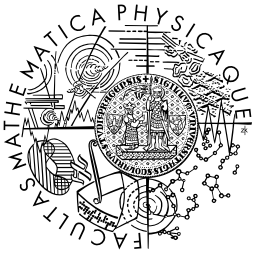

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Filip Hauptfleisch

Interactive Example-based Stylization of Architectural Models

Department of Software and Computer Science Education

Supervisor of the master thesis: prof. Ing. Daniel Sýkora, Ph.D.

Study programme: Computer Science

Study branch: Computer Graphics and Game Development

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedicated to Jaroslav Křivánek, who was a supervisor of this thesis for most of the time. Without him and his passion for computer graphics, this project would never arise.

I am also grateful to everyone who was part of this project, most notably prof. Ing. Daniel Sýkora, Ph.D. for his valuable advice, Ondřej Texler for his help and everybody at Chaos Czech a.s.

I would also like to thank my friends and family for supporting me.

Title: Interactive Example-based Stylization of Architectural Models

Author: Filip Hauptfleisch

Department: Department of Software and Computer Science Education

Supervisor: prof. Ing. Daniel Sýkora, Ph.D., Department of Computer Graphics and Interaction, Faculty of Electrical Engineering, Czech Technical University in Prague

Abstract: One of the ways in which architects communicate their designs is through hand-made architectural sketches. Often it would be beneficial to have a sketch drawn from several angles, but producing many sketches is time-consuming for the architect. Another interesting visualization tool could be an interactive smooth rotation around a stylized scene, which would be impossible to do by hand.

Style transfer is an area from non-photorealistic rendering. It tries to solve the problem of transferring a style from an image to another one. This work introduces the current state of style transfer and compares several methods of transferring an architectural sketch style from one view of a 3D scene to another view of the same scene. The StyLit algorithm is then expanded to run the style transfer for multiple viewpoints at the same time while maintaining spatially coherent results. Finally, a method to run an interactive smooth movement around the stylized scene is introduced.

Keywords: style transfer, architectural design, 3D models, animation,

Contents

Introduction

Non-photorealistic rendering (NPR) is a well-established area of research in the field of computer graphics. In NPR the goal is generating stylized imagery in a wide variety of styles such as cartoon animation, mimicking different artistic media, cell shading, technical illustrations or architectural sketches. One of the specific subareas of NPR is style transfer, which focuses on transferring style from one image to another scene. In this thesis, we will focus on style transfer in the specific setting of architectural sketches.

The style of architectural sketches is quite unique. Although the styles can differ a lot, there are some unifying features present in the majority of the sketches. These features which make them different from other artistic styles include emphasis on straight lines, emphasis on perspective and attention to detail in important areas while omitting details in other areas. All these features cause that some of the style transfer techniques fail on these sketches.

Architects often sketch a building according to a 3D scene they have already modeled as a mean of visualization of their design. But producing sketches is a tedious and time-consuming activity. That makes producing sketches from many views unfeasible. There is also an unexplored area of real-time style transfer in architecture. It might be interesting for the presentation of architectural projects to have a scene stylized in a certain way that allows interactive movement.

We aim to introduce techniques that would allow transferring a style from a sketch from one view of a 3D scene to different views of the same scene. We also aim to generate these stylized results in a spatially coherent way. Then we want to enable interactive interpolation between these stylized views.

Goals

First, we aim to discuss current approaches to style transfer and find how they work in our settings of architectural sketches. We will focus on the most prominent approaches. We aim to choose the algorithm that we deem as the most viable for transferring style from an architectural sketch of one view to other views of the same 3D scene. This algorithm needs to be able to transfer the highly structured architectural sketches. It should also allow us to use the different data we can get from a 3D scene.

Next, we want to expand this algorithm to produce results on multiple viewpoints in a spatially coherent way. This means that we want to be able to produce results of high quality, which we want not to flicker when moving between two of the stylized viewpoints.

Lastly, we want to come up with techniques that would allow an interactive movement between the stylized viewpoints while maintaining a high quality of the stylization. This poses several constraints, as this technique needs to be fast enough to run in real-time, while producing results of high visual quality.

Outline

In the first chapter, we will introduce some of the current methods in the area of style transfer of images and videos. We will describe some of the most important

algorithms and discuss their strengths and weaknesses. In the second chapter, we will introduce the StyLit algorithm in detail, since we chose it as the base for the work in this thesis. We will also introduce a few small changes to the StyLit algorithm which aim to improve the results for architectural sketches. In the third chapter, we present a way to run the modified StyLit algorithm on a grid in space in a spatially coherent way and we introduce ways to interactively compute the stylized result in between the points of the grid. The fourth chapter provides details on the implementation and the fifth chapter shows results of all the implemented algorithms.

1. Current methods of style transfer

Style transfer is a research area in non-photorealistic rendering (NPR). In style transfer problem, there is a style image A' and a content image B. The goal is to produce an image B' that has the content of B but the style of A'. The problem of style transfer could be generalized as a guided texture synthesis problem, where the produced texture should be similar to the style image A', but it should be guided by B. The field of style transfer can be useful in various areas.

One of these areas is movie production. A lot of work is put into animated movies and TV shows and style transfer techniques could help animators. In some cases, the animation is drawn over a crude 3D animation so the underlying 3D geometry can serve as the content, while some drawn keyframes can serve as the style. This case is addressed by Bénard et al. [2013]. Another technique used in the movie industry is rotoscope animation. The technique of rotoscoping means shooting live-action scenes that are then painted over frame by frame. This technique can help in achieving interesting results, in which realistic paintings mix with unreal things, but this technique is also very time-intensive. One example of this technique is Undone (Raphael Bob-Waksberg [2019]), a series by Amazon Prime Video. In the case of rotoscoping animation, an algorithm that could transfer a style from a few keyframes to a video sequence could help to make this technique less time-consuming. One of the algorithms addressing the problem of transferring a style from several keyframes to a whole video sequence was introduced by Jamriška et al. [2019].

Another popular application of style transfer is the use as an image filter. It tries to answer the question: "What would my photo look like if it was painted by a famous artist?" There are several websites and applications dedicated to this, such as Deepart.io[1] or Deep Art effects[2]. These websites allow the user to upload their photo (content image B) and their style image (A') from which the website computes a stylized result of the photo (B'). With the arrival of such applications on mobile devices (one of such applications is Prisma[3]), this field is gaining even more exposure. Another related use is in interactive filters for mobile devices, which are popular on social media. These filters can change the output from the camera in real-time to look as if stylized in a certain artistic style.

In this chapter, we will look at current algorithms used for style transfer.

1.1 Arbitrary style transfer

The methods that address arbitrary style transfer aim to transfer a style from an arbitrary style image A' to an arbitrary content image B. These images are not required to have any shared characteristics. Usually, B is a photograph and A' is an artwork.

[1] https://deepart.io (Accessed December 21, 2019)

[2] https://www.deeparteffects.com (Accessed December 21, 2019)

[3] https://play.google.com/store/apps/details?id=com.neuralprisma&hl=en (Accessed December 21, 2019)

1.1.1 Patch-based methods

Patch-based methods are based on working directly with the pixel values of the images. For each pixel in the synthesized image, these algorithms use a patch around the pixel to decide what value the synthesized pixel should have. These local patches are used to transfer the texture of the style, while color is transferred globally.

One of these methods is the one introduced by Frigo et al. [2016]. It transfers the texture in form of luminance, while color and contrast is transferred globally. It adaptively splits the content image into a quadtree. The splitting is based on variance on the pixels in the current patch and on the distance of the current patch to the closest patch in the style image. For each of the patches in the quadtree, correspondence to a patch from style image is found. As the last step, the global color and contrast are transferred. The issue with this approach is that it establishes correspondence among patches based on the distance of the values of the pixels in A' and B directly. But a small error between a patch from the content image and a patch from the style image does not mean that the patch is a good fit (a dark patch in the content image might be better represented by a bright style patch).

Temporal coherence

The approach from the previous section can be extended to transfer arbitrary style from a style image to a video sequence, as shown in another paper by Frigo et al. [2019]. The stylization of a video sequence is not as straightforward as stylizing all the individual frames. That is because temporal coherence plays an important role in the stylization of video sequence. In this generalization, the authors propose first propagating coherence across keyframes taken at uniform time intervals and then propagating these stylized keyframes to frames between them.

1.1.2 Neural-based methods

Recently, style transfer using neural networks started gaining more attention. This is a relatively new field, which started with the paper by Gatys et al. [2016]. The use of neural networks for style transfer is difficult because there are no datasets for straightforward training of style transfer. While we have access to large databases of artwork, we do not have the underlying content images for them. That is why these neural methods use already existing neural networks trained for different purposes. These methods then perform the synthesis using features from different layers of the neural networks.

Imag Style Transfer This paper by Gatys et al. [2016] introduced the idea of using neural networks for artistic style transfer. Because of the lack of datasets mentioned before, they took a different approach. They came up with the idea of using a network trained for image recognition (a deep convolutional neural network by Simonyan and Zisserman [2014]) for style transfer. The algorithm is based on the idea of separating the semantics of an image (the content) from the textural data (the style). If the algorithm could separate these two parts, it

could perform the style transfer by combining the texture of the style image and the semantics of the content image. It does so by the assumption that in the higher levels of the image recognition neural network, the features represent the content of the image well, so these features are used. Style is extracted as texture capturing feature space introduced in another paper by Gatys et al. [2015]. The resulting image is synthesized by minimizing the distance to the content statistics of the content image and the style statistics of the style image.

The approach can generate impressive results on some styles but fails on the highly structured architectural sketches styles. Results from this algorithm on selected architectural problems can be seen on Figure 1.1 a.1) – a.3). These results were generated using the Deepart.io website [4], which provides results based on an improved implementation of the algorithm in the paper. It can be seen that this algorithm is not suitable for our purpose since it fails to preserve the look of the original style in all the cases we tried.

