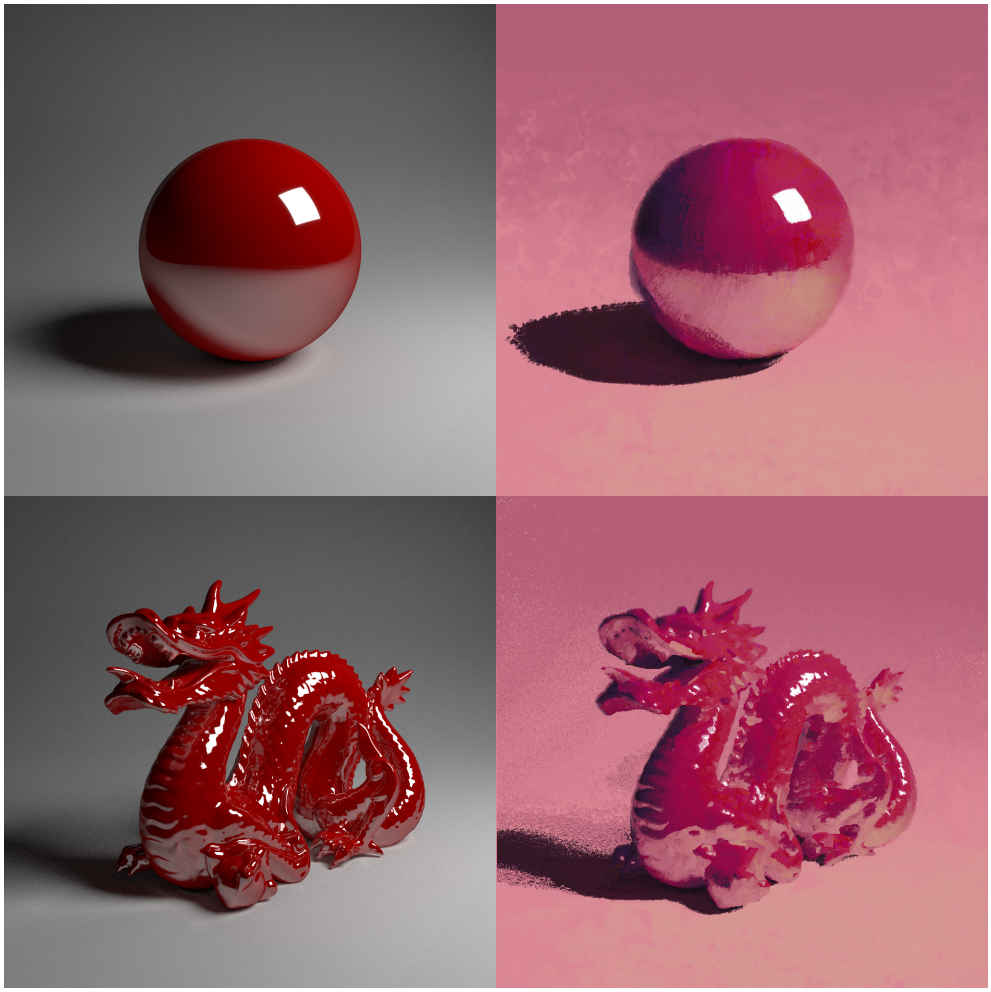
INF584 Final project - StyLit

Octave Crespel

March 31, 2021



Introduction

As an extension to the pathtracer I developed during the INF584 practical sessions, I decided to explore the paper *StyLit : Illumination-Guided Example-Based Stylization of 3D Renderings*. While I did not implement the technique in its entirety, I wrote implementations of key parts in C++.

1 Overview of the *StyLit* technique

This paper is heavily based on a 2001 paper called *Image Analogies*, by Hertzmann et al. As such, I will first go over the material in *Image Analogies*, then describe the main improvements brought forward by the StyLit team.

1.1 Image Analogies

Image analogies are a framework for processing images based on example. It operates on four images A, A', B, and B', which are such that "B' is to B what A' is to A". The authors use the notation $B' : B :: A' : A$ to describe that relationship.

The algorithm proposed by Hertzmann et al. synthesizes B' from A, A' and B. This high-level approach has a number of applications, such as image filtering, image synthesis, or style transposition : for instance,

- if A' is a blurred version of A, then it is expected that the algorithm will produce an image B' that is a blurred version of B.
- if A' is a picture of a landscape, with features described by a discrete set of colors in A, then B' is expected to be another landscape whose features are described by the colors in B.
- if A' is an artist's rendition of a picture A, then B' is expected to represent the picture B with the same stylistic features as those in A'.

