train using CycleGAN model:

1. Launch visdom by typing *visdom* in one terminal and open see the progress of your training at http://localhost:8097/
2. Decide the number of GPU that you will use, for example, export CUDA_VISIBLE_DEVICES=0,1 to use only 2 gpus
3. Train the model giving the batchsize, the number of threads to use, the dataroot and whether you want to use Cuda (more options are available, type python train.py -h for detailed information). For example: python train.py –dataroot /scratch/barthas/datasets/2500_4500 –cuda –n_cpu 48 –batchSize 8.

6.2.3 Testing CycleGAN

Once you trained a model on a given dataset you will want to test on the test samples. To do it you have to type, for example: python test.py –dataroot /scratch/barthas/datasets/2500_4500 –n_cpu 48 –cuda. It will export fakeA and fakeB images.

6.2.4 Changing white balance with linear operations

Using the notebook white_balance.ipynb, you can change the color temperature of given images using linear operations. The python script test_linear.py allows us to get linear output from our sample test dataset.

References

[1] Paul Debevec et al. “Acquiring the Reflectance Field of a Human Face”. In: ACM SIGGRAPH (2000).

[2] Michael Holroyd, Jason Lawrence, and Todd Zickler. “A Coaxial Optical Scanner for Synchronous Acquisition of 3D Geometry and Surface Reflectance”. In: ACM SIGGRAPH (2010).

[3] Alain Horé and Djemel Ziou. “Image quality metrics: PSNR vs. SSIM”. In: Aug. 2010, pp. 2366–2369. DOI: 10.1109/ICPR.2010.579.

[4] Phillip Isola et al. “Image-to-Image Translation with Conditional Adversarial Networks”. In: CoRR abs/1611.07004 (2016). arXiv: 1611.07004. URL: http://arxiv.org/abs/1611.07004.

[5] Justin Johnson, Alexandre Alahi, and Fei-Fei Li. “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”. In: CoRR abs/1603.08155 (2016). arXiv: 1603.08155. URL: http://arxiv.org/abs/1603.08155.

[6] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: (2014). cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. URL: http://arxiv.org/abs/1412.6980.

[7] Pierre-Yves Laffont et al. “Transient attributes for high-level understanding and editing of outdoor scenes”. In: ACM SIGGRAPH (2014).

[8] Jean-François Lalonde and Iain Matthews. “Lighting Estimation in Outdoor Image Collections”. In: International Conference on 3D Vision (2014).

[9] Christian Ledig et al. “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network”. In: CoRR abs/1609.04802 (2016). arXiv: 1609.04802. URL: http://arxiv.org/abs/1609.04802.

[10] Chuan Li and Michael Wand. “Precomputed Real-Time Texture Synthesis with Markovian Generative Adversarial Networks”. In: CoRR abs/1604.04382 (2016). arXiv: 1604.04382. URL: http://arxiv.org/abs/1604.04382.

[11] Xudong Mao et al. “Multi-class Generative Adversarial Networks with the L2 Loss Function”. In: CoRR abs/1611.04076 (2016). arXiv: 1611.04076. URL: http://arxiv.org/abs/1611.04076.

[12] Ankit Mohan et al. “Tabletop computed lighting for practical digital photography”. In: IEEE Transactions on Visualization and Computer Graphics (2007).

[13] Lukas Murmann et al. “A Dataset of Multi-Illumination Images in the Wild”. In: ICCV. Oct. 2019.

[14] Weichao Qiu and Alan L. Yuille. “UnrealCV: Connecting Computer Vision to Unreal Engine”. In: CoRR abs/1609.01326 (2016). arXiv: 1609.01326. URL: http://arxiv.org/abs/1609.01326.

[15] Stephan R. Richter et al. “Playing for Data: Ground Truth from Computer Games”. In: CoRR abs/1608.02192 (2016). arXiv: 1608.02192. URL: http://arxiv.org/abs/1608.02192.

[16] Yichang Shih et al. “Data-driven hallucination of different times of day from a single outdoor photo”. In: ACM SIGGRAPH Asia 2013 (2013).

[17] Jessi Stumpfel et al. “Direct HDR capture of the sun and sky”. In: (July 2004). DOI: 10.1145/1186415.1186473.

[18] Tiancheng Sun et al. “Single Image Portrait Relighting”. In: CoRR abs/1905.00824 (2019). arXiv: 1905.00824. URL: http://arxiv.org/abs/1905.00824.

[19] Kalyan Sunkavalli et al. “What do color changes reveal about an outdoor scene?” In: CVPR (2008).

[20] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. “Instance Normalization: The Missing Ingredient for Fast Stylization”. In: CoRR abs/1607.08022 (2016). arXiv: 1607.08022. URL: http://arxiv.org/abs/1607.08022.

[21] Yair Weiss. “Deriving intrinsic images from image sequences”. In: ICCV (2001).

[22] Zexiang Xu et al. “Deep image-based relighting from optimal sparse samples”. In: ACM TOG 37.4 (2018), p. 126.

[23] Jun-Yan Zhu et al. “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”. In: CoRR abs/1703.10593 (2017). arXiv: 1703.10593. URL: http://arxiv.org/abs/1703.10593.