Style Transfer by Relaxed Optimal Transport and Self-Similarity One of the many follow up papers trying to improve the results of the original Gatys et al. [2016] paper is Style Transfer by Relaxed Optimal Transport and Self-Similarity by Kolkin et al. [2019]. This paper uses the same reasoning, but the error terms differ. They use a content loss based on self-similarity. We present results of this paper from a web demo of the algorithm by the authors [5] in Figure 1.1 2.a) – 2.c). The results of this algorithm also generally do not resemble the original style in the architectural sketches we tried.

Temporal coherence

The neural methods from the previous section are not suitable for the stylization of video, because they are not temporally stable. That means that there may be a lot of changes even in two subsequent frames that are almost the same. That produces unwanted flickering when a video sequence is stylized. This problem was addressed by Gupta et al. [2017] and Ruder et al. [2018], where models enforcing temporal coherence were introduced.

1.2 Guided style transfer

Another group of style transfer algorithms is the category of guided style transfer. For these algorithms, the assumption is that the style image and content images are semantically close to each other. These algorithms also usually use additional guiding information, which help to perform the style transfer in a semantically meaningful way.

1.2.1 Patch-based methods

Patch-based methods work on the local scale of pixels. Some of the algorithms transfer only the texture locally and the color globally, but some transfer the textural information together with the color information in a local way. Most

[4] https://deepart.io (Accessed December 21, 2019)

[5] http://style.ttic.edu (Accessed December 28, 2019)

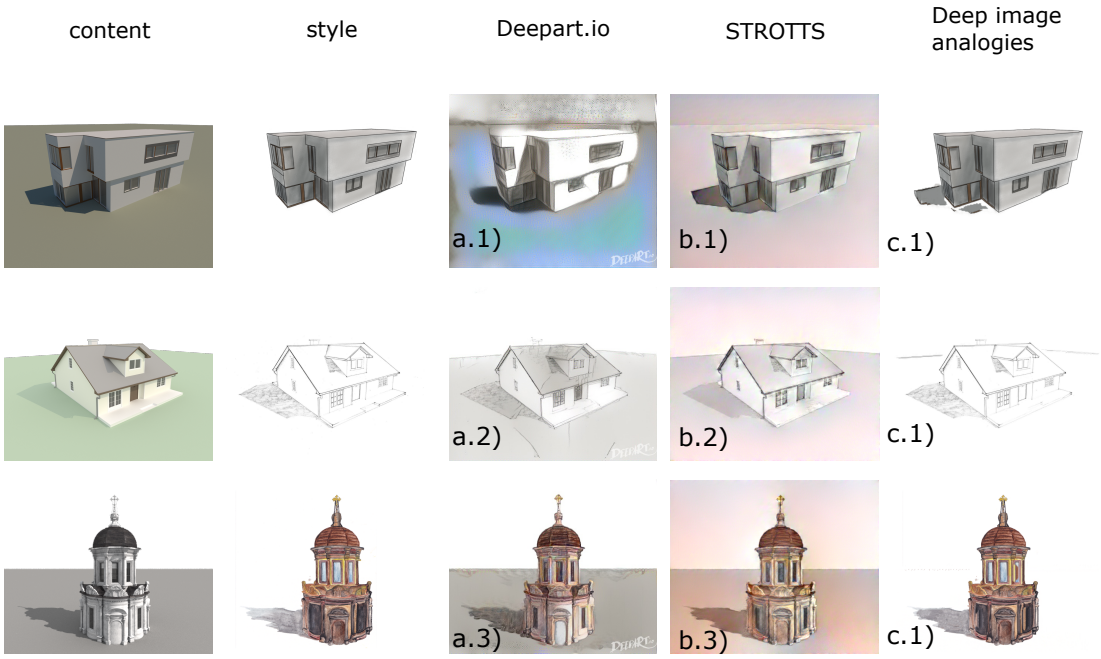

Figure 1.1: Neural algorithms Results
results of several neural style transfer algorithms on identity (content is from the same view as the view the style was painted from).

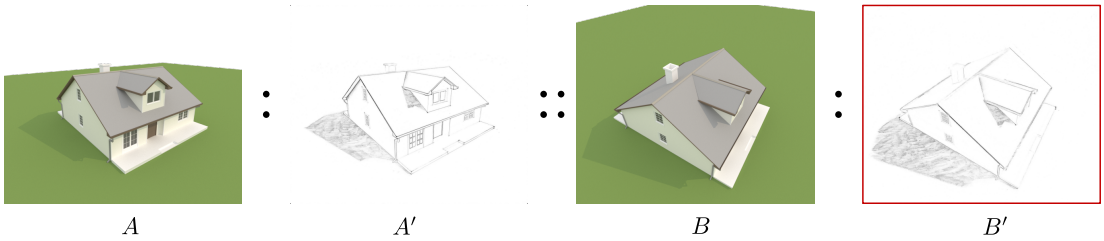

Figure 1.2: Image analogies
The image analogies problem, with B' being the synthesized image.

of these algorithms are based on texture synthesis algorithms since style transfer can be understood as guided texture synthesis, where a texture with the original style is synthesized according to the content image.

Image Analogies This paper by Hertzmann et al. [2001] does not address only style transfer, the authors introduced more general problem:

Definition 1. *Given a pair of images A and A' (the unfiltered and filtered source images, respectively), along with some additional unfiltered target image B, synthesize a new filtered target image B' such that*

$$A : A' :: B : B'.$$

In other words for images A, A' and B, the problem is to find an image B' that is in the same relation to B as A' is to A (see Figure 1.2).

The algorithm works on multiple scales, from coarsest to finest with results from each level upscaled and used as a base for the next level. For each pixel in

the currently synthesized image, the algorithm finds the best pixel from the style image, while preferring pixels preserving the coherence of the texture.

In the specific case of style transfer, A would be source content image, A' would be source style image, B would be target content image and B' would be the wanted target style image. The content images A and B are not necessarily just RGB images, they can contain multiple channels (more than the typical three or four channels) of information.

With this definition of the problem, we require additional information for the style A' in the form of the content image A. Because of this, it may be difficult to use currently available artwork, since we would need to generate this content image for it. In the paper, a blurred version of the style image is used as the content image A and photograph as the content image B, but this does not work for all styles.

StyLit This algorithm by Fišer et al. [2016] is a current state-of-the-art technique in patch-based guided style transfer. The algorithm is based on the Texture Optimization for Example-based Synthesis algorithm by Kwatra et al. [2005], but it is formulated specifically for style transfer on 3D scenes. We describe this algorithm in detail in Chapter 2.

StyleBlit This algorithm by Sýkora et al. [2019] is capable of producing images of comparable quality to the StyLit algorithm but in real-time. The algorithm works by searching large coherent chunks in the style exemplar which are then copied to the synthesized texture.

But this approach is not suitable for architectural sketches. That is because it has problems in large areas which have fairly constant guides. This is a limitation that makes this approach inappropriate for artistic sketches since models of buildings often contain large flat areas that are fairly uniform. In all these areas, this algorithm fails.

Temporal coherence

Example-based synthesis of stylized facial animations This paper by Fišer et al. [2017] presents a method specialized in style transfer of videos with faces in them. In this paper, the authors do not aim to produce a sequence with full coherency, instead, inspired by Fišer et al. [2014], they want overall coherency but with some flickering that would resemble hand-drawn animation.

Stylizing Animation By Example In this paper by Bénard et al. [2013], the authors extend the Image analogies (Hertzmann et al. [2001]) algorithm for animations. This paper addresses a problem similar to ours. They expect a 3D underlying geometry of an animation and they want to make a stylization in a temporally coherent way. They expand the original Image analogies algorithm by changing the error term to improve the results, including an error term enforcing temporal coherency.

The algorithm works on multiple scales same as the original Image analogies algorithm. On each level, the whole sequence is stylized twice, once sequentially

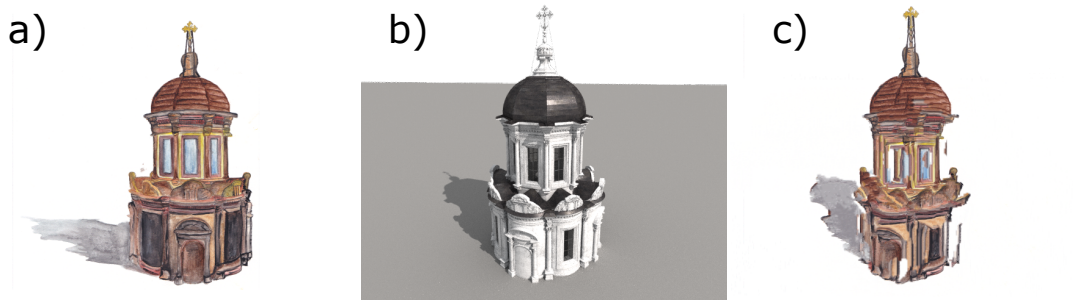

Figure 1.3: Deep image analogies result on different view
a) original style b) content c) Deep image analogies result

in a forward manner and then sequentially in a backward manner with coherence enforced in the respective direction by shifting resulting frames according to motion fields.

Stylizing Video by Example This paper by Jamriška et al. [2019] describes a method for style transfer for videos. In this paper, the authors do not require a 3D scene underlying the frames. That means that this method does not have access to motion fields. Instead, they use an approximation based on SIFT Flow by Liu et al. [2010]. For the stylization itself, they use the StyLit algorithm with several generated guides. The sequence is stylized sequentially and the result of each frame is shifted to the next frame using the approximated motion field. These shifted results are then used as coherence guides.

1.2.2 Neural-based methods

There are also neural-based methods that expect the style image A' and the content image B to be semantically close.