The algorithm operates by constructing B' with samples taken from A', which are chosen using two main approaches. Suppose that we would like to synthesize an image patch in B' located around a given pixel q :

1. if an image patch in A, around some pixel p, much resembles the image patch in B that is located around q, then an image patch in A' that surrounds pixel p is a good candidate for our target image patch around q in B'.
2. if the image patch in B' around some $q + \delta$ (a pixel close to q) has already been synthesized from an image patch around p in A', then the image patch in A' that surrounds $p - \delta$ is a suitable candidate for our target patch.

These two strategies optimize for two desirable qualities in the synthesized image : (1) creates patches that locally fit the image analogy framework ($B' : B :: A' : A$), whereas (2) preserves spatial coherence in the final image. Strategy (1) is implemented with a nearest neighbor search over all image patches in A, whereas brute-force is employed for (2), with δ taking all integer values in the range $[-r, r]^2$, with $r = 1$ or 2.

These strategies are applied at increasing resolutions : in effect, the algorithm uses mip-map pyramids of A, A', and B' to produce a mip-map pyramid of B'. At each resolution, the target patches are synthesized in scan-line order. Both strategies are used to compute a candidate patch, and one of them is chosen as the definitive solution based on :

- how close the patches match – both on the filtered image and on the exemplar image,
- the resolution level – indeed, prioritizing strategy 2 at higher resolutions means that high-frequency detail such as brush strokes will be preserved, while broader regions in the image will better maintain the image analogy,
- and a user-provided bias : depending on the application, one might prefer to prioritize either strategy.

While this framework does not dictate how the distance of image patches should be computed, the original authors work with RGB images and use the euclidean norm in luminance space as a measure of perceptual difference.

1.2 StyLit

StyLit is an implementation of the Image Analogies framework that focuses on one specific use case : the stylization of pathtraced 3D renders. Here, A and B are pathtraced renders of two 3D scenes, and A' and B' are stylized versions of those renders. As such, it makes several improvements to the original technique to better fit this application :

- While it still uses RGB images for A' and B', A and B are multi-channel images representing light path expressions passes of the 3D renders. Intuitively, this means that the algorithm can generate the target image in a similar way to what an artist does : different styles or colors can be chosen based on the contributions of direct diffuse illumination (`LDE`), specular highlights (`LSE`), bounce lighting (`L.*DDE`), or even caustics (`L.*SDE`).
- StyLit uses a more elaborate optimization procedure : rather than using either strategy 1 or 2, both are used and the resulting patches are averaged together. Furthermore, to avoid the wash-out effect that this average creates, the assignment of patches from A' onto B' is carefully controlled. Patch reuse is initially disabled. Once a suitable source patch has been found for all patches in B', these assignments are sorted by increasing error. Assignments whose error is higher than an empirical threshold are cancelled, whereas those whose error is below are validated. The procedure is then restarted for the cancelled patches, up until all the patches in the final image have been assigned. This system enforces uniform patch usage as long as it does not highly degrade the image quality : for instance, if the specular highlights in B' the target image occupy a larger area than in A', enforcing uniform patch usage would fill the highlights in B' with non-highlight samples.

2 Implementation efforts

2.1 Improvements to the pathtracer

Any defects in the renders used by the StyLit algorithm will propagate onto the stylized result. As such, before going into the implementation of StyLit, I implemented a few well-known techniques to improve the images produced by my renderer and make later progress easier :

- **Using a linear space for colors** : for the blending and summation of RGB colors to make physical sense, those must lie in a linear space. However, the final image file is interpreted by most image viewers and monitors as lying in the sRGB color space. As such, I performed the conversion before saving the image to disk.
- **Filtering image samples** : when accumulating samples into the final image buffer, the most straight-forward behaviour is to sum onto the pixel closest to where the sample was taken. In effect, this reconstructs the sampled image with a box filter. By instead using a Mitchell-Netravali filter with a radius of 1.5 pixels, I obtained smoother renders with the same amount of samples.
- **Importance sampling the microfacet material** : with cosine-weighted hemisphere sampling, smooth specular materials made the renders converge very slowly. By importance sampling bounce rays according to the G term of the microfacet BRDF, smoother highlights were produced with little runtime cost.
- **Reading scenes from files** : in order to quickly iterate on scene setups and avoid recompilation delay, I implemented a simple importer that reads 3D scenes from a `.toml` configuration file, with the `toml11` library.

- **Synthesizing a distinct pass for each type of light path** : critical to the effectiveness of StyLit is the use of multi-channel images for multiple light path expressions. In order to synthesize those, I slightly altered the structure of the pathtracer : rather than directly computing radiances, a light tree[1] is computed for every camera ray, which records the radiance computed along every branch as well as the surface type of each bounce. The tree is then traversed, and paths matching the light path expressions chosen by the user are accumulated onto the appropriate image buffer. To keep parsing and matching simple, I used a simpler LPE dialect than the StyLit authors :
  - `'L'`, `'D'`, `'S'`, and `'E'` match a bounce at a light, a diffuse surface, a specular surface, and the eye at the root of the light tree, respectively.
  - `'.'` matches any type of surface, once.
  - `'*'` matches any type of surface, zero or more times.

2.2 Implementation of *Image Analogies*

In order to implement the base algorithm, I followed the structure of the *Image Analogies* paper. After building gaussian pyramids of the feature images, kd-trees are built for all levels of the pyramid, which all contain the feature vectors of A at that level. The pyramid for B' is then synthesized, from coarsest to finest resolution. For each patch q to be synthesized, the kd-tree is queried for its nearest neighbor p_{app}. p_{app} is considered, along with the points p_{coh} that were used to synthesize the neighborhood of q at a resolution one level coarser than the one being synthesized. That of p_{app} and p_{coh} which best maintains the perceptual quality of the neighborhood being synthesized is chosen :

$$B'_l(q) = \begin{array}{ll} A'_l(p_{coh}) & \text{if } \|\mathcal{A}_l(p_{coh}) - \mathcal{B}_l(q)\|^2 \leq (1 + 2^{-l}\kappa)\|\mathcal{A}_l(p_{app}) - \mathcal{B}_l(q)\|^2 \\ A'_l(p_{app}) & \text{otherwise} \end{array}$$

Where U_l is the l -th level of the pyramid U (0 being the finest), and $\mathcal{U}_l(p)$ is a concatenation of all feature vectors in a neighborhood around p, at levels l and $l+1$.

I used `libANN` for nearest-neighbor queries, and brute-force for the computation of the optimal p_{coh}.

2.3 Main implementation hurdles

Most improvements to the pathtracer were rather straightforward to implement, or had existing reference materials – such was the case for the GGX probability distribution, for instance. Correctly implementing light path expression filtering was a bit more complicated : while not changing the main algorithm at all, it required substantial changes to the flow of data throughout the program. Rather than just carrying RGB light, ray tracing functions now have to accumulate information about every bounce onto a `LightTree` data structure.

As for image analogies, I made a few unsuccessful attempts before reaching a correct implementation. I initially used simple hand-crafted math data structures, which made building the neighborhood vectors as well as interfacing with `libANN` rather difficult. In the current implementation, I use `Eigen` to store and manipulate image data. This makes vector operations much more comfortable to use. Because `Eigen` does not support 2D arrays of vectors, I implemented a simple wrapper called `EigenArray2D`, which allowed me to index into feature images intuitively.

Furthermore, interfacing with `libANN` proved error-prone as well. The library makes heavy use of C-style arrays in place of STL containers, which is in itself manageable though it requires rigorous bounds checking and data layout considerations. However, `libANN` provides destructors for its objects but no copy constructors or assignment operators. As such,

[1] A tree is computed, rather than a list, because several samples are taken at each bounce : one for each BRDF the surface's material is composed of.

it would often pull the rug from under my program without warning : for instance, the automatic resizing of an `std::vector` of ANN kd-trees would release the allocated memory of the trees without transferring it to the new copies. This resulted in data corruption, which was difficult to debug.

Finally, a particular issue took much longer to fix than I would like to admit. To display and save images, I copied data from `Eigen` arrays to my own `Buffer2D` data structures. Because `Eigen` arrays are stored in column-major order by default, I interpreted color data which was laid out as `RRRRGGGGBBBB` as `RGBRGBRGBRGB`. This resulted in puzzling corrupted images. Because the results displayed some noticeable structure rather than being entirely gibberish, I initially faulted my algorithms rather than my data layout, which proved both time-consuming and unsuccessful.

Unfortunately, I did not have time to implement the error-budgeting and clever patch distribution of *StyLit* : as such, my program is merely an implementation of *Image Analogies* with light path expression passes as feature vectors. I was nonetheless able to synthesize reasonable stylizations.