Visual Attribute Transfer Through Deep Image Analogy One of these methods is described in the paper by Liao et al. [2017]. This method aims to solve similar problem to the image analogies problem introduced in Definition 1. But instead of producing B' for given A, A' and B, this method expects A and B' as input and produces A' and B. The method expects that A and B' have a semantically similar structure and the algorithm tries to find a correspondence between A and B' through A' and B. The results of this method are in Figure 1.1. It can be seen that this method has significantly better results than any of the other tested neural approaches, but it still suffers from several issues. It does not honor artistic choices that are not based on the underlying picture, e.g. the shadow in Figure 1.1 c.1), which is forced by the image analogy, although the artist chose not to draw the shadow, or the line on the horizon in Figure 1.1 c.2). It also does not transfer texture well, as in the watercolor style shadow. But because it shows better results on the identity, we also provide a result of this algorithm on different view in Figure 1.3. In this figure, it can be seen that even this approach fails on different views. The overall structure breaks down in this case.

Image-to-Image Translation with Conditional Adversarial Networks Another method was presented by Isola et al. [2017]. This is not a model proposed for style transfer, but rather for a general image-to-image translation, such as a black and white photo to colored, an aerial photo to a map, etc. This method is based on training conditional generative adversarial networks. For use on style transfer, the network would have to be trained on a certain artistic style. But this method is not suitable for our use, because it requires a dataset of pairs to train. A method by Zhu et al. [2017] removes the need for paired datasets.

Temporal coherence

In video to video synthesis by Wang et al. [2018] and the follow-up work (Wang et al. [2019]), a new model for producing a video sequence out of input source and content sequence is introduced. Temporal coherence is implicitly included in the model.

2. StyLit algorithm

In this chapter, we will describe the StyLit algorithm by Fišer et al. [2016] in detail and introduce a few small changes to it. It is a patch-based style transfer algorithm with very promising results. This algorithm without any changes is already capable of producing results of impressive quality on many different styles (see Figure 2.1 and Figure 2.2), but it still has some problems in certain scenarios. Based on these results and our requirements we chose to use the StyLit algorithm, as it is the current state-of-the-art patch-based style transfer algorithm.

2.1 Underlying texture synthesis algorithm

The StyLit algorithm is based on an example-based texture synthesis algorithm by Kwatra et al. [2005]. In this section, we will introduce this underlying texture synthesis algorithm.

It is an algorithm for example-based texture synthesis, so for the given texture sample, the algorithm aims to generate a new sample with the same visual quality as the original, but usually of a bigger size.

The algorithm expects Markov random field (MRF) property on the textures, so it expects locality and stationarity. Locality means that a pixel color is dependent only on the colors of pixels in its neighborhood. Stationarity means that the pixel color is independent of the location of the pixel in the picture. This allows the introduction of an energy function on the synthesized texture and then minimization of this energy in an Expectation-maximization (EM) like algorithm.

Algorithm

The algorithm defines a similarity measure between the synthesized image and the original image. This similarity measure is based on local neighborhood similarities combined into a global measure, which is then minimized.

The algorithm expects a source texture A' and produces a target texture B'. For each pixel p, we will denote the neighborhood around p by N_p. A vectorised version of B' will be denoted by $\boldsymbol{B'}$ and it will contain pixels from B' in a vector. For pixel p we will denote by $\boldsymbol{x_p}$ a vector of all the values from N_p. The closest neighborhood from A' to a neighborhood $N_p \in B'$ (using Euclidean measure) in

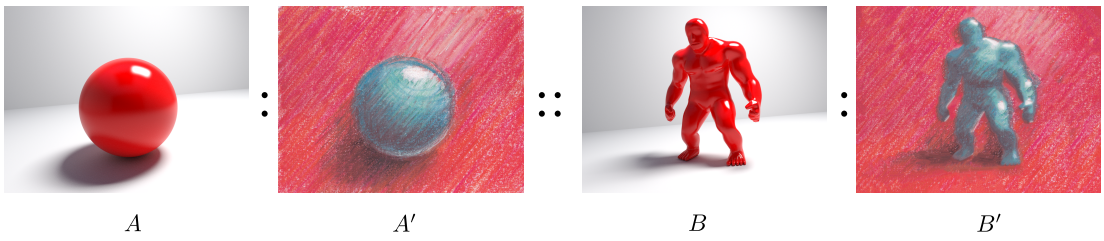

Figure 2.1: StyLit result
Results of the StyLit algorithm, with the sphere scene in A, source exemplar in A', target scene in B and stylized result in B'. Style exemplar and the guides are from the original StyLit paper.

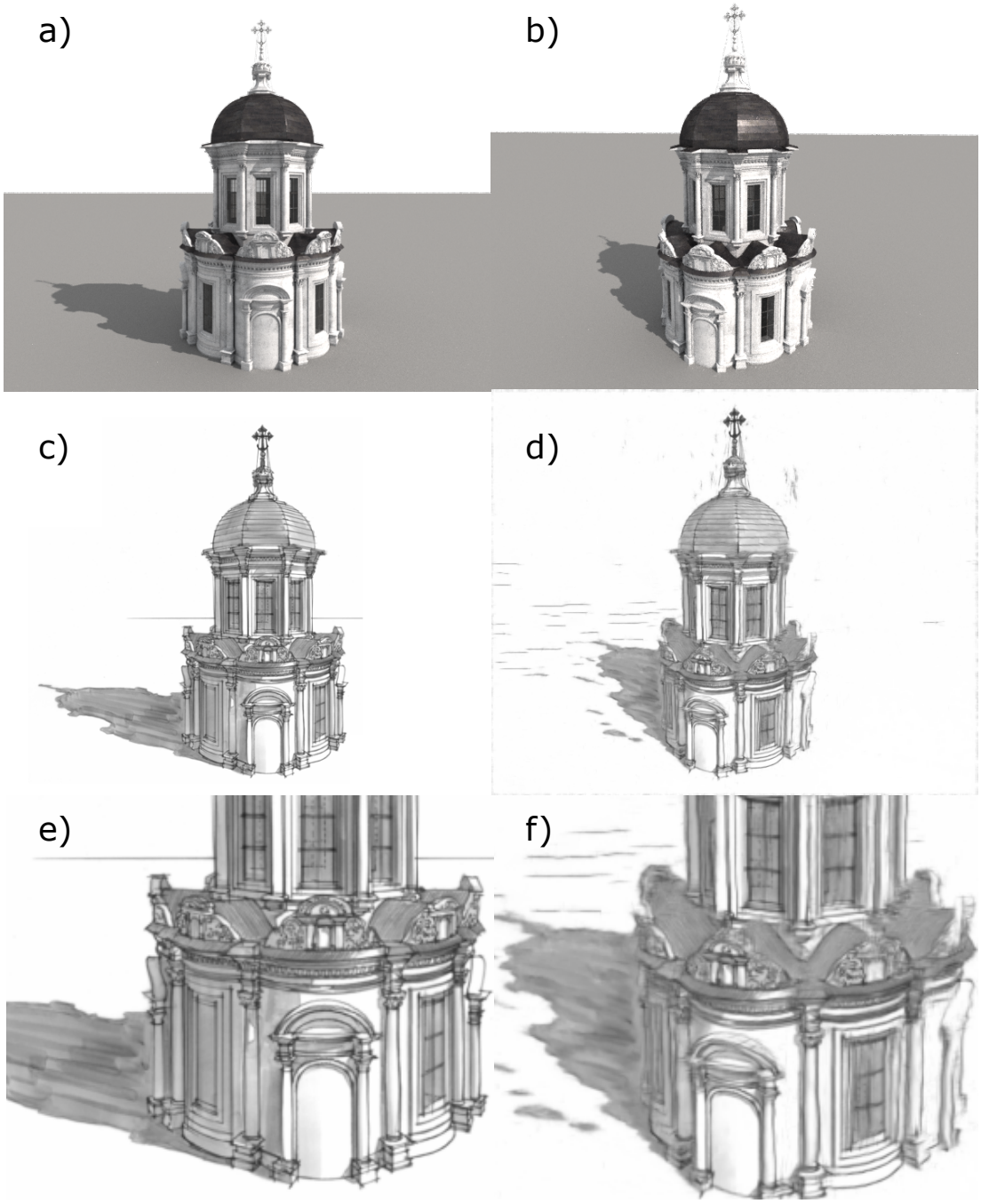

Figure 2.2: StyLit result on architecture
a) original view b) new view c) original style d) synthesised image e) original style detail f) synthesised image detail

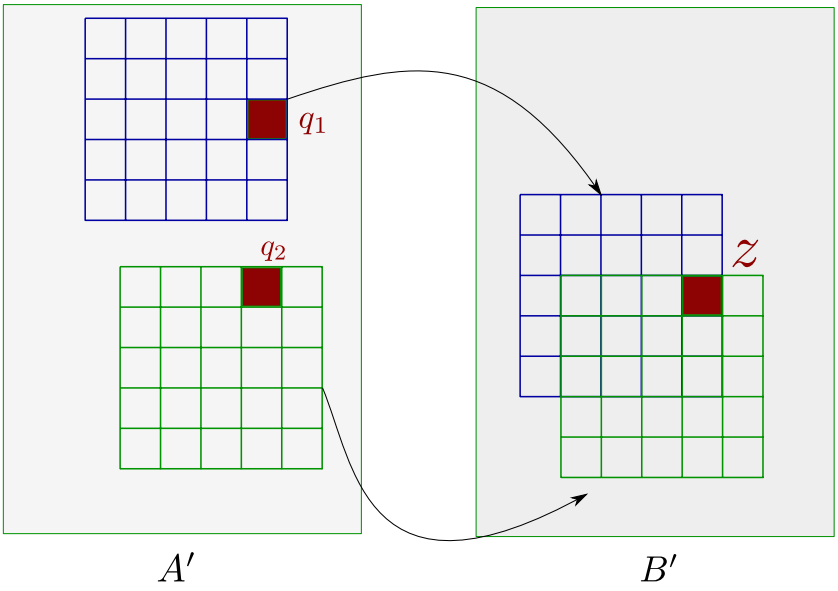

Figure 2.3: E-step
For simplicity there are only two neighborhoods in the target image B'. These two patches have their closest patches from A' in the same color and the correspondences are illustrated by arrows. In this case, z is part of these two neighborhoods, so the value of pixel z in B' will be an average from q_1 and q_2, since they are on the corresponding positions of the blue, respectively green patches.

a vectorized form is denoted by $\boldsymbol{c_p}$. The global energy of the target B' is defined in the paper as

$$E = \sum_{p \in B'} \|\boldsymbol{x_p} - \boldsymbol{c_p}\|^2. \tag{2.1}$$

The algorithm then iteratively minimizes this energy. It does so in a manner similar to the EM algorithm. It runs in two steps, where the E step minimizes the energy with respect to $\boldsymbol{B'}$ and the M step minimizes the energy with respect to the set $\{\boldsymbol{c_p}\} = \{\boldsymbol{c_p} : p \in \boldsymbol{B'}\}$.

E-step To minimize the energy with respect to $\boldsymbol{B'}$ with fixed $\{\boldsymbol{c_p}\}$, the derivative of 2.1 with respect to $\boldsymbol{B'}$ must be placed equal to zero. Solving the resulting set of equations leads to

$$\boldsymbol{B'}[i] = \frac{1}{|\{p : \boldsymbol{B'}[i] \in N_p\}|} \sum_{p:\boldsymbol{B'}[i] \in N_p} \boldsymbol{c_p}[j],$$

where j is index of the pixel $\boldsymbol{B'}[i]$ in the neighborhood $\boldsymbol{x_p}$. For pixel i that means the value minimizing the energy is average of values of the pixels at the same position as $\boldsymbol{B'}[i]$ from all the patches $\boldsymbol{c_p}$ for which i is part of p. See Figure 2.3.

M-step In the M step, the energy is minimized with respect to $\{\boldsymbol{c_p}\}$ with $\boldsymbol{B'}$ fixed. For each pixel p and its neighborhood N_p in the image B', the algorithm finds closest neighborhood from A' which is saved as $\boldsymbol{c_p}$.

The authors of the paper suggest running this algorithm on multiple scales. From coarse to fine level with the upscaled results used as a base for the next level.

This algorithm can be modified to perform controlled synthesis, by modifying the error function to account for the guide channels (A and B in the terminology of Image analogies). The base error metric is the difference between neighborhoods in the style images, so if we add the difference between the neighborhoods in the content images, we get a style transfer algorithm.

2.2 Synthesis data

The StyLit algorithm is designed to transfer style from one 3D scene (source scene) to another one (target scene). The algorithm transfers style from one scene to another by using data from path tracing renderers. In this section, we will look at the data this algorithm uses.

2.2.1 3D scenes

In the paper, the authors propose a source scene that could capture enough information to be generalizable to other arbitrary scenes. The idea behind this is that this source scene could then be painted and the style of the artist could be transferred to arbitrary other scenes.

This idea is similar to the approach used in the Lit sphere by Sloan et al. [2001], where a sphere is used to capture information about artistic style. In the StyLit paper, a sphere is also used, but it is placed on a ground plane. The addition of the plane allows capturing additional information about shadow an object casts, which can be stylized in a different way. This allows transfer to target scenes where objects are placed on the ground. For this basic scene see A in Figure 2.1.

2.2.2 Guide channels

In order to perform the style transfer, StyLit algorithm uses guide channels to run the synthesis of the stylized texture. Even though the algorithm uses data from 3D scenes, the style transfer itself runs on 2D images. Guide channels, in this case, are images rendered from the underlying 3D scenes. These guide channels control the style transfer, so the basic requirement for them is that if guide channels in an area of the source image are similar to the guide channels in an area of the target image, the stylization in these areas should also be similar.

Thanks to the availability of the underlying 3D scenes, the algorithm can use all the data that a 3D path tracing renderer can provide. The guide channels should contain data that can help guide the synthesis well, so the best choice would be to extract the information that artists emphasize.

In the paper, the authors suggest using several light path expressions (LPEs) channels. These LPEs can be used to get multiple different light interaction patterns separated, such as direct, indirect or reflected light. In the paper, the authors provide a detailed analysis of the channels they used, but the style transfer algorithm itself can be used with any guides.

In the language of the image analogies problem from Definition 1, all the guide channels represent the image A, the original style is A', target guide channels are B and the stylized result is B'.

2.3 Synthesis algorithm

Since the algorithm is based on the texture synthesis algorithm by Kwatra et al. [2005] as introduced in Section 2.1, it also minimizes the following energy:

$$E = \sum_{p \in \boldsymbol{B}} \|\boldsymbol{x_p} - \boldsymbol{c_p}\|^2, \tag{2.2}$$

where $\boldsymbol{B}$ is the target image consisting of the guide channels and style, $\boldsymbol{x_p}$ is a vectorized neighborhood around p and $\boldsymbol{c_p}$ is a source vectorized neighborhood, that is closest to $\boldsymbol{x_p}$. A vectorized neighborhood is a vector of the data in all the pixels from the neighborhood in a scanline order. In these vectorized neighborhoods, there are values from all the channels, that is, all the guide channels and the style.

As explained in Section 2.1, this can be achieved by running an iterative algorithm similar to an EM algorithm.

But using just the texture synthesis algorithm expanded by guides produces a lot of blurring. This blurring is caused by smooth patches from the original style image being overused in the synthesized image. These smooth patches then cause blurring of the result. The reason why these smooth patches are preferred in this situation was described by Newson et al. [2014]. To overcome this issue StyLit uses a different approach in the E-step of the algorithm inspired by reversed NNF retrieval. This technique was also used in LazyFluids by Jamriška et al. [2015] for this purpose.

2.3.1 Search

This part of the algorithm establishes a correspondence between patches in the target image and patches in the source image. For each patch in the target image, exactly one corresponding patch in the source image is found. The metric used to get the error between two patches is (as defined by Jamriška et al. [2019])

$$E(A', B'_i, G^S, G^T, p, q) = ||A'(p) - B'_i(p)||^2 + \sum_{g \in \mathbb{G}} \lambda_g ||G^S_g(p) - G^T_g(q)||^2, \tag{2.3}$$

where A' is a source style, B'_i is current stylized result from i-th iteration, G^S are source guides, G^T are target guides. With the patch error written like this, the minimized energy from Equation 2.2 can be written equivalently

$$E(A', B'_i, G^S, G^T) = \sum_{q \in \mathbb{Q}_\mathbb{T}} E(A', B'_i, G^S, G^T, p_q, q), \tag{2.4}$$

where $\mathbb{Q}_\mathbb{T}$ is a set of all neighborhoods in target image, p_q is a patch from source image currently corresponding to q.

But as explained before, target-source search, which finds the closest patch in the source image for each patch in the target image, produces blurry results, so

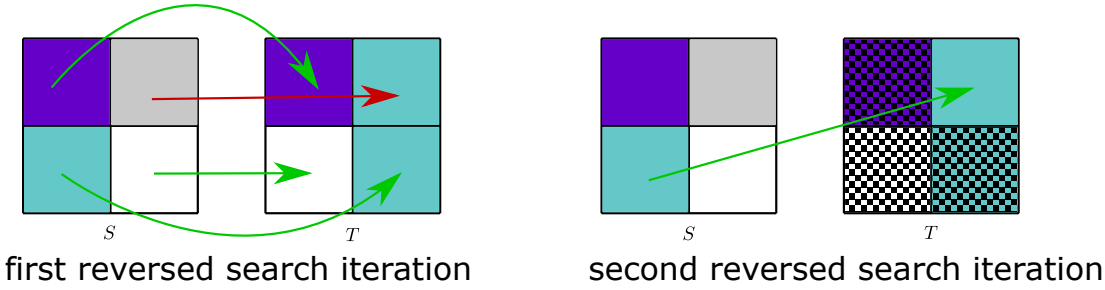

Figure 2.4: Search iteration
Two iterations of the reversed search. Green arrows represent accepted patches, red arrows rejected patches. In the second iteration, the checkered areas are already fixed from the first iteration.

the approach of StyLit is reversed – it searches for the best patch in the target image for each patch in the source image (we will call this the source-target search).

If the errors of all these correspondences are sorted from lowest and plotted in a graph, the authors of the paper claim that the graph resembles a hyperbolic function.

A hyperbolic function $f(X) = (a - bx)^{-1}$ is fitted to the data and a knee point x_k in this hyperbola is found as a point where $f'(x_k) = 1$. All correspondences with errors that are above $f(x_k)$ are probably wrong assignments. An error budget T is then set as sum of all the errors with indices lower than x_k. With this error budget they propose maximizing the number of fixed source-target patches $|A^*|$, while satisfying

$$\sum_{p \in A^*} \min_{q \in \boldsymbol{B}} E(A', B'_i, G^S, G^T, p, q) < T. \tag{2.5}$$

After one iteration of this source-target search, not all patches in the target image have fixed correspondences. In the next iterations for each patch in the source image, we find the closest patch from the target image again, but the search ignores patches in the target image that have already fixed correspondence from previous iterations (see Figure 2.4).

The iterations can be run until all patches in the target image have a fixed correspondence, or until a certain percentage of these patches have correspondences and then a target to source search can be performed to find correspondences for the rest of the unmatched patches. After this step, all patches in the target image have fixed corresponding patches from the source image.

2.3.2 Voting

This is the E-step of the EM-like algorithm. It minimizes the energy 2.4 with respect to B' with $\{p_q\}$ fixed. That is the same as in the Texture synthesis algorithm by Kwatra et al. [2005] as explained in Section 2.1. As in the texture synthesis algorithm, an average is used to minimize the energy. For each pixel $B'(i)$ in the target image, all the patches that contain the pixel $B'(i)$. Then, for each of these patches, the corresponding patch from the source image is taken and a position corresponding to the position of $B'(i)$ in the target patch is taken

into account for the resulting average

$$B'(i) = \frac{1}{|\boldsymbol{P}|} \sum_{N_p \in \boldsymbol{P}} \boldsymbol{c_p}(j),$$

where $\boldsymbol{P}$ is a set of all the patches containing the pixel $B'(i)$, j is the position of $B'(i)$ in N_p. For a patch N_p, $\boldsymbol{c_p}$ is the corresponding patch as established in the M-step.

2.3.3 Full algorithm

Apart from the search phase, the StyLit algorithm works as the Kwatra algorithm (Kwatra et al. [2005]), so in order for the stylized image to contain even the larger-scale features from the source style, the algorithm is performed on multiple scales. A Gaussian pyramid of both the source and target images is made and the algorithm first runs on the lowest resolution. The style in the target image is then upscaled and used as a starting style in the upscaled level. In the first search-vote iteration of the lowest resolution level, there is no base style to use so a random noise or white image is used. Since on the lower resolution levels the neighborhood size stays the same, it encompasses a larger area of the original image. Thanks to that, it manages to preserve even larger features from the original style.

2.4 StyLit in architecture

In architectural sketches, it is generally not possible to use the approach with the sphere representing the style. That has several reasons. First, in normal architectural sketches, there are many materials that are conveyed differently by the sketch (such as glass, concrete, vegetation, etc.), so to use a scene as a base for the style, we would need several copies, one for each material. Second problem is that for architecture it is very difficult to convey the geometry. Typical architecture scene has a wide variety of geometric shapes, often mixing round and sharp geometry, which makes the sphere as base geometry type useless, since it cannot convey features such as sharp edges and flat surfaces well. But since the ultimate goal of this thesis is different (we aim to transfer style just from one view of a scene to a different view of the same scene), we do not aim to find a universal style holder scene for architectural sketches. Instead, we just use the original view that we have sketched as a source image, similarly to having a stylized keyframe in video stylization techniques such as Jamriška et al. [2019].

2.4.1 Changes in the StyLit algorithm

In this part, we propose a few changes to the StyLit algorithm described in the previous section that can improve the results on the style transfer in architectural sketches.

Guide selection

The first question we need to address is the selection of guide channels for the style transfer. We use mostly similar channels like the ones in the original paper.

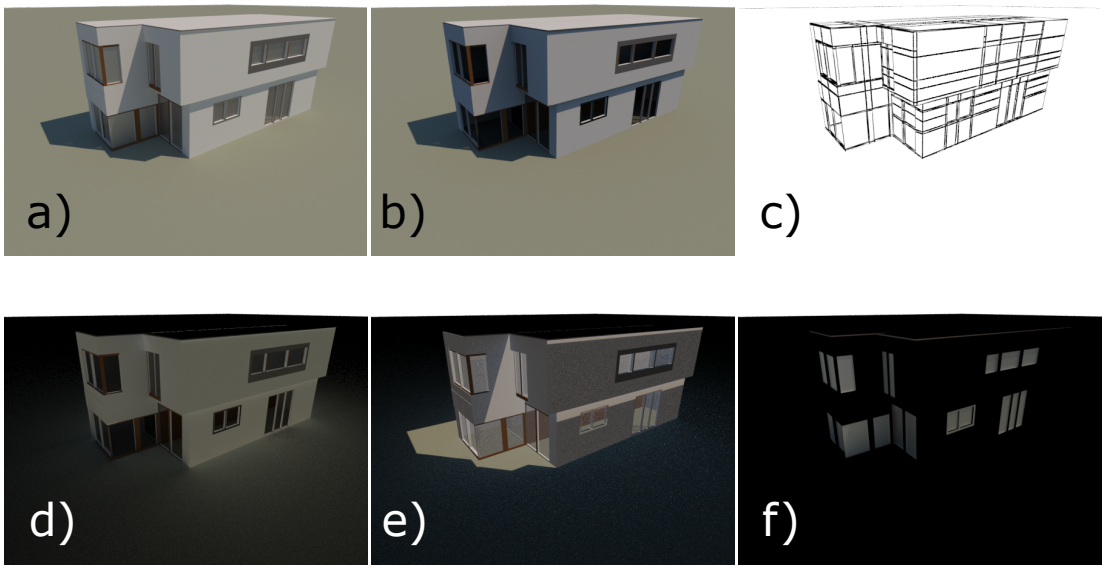

Figure 2.5: Guide channels used for the style transfer
a) beauty render b) direct lightning c)wireframe render d)indirect lightning e) shadows f) reflected light

We use beauty render, direct illumination, indirect illumination, reflected light, shadows and wireframe render (see Figure 2.5). The only notable change is the wireframe render, which was also used by Bénard et al. [2013]. Because architecture depends heavily on straight lines, in order to preserve them, we need some guide to enforce the lines to be where they should be. The wireframe render helps with that since usually straight lines in the sketch are along important lines in the model, which usually have a lot of edges around.

The problem with that render is that it is heavily dependent on the artist who models the scene and thus in some scenes, it may confuse the synthesis algorithm. But we found better results with this wireframe render than with any other approach to preserve straight lines we tried (such as edge detection in different channels). Usually, the wireframe render also conveys the importance of certain areas. If an area is important for the architect, there is more detail in the model, thus making more lines in the wireframe render. In the sketch, emphasis will probably be put on the same areas which were already modeled in detail. The impact of the wireframe guide can be seen in Figure 2.6. It can be seen that the wireframe guide can help enforce continuous lines in appropriate directions.

Error limitation

The StyLit algorithm works well for many styles, but we still found a few problems in the specific field of architectural sketches. One problem that the StyLit algorithm produces is a lot of noise in large areas of one color. This problem is quite noticeable in architectural sketches since it is not unusual to have large uniform areas (usually white), especially in the background. The cause of this issue in the original algorithm is in the source-target direction of search. A patch from the source image that is mostly one color may well have its closest patch in the target image in a uniform area (see Figure 2.7).

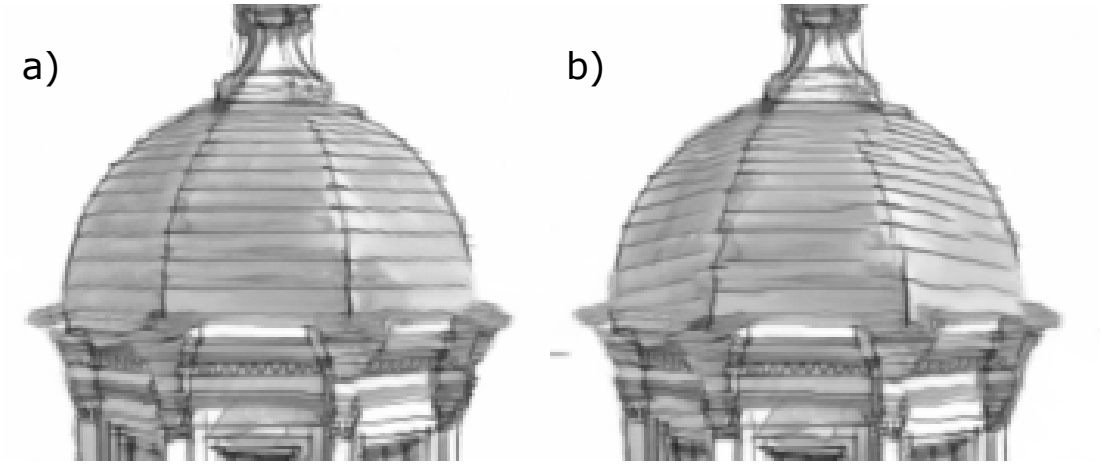

Figure 2.6: Impact of wireframe guide
a) detail of result with wireframe guide b) detail of result without wireframe render

This problem arises from the fact that in the StyLit algorithm error limitation (defined in Equation 2.5) a patch is rejected only if its error does not fit in the error budget. But this takes into account only the absolute value of the errors, and while a patch with a certain error may not introduce any visible artifacts in certain areas (areas with a lot of variance), in other areas a patch with the same error may cause visible artifacts (one-colored areas). An example of how this error may be introduced is in Figure 2.7 in the source-target search. There is just a small error in a uniform patch. Because the error is small, the patch gets fixed and an error is introduced.

One modification we propose to address this problem is as follows. Instead of just looking at the global error budget as in Equation 2.5, we want to also look locally at the current patch. In each search-vote iteration, we perform one target-source direction search in the end to find correspondences for unfixed patches. Because this search is independent of the source-target searches in the same iteration, we may perform this search at the beginning of the iteration instead. From this search, we get the lowest possible error $E_b(p)$ that can be found for each patch (see Figure 2.5). Then we may use this information from the search in the process of fixing correspondences among patches.

The idea is that if the source-target search finds a match that has an error that gets accepted under the error budget condition, but that is much higher than the best error $E_b(p)$, the match is probably a mismatch even though it would get accepted. So for each patch p in the target image, we save the error $E_b(p)$ from the target-source search. For patches that fit under the error budget T, we perform another condition check. This condition ensures that $E(p) < \alpha E_b(p)$ where $E(p)$ is the error from source-target search, and α is a constant (we usually took $\alpha = 5$). This condition ensures that if we know that there is a much better patch to be found, we may want to avoid fixing the current correspondence. So the patch remains unfixed and is considered again in following iterations.

The downside is that this approach may increase the number of source-target searches needed to fill a given percentage of patches and in this way slow down the synthesis. But in reality, the number of rejected patches this way is quite low. As can be seen in Figure 2.8, this approach reduces the amount of noise in

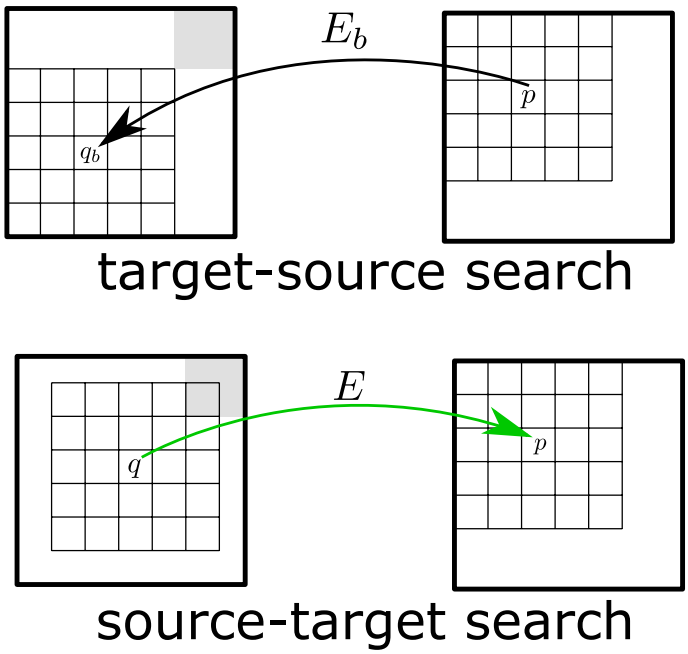

Figure 2.7: Per patch error
Target-source search finds the patch with lowest possible error E_b, then if source-target search returns a mismatch with low error, the result from target-source search may help to avoid fixing it.

the white background quite considerably and in some areas helps to improve the overall quality.

Voting function

Another area where an improvement of quality can be achieved is the voting part of the algorithm. Since the sketches are often done by pencil or ink, they contain a lot of sharp lines. Because of that, the average function in voting may not be the best fit. That is because the average tends to blur the results.

Some of the blurring comes from the following scenario. A correspondence is established between a patch from the target image p and patch from the source image q. These two patches may be a close match in most of the patch area, but on a few pixels of the patches, the difference may be high (see Figure 2.9). This does not necessarily mean that the patch is misplaced since there may not be any better match and in most of the area it fits very well. But the pixels that differ a lot will cause blurring in the voting function.

So ideally, we would want to take just the pixels that are a good match into consideration. But taking just a part of the patch to vote is not possible since there could be a place in the target image which has no good correspondence in the source image. This would cause all the pixels in this area to have high error and thus ending ignored and we would get pixels in the target image with no data.

With this in mind, we still wanted to exclude the pixels that were probably wrong. But instead of excluding them from the patches themselves, we exclude some pixels in the voting function. In the voting function for each pixel, we have a list of pixel values $\boldsymbol{V}$ to be averaged. What we did was computing average

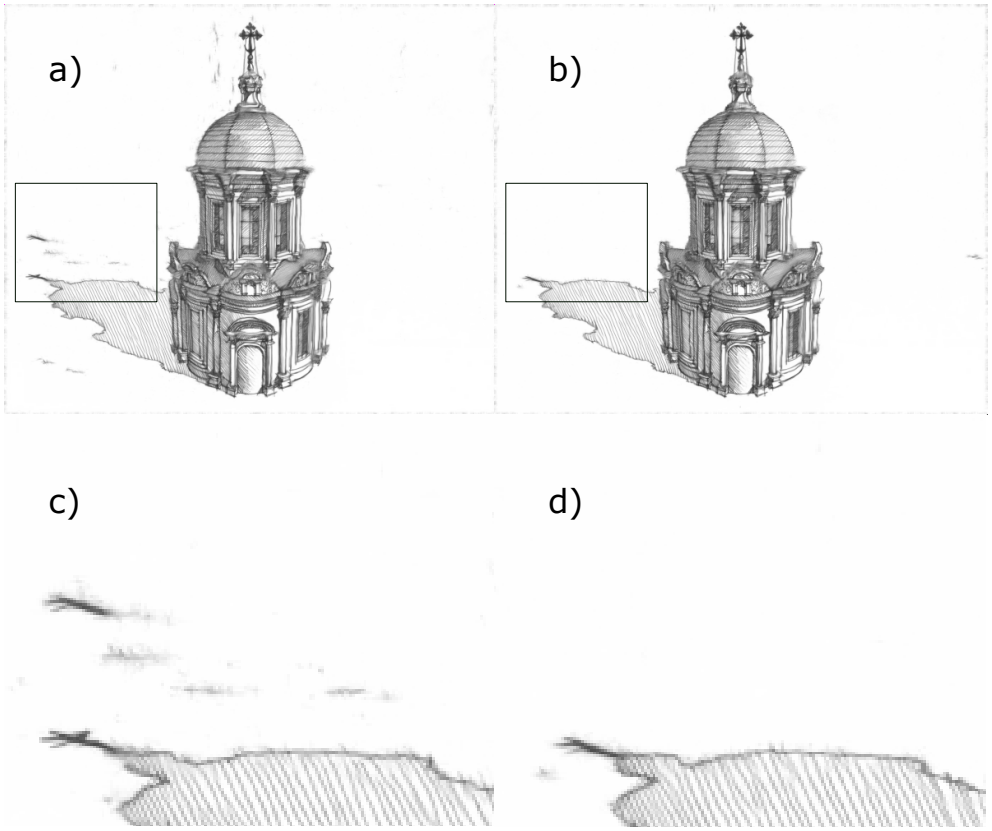

Figure 2.8: Per patch error limitation
a) result without per patch error b) result with per patch error c) result without per patch error detail d) result with per patch error detail

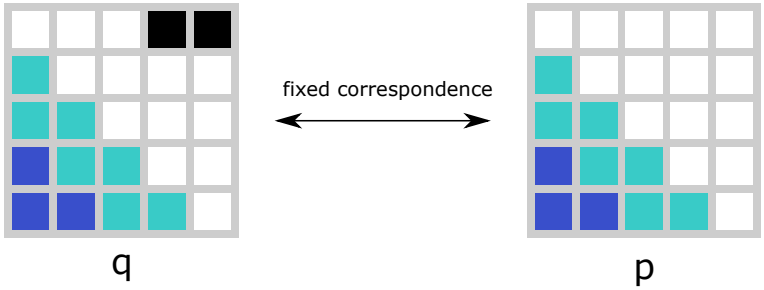

Figure 2.9: Similar patches causing blurring
With a patch q from the source image and p from the target image, there may be error low enough for the correspondence to get fixed, but the two pixels in the top right will cause blurring.

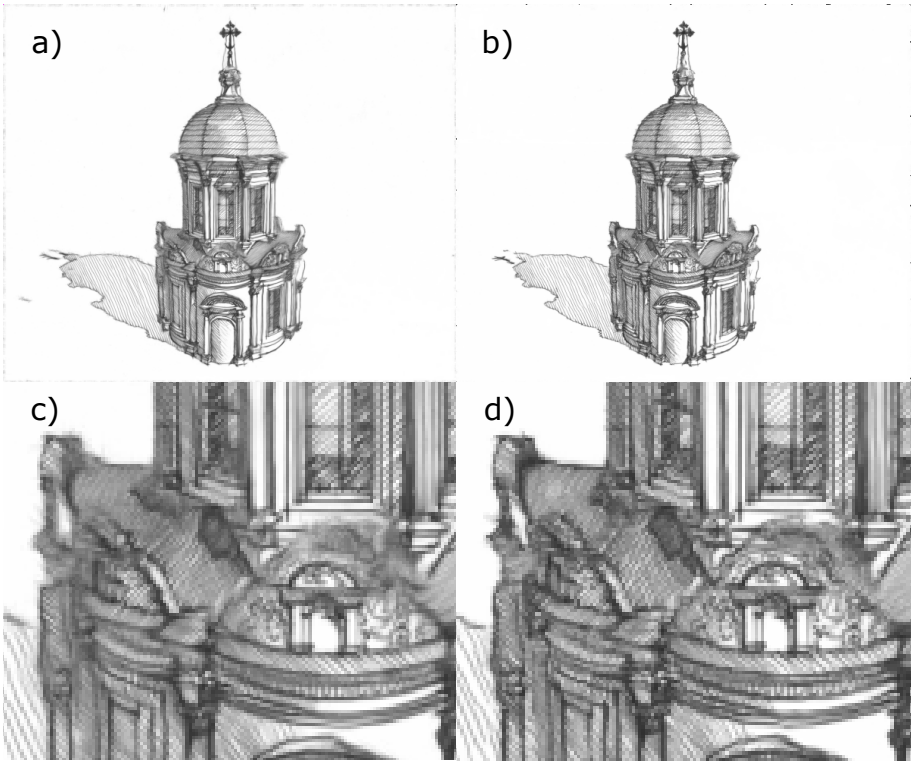

Figure 2.10: Changed voting function
a) result with simple average b) result with average with omitted pixels, c) simple average detail d) average with omitted pixels detail

$av = \frac{1}{|\boldsymbol{V}|} \sum_{v \in \boldsymbol{V}} v$ from all the pixels as before. Then we found a given number t of pixels that are farthest from the average av. These pixels are then excluded and an average without these pixels is computed as a result for given pixel $av = \frac{1}{|\boldsymbol{V} \setminus \boldsymbol{T}|} \sum_{v \in \boldsymbol{V} \setminus \boldsymbol{T}} v$. We usually used t from 3 to 5. If there are just a few (less than t) outliers that are wrongly assigned pixels, this method excludes these outliers and the resulting value will not be so blurred. If on the other hand, all the values are relevant and are close to each other, this method will not change the resulting pixel value much. This is similar to the more robust solution used by Wexler et al. [2004], where they use Mean-Shift by Comaniciu and Meer [2002] in a similar EM-like algorithm. This gives improved and sharper results as seen in Figure 2.10.

2.4.2 Limitations

The style transfer from one view to another cannot work on arbitrary new views. That is because, for each patch, the algorithm needs to find a similar patch in the source view that makes sense in the context. If we look at the model from the side that is opposite to the original view, the model may be completely different and parts that were not visible from the original view may show up (see Figure 2.11). The limits where the algorithm stops working are different for each scene, but generally, it is when places that were not visible in the original view start to show up which are different in a significant way (as lighting conditions, color, texture, etc.). This is something that cannot be overcome with the algorithm and

needs to be taken into consideration when choosing the view to be stylized. An example of a view where the style transfer completely fails can be seen in 2.11 d). The view in this example is from the opposite side of the chapel, where the lighting conditions are different, so the stylization fails.

We found out that usually, the algorithm works well for rotations from $-20°$ to $20°$ on both axes. Outside of this, it starts to break down.

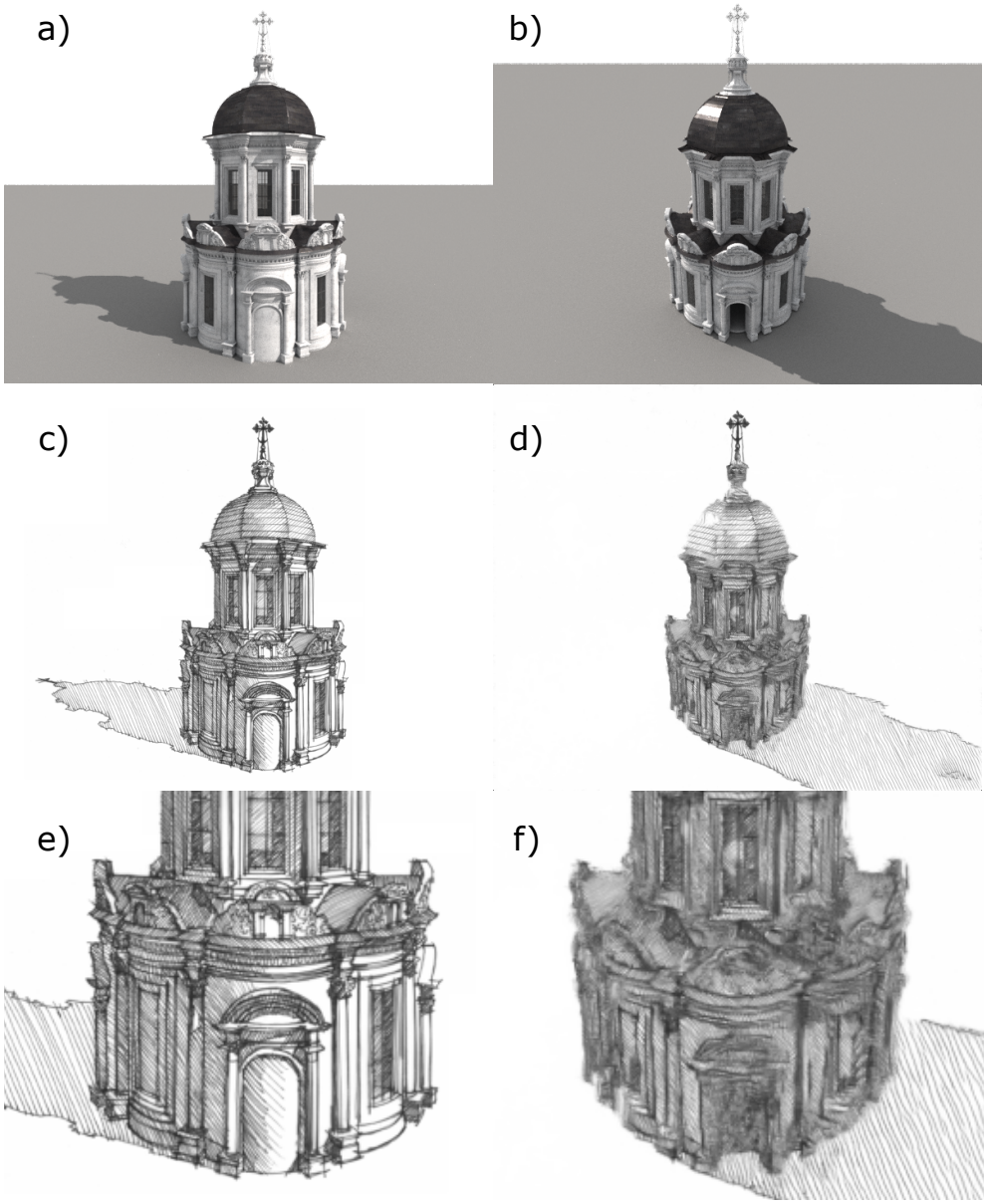

Figure 2.11: Stylization from opposite side
a) original view b) view from opposite side c) original style d) failed stylization on the view from the opposite side e) original style detail f) stylization from the opposite side detail

3. Style transfer on multiple views

In this chapter, we will take the algorithm from the previous chapter and expand it. We want to modify it to transfer the style from the original view to multiple new views in a coherent way. We will make an evenly spaced grid of these new views that will be placed on a sphere around the scene (Figure 3.1). This will allow us to rotate around the scene.

Next, we will look at real-time movement between the stylized viewpoints from the modified algorithm. We will introduce two ways of achieving this. One is simple shifting and interpolating of images, the second one is a more complex shifting of NNFs and a limited stylization. All this needs to run fast enough on a GPU to achieve real-time experience.

3.1 Data acquisition

Generating data for one different view is done simply by any 3D renderer capable of providing the channels we want to use as guides. We used Autodesk 3ds Max with Corona renderer to generate all the guide channels mentioned before.

For a 1D case of the stylized space, the generation of the data is still easy. Since the 1D case can be understood as an animation, most 3D software should be able to generate the data needed. For more dimensions, it starts to get more difficult since we need not only the guides but also motion fields for each pair of neighbor points in the grid.

For each point in the grid, we rendered all the guides, but we also need a motion field for each neighbor of the point. In order to get them, we make another render for each pair of neighbors in the grid. This gets impractical for grids with many points. For grids with thousands of points, the rendering itself takes days even for lower resolutions. Memory starts to be another problem since the motion fields need to be stored in floating-point image format, thus the memory demands get very high.

Because of this, we need to try to make the interpolation work for as sparse grid as possible to make the data generation feasible.

3.2 Enforcing spatial coherence

As a first step, we want to be able to generate a coherent stylized result from a grid of viewpoints (Figure 3.1) for which we have generated all the necessary guide channels. By coherence, we mean that if we move from one point of the grid to another that is close enough, there should not be any abrupt changes in the stylized result. The stylization should move with the model, while still retaining a 2D painting look. The reason for this is that our goal is providing smooth movement between the stylized viewpoints. If these stylized results were incoherent, the sudden changes would disrupt the movement.

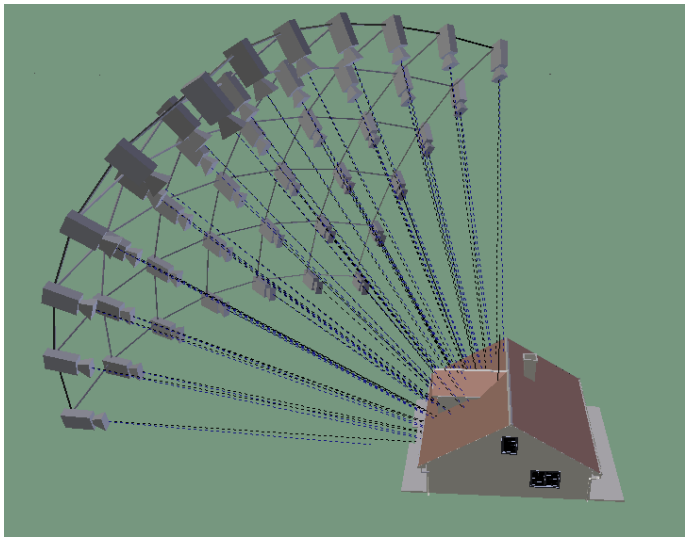

Figure 3.1: Camera grid
Visualization of the grid of views on which the algorithm performs the style transfer.

3.2.1 Frame shifting in animations

The most straightforward way to incorporate coherence in the algorithm is a way used in the stylization of videos such as Jamriška et al. [2019]. In videos, there is a temporal coherence to be enforced. The idea is that frames are sequentially stylized and a result of one frame is shifted to the position of the next frame. In our case, since we have motion fields from the underlying geometry, that means just shifting all the pixels of the stylized image according to the motion field.

The shifted images can be used as coherence guides, so a new term is added to the patch error from Equation 2.3, so it becomes

$$\begin{aligned} E_c(A', B'_i, G^S, G^T, p, q) =& ||A'(p) - B'_i(p)||^2 \\ &+ \sum_{g \in \mathbb{G}} \lambda_g ||G^S_g(p) - G^T_g(q)||^2 \\ &+ \sum_{c \in \mathbb{G}_c} \lambda_c ||A'(p) - G^T_c(q)||^2, \end{aligned} \tag{3.1}$$

where $\mathbb{G}_c$ is a set of shifted images (in the case of animation just one element set). By taking the difference between the shifted image and the style image at p, coherence is promoted.

With more dimensions (as in the Figure 3.1, we can just expand the set of coherence guides $\mathbb{G}$, but a new question arises as to in what order should the viewpoints be stylized. The same issue arises for the approach used by Bénard et al. [2013], where at each level a forward and backward sweep is done.

The minimized energy from equation 2.4 is defined just for one frame, so in the case of stylization of multiple viewpoints at once the minimized energy becomes

$$E_M = \sum_{T \in \mathbb{F}} E(A', B'_i, G^S, G^T) = \sum_{T \in \mathbb{F}} \sum_{q \in \mathbb{Q}_\mathbb{T}} E_c(A', B'_i, G^S, G^T, p_q, q) \tag{3.2}$$

where $\mathbb{F}$ is the set of all the view points. Our goal in the stylization on the grid is minimizing the sum of the errors of the individual frames. That means that a

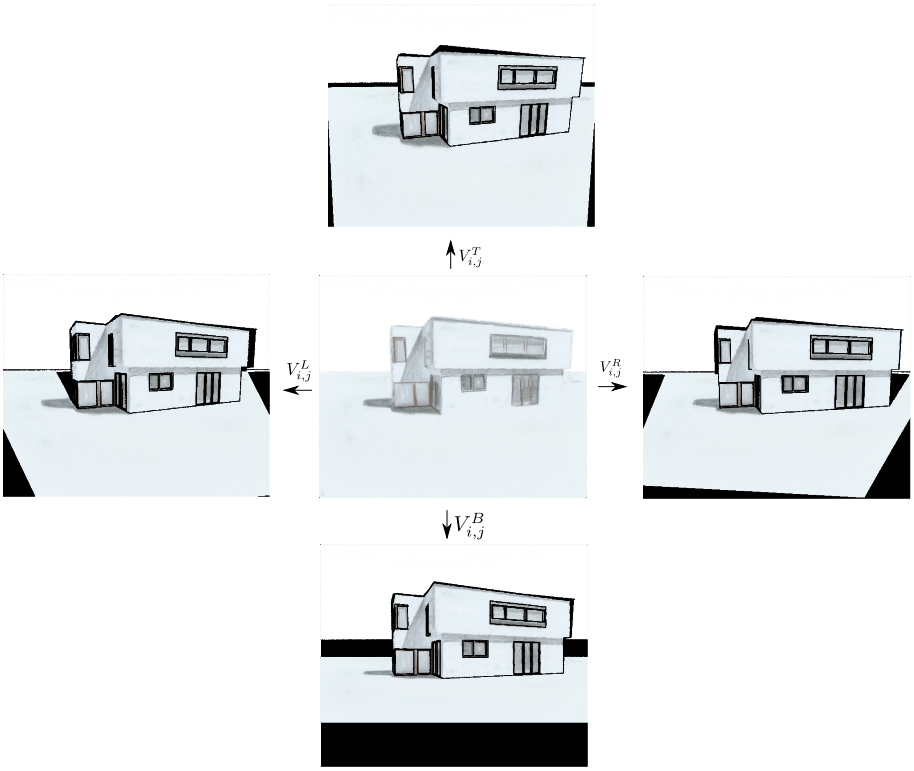

Figure 3.2: Shifting scheme
One frame shifted to four neighboring viewpoints on the grid. Black pixels are masked pixels.

stylization of individual frames independently is not going to work in this case, because the errors of the individual frames are dependent on the results of the neighbor frames.

3.2.2 Intermediate shifting

Because of the problems with ordering described in the previous section, we propose a different way of propagating coherence. Instead of computing the frames sequentially in a given order, we will stylize all the frames at the same time in parallel.

To do that, we modify the StyLit algorithm a little bit. After each search-vote iteration, we get an intermediate result. We run one search-vote iteration for every frame, then shift these intermediate results to all neighboring viewpoints on the grid (see Figure 3.2) and use them as guides. With this approach, we are minimizing the error E_M from Equation 3.2.

This approach also brings computational advantage as in each search-vote iteration computation of the frames is independent and thus it can be parallelized easily.

Results can be seen in Figure 3.3. In the figure, in a.1) - a.3) there are results without coherence and in b.1) - b.3) there are results with coherence. Some more significant changes between the neighboring frames of the result without coherence are visible in red rectangles. Even though overall the changes do not seem very prominent on still images, these changes are distracting in a smooth

movement.

3.3 Real-time interpolation

The algorithm described in the previous chapter has a series of images as output, which are stylized views on a grid around the scene. With these data, we want to achieve a continuous experience, in which we could move around the model in the space defined by the grid. Computing the stylization as described before in a grid that would be fine enough to provide this experience is not feasible. Because of that we need to implement an algorithm that would provide stylized results for viewpoints that are not on the grid. The algorithm needs to provide a smooth transition from one point of the grid to another while preserving the stylized result.

3.3.1 Interpolation of images

The first option is simply taking the resulting stylized images and then interpolating between them.

Blending Simple linear blending does not match our requirements since the grid is quite coarse in most examples, so linear blending causes visible blurring and makes for an unpleasant experience.

Shifting With the underlying geometry, we can shift the results from the views of the grid to the current position. We shift the close grid points $g_{i,j}$, $g_{i+1,j}$, $g_{i,j+1}$ and $g_{i+1,j+1}$, which are around the current position, to the current view v as shown in Figure 3.4. We also consider masking of the shifted pixels – shifted pixel needs to have a similar color in original mask at the position from which it was shifted and in current mask at the position to which it was shifted. These masks are produced as color renders without any light applied from the viewpoints of grid points and from current viewpoint. Pixels that are not masked in the shifted image from the closest grid point (on the Figure $g_{(i+1,j)}$ is the closest) are shifted from this closest grid point. Masked pixels are shifted from the other grid points. If all pixels are masked, the closest grid point is used.

If masking was not taken into account, a lot of pixels would be shifted according to different objects than the ones they belong to. But even when only unmasked pixels are shifted, some artifacts are still visible – since the style exemplar is never drawn exactly over the geometry (parts of outlines are drawn over geometry that corresponds to background). These artifacts can be seen in Figure 3.5 a).

We can also blend all these four shifted images, which alleviates the problem with the artifacts and makes them less noticeable, but at the same time blurs the results.

Conclusion

The problem with this approach of image interpolation is that the grid of points to interpolate needs to be quite fine to produce acceptable results. When the grid

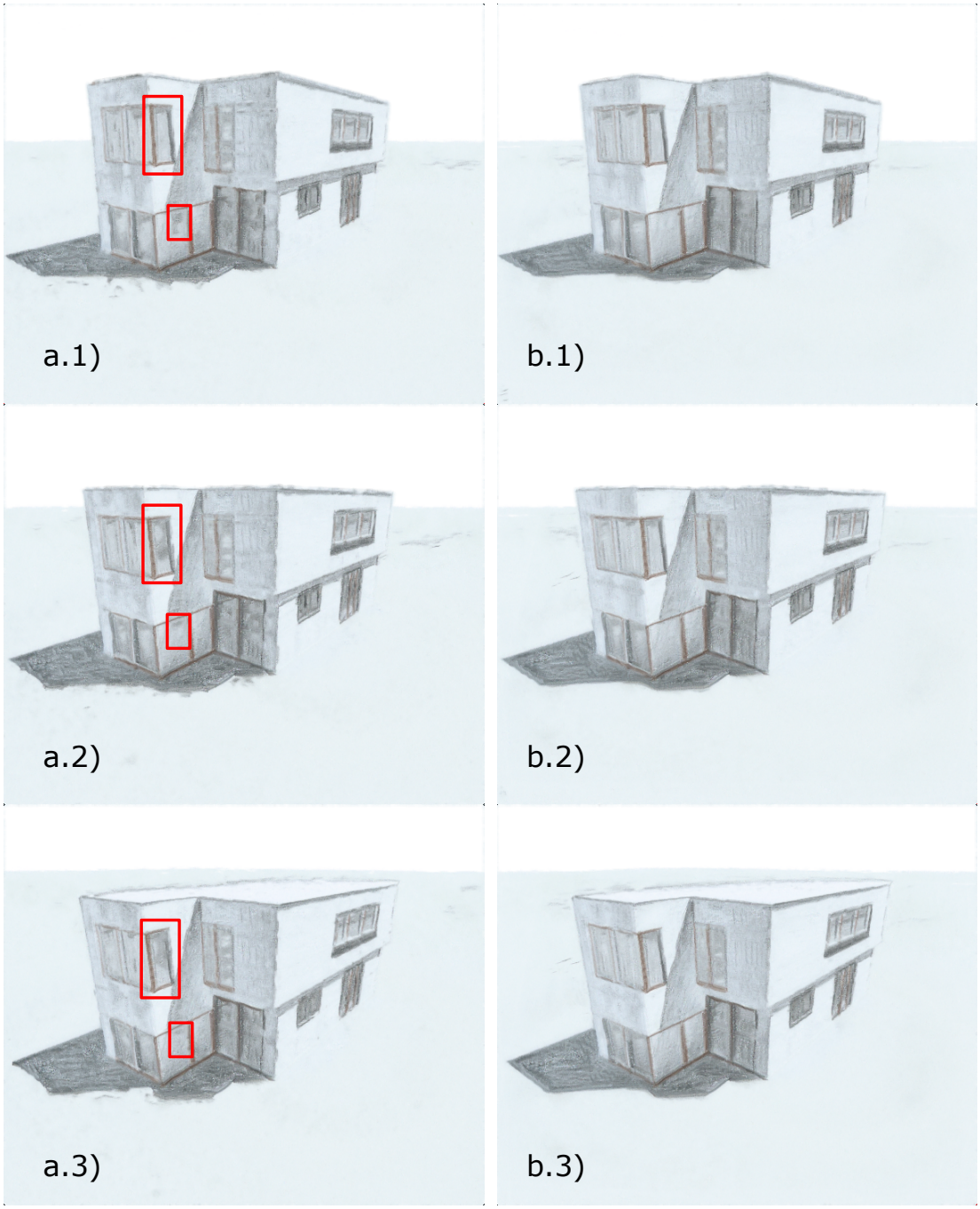

Figure 3.3: Coherence comparison
a.1) - a.3) are three results generated without coherence (changes visible in red rectangles) b.1) - b.3) are three results generated with coherence.

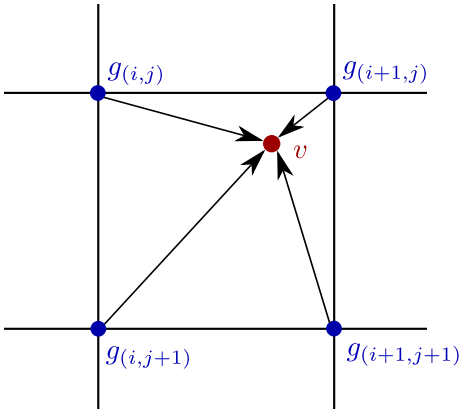

Figure 3.4: Real-time shifting
Schema of shfiting of points on grid ($g_{(i,j)}$,$g_{(i+1,j)}$, $g_{(i,j+1)}$, $g_{(i+1,j+1)}$) to the current view v.

becomes sparse, the interpolation overall starts to look more like a 3D model with a sketch texture. Artifacts also