3 Results

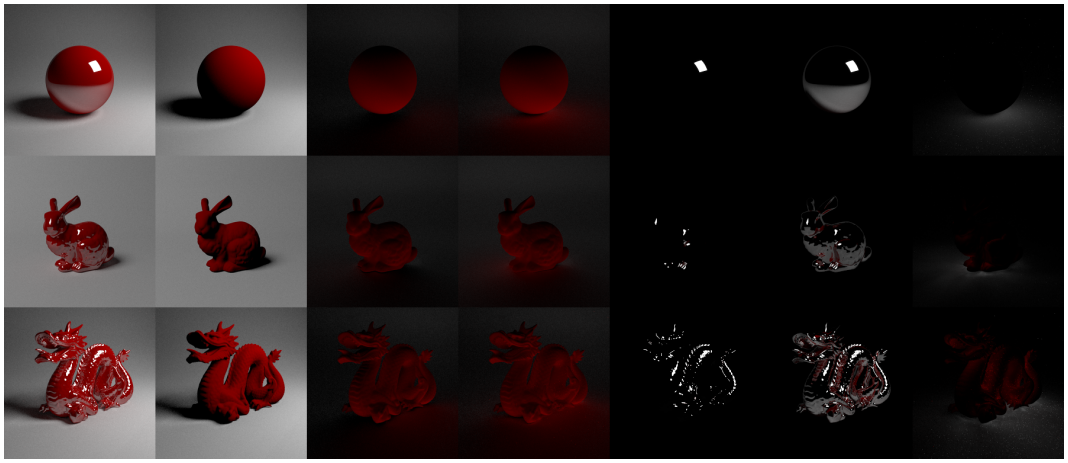


Figure 1: Light passes used for my tests. From left to right : `L*E`, `LDE`, `LDDE`, `L*DDE`, `LSE`, `L*SE` and `L*SDE`

Filtering the render with light path expressions proved to be an interesting tool, both for the results themselves and as a debugging aid. As can be seen in figure 1, the different passes exhibit multiple light behaviours : direct diffuse lighting with `LDE`, specular highlights with `LSE`, bounce lighting with `L*DDE`, and even caustics with "L*SDE". Furthermore, passes with more bounces are more noisy, because the number of path possibilities is higher, which increases variance. On the `L*SDE` pass especially, there are a few extremely bright spots. This is due to the low likelihood of hitting the few particularly bright paths.

The addition of light path computations does not change the algorithms used, and therefore does not impact performance much. I noticed a 5% overhead in total program time when outputting 6 light path expression buffers rather than only accumulating onto a single RGB buffer.

The results of the image analogies algorithm are presented in figures 2. The parameter κ influences how much spatial coherence is favoured against nearest neighbor queries. Increasing it allows for better preserved artistic features, but going past a certain threshold removes the overall structure of the image B in favour of copying larger patches from A' onto B'.

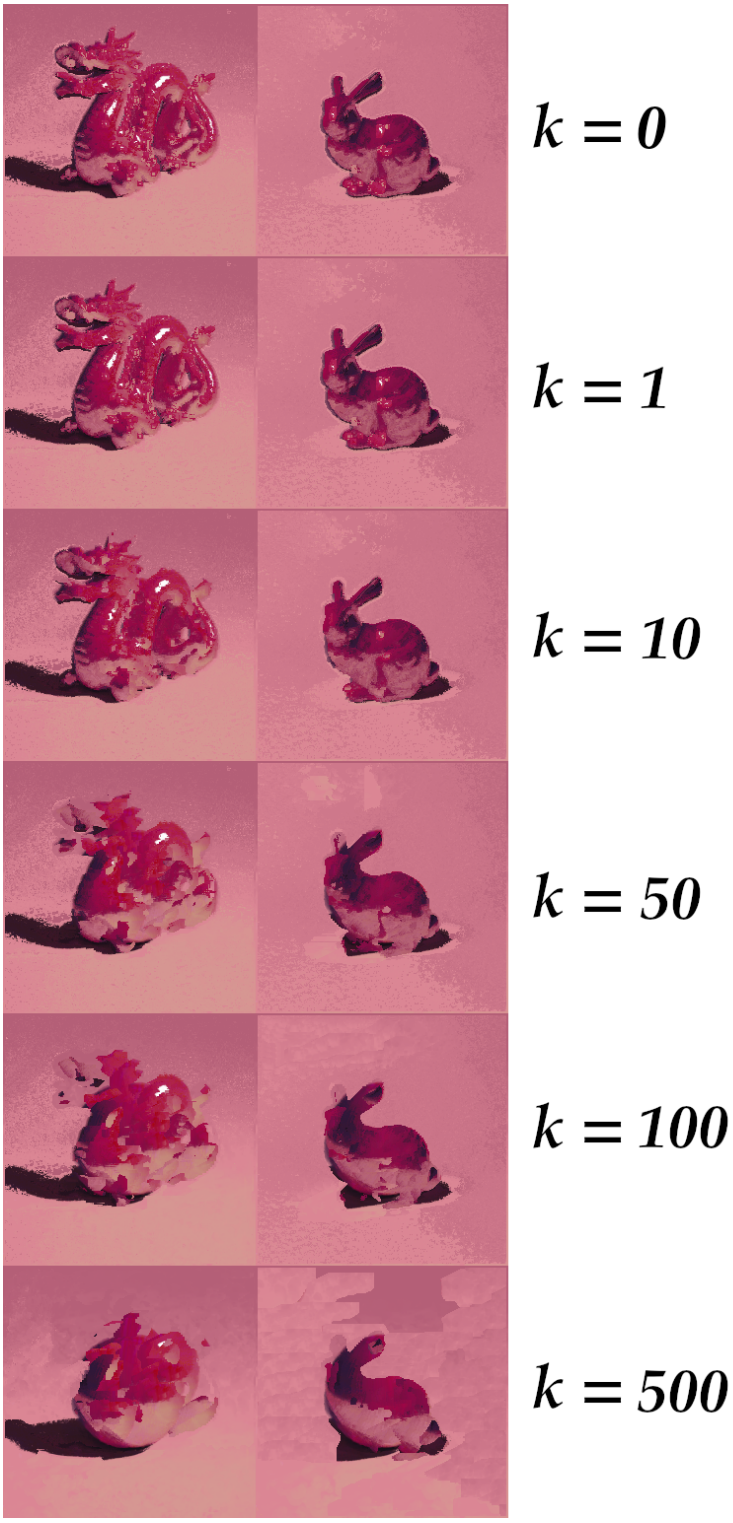


Figure 2: Influence of κ over the synthesized picture.

I did not attempt to optimize performance, but still reached usable results : my single-threaded, CPU-only program can synthesize a 256x256 image with 6 pyramid levels based on 8 light passes in 15 seconds. I also synthesized 1024x1024 images with the same parameters in about 5 minutes.

More than half of the execution time is spent building the kd-tree data structure. In a production setting, this step could be cached, as it depends only on A : the user could then change A', B and the program settings without as high of a penalty. For large resolutions and channel counts, up to 40% of the remaining time may be spent in nearest neighbor queries. Accelerating those is therefore critical. ANN provides a parameter ϵ which allows for faster queries, at the cost of accuracy : the resulting point's distance is guaranteed to be a distance $(1 + \epsilon)d_{actual}$ away from the query point, where d_{actual} is the actual distance to the nearest neighbor. Figure 3 shows results with multiple values of ϵ. Increasing ϵ increases performance with minimal perceptual differences in the final result : as such, it is a good way to create quicker previews before the final run of the program.

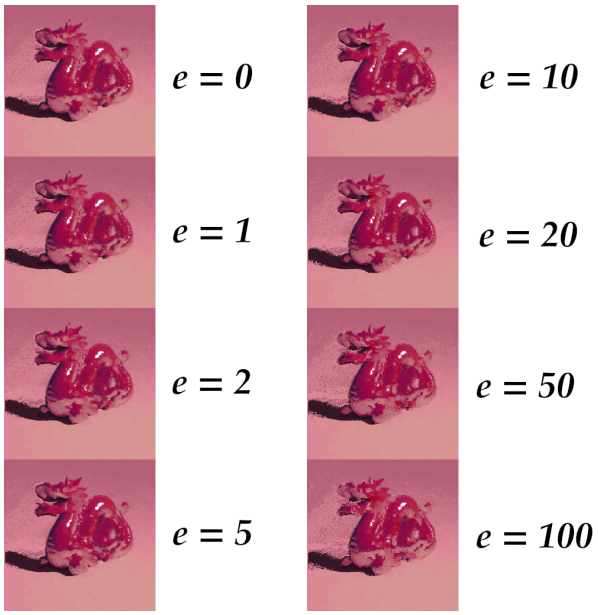


Figure 3: Influence of ϵ over the synthesized picture.

4 Acknowledgements

My warmest thanks go to :

- Sofia Robert for her examplar digital painting
- Camille Huynh and Juliete Rossie for fruitful discussion about the project
- The Stanford scanning repository for the 3D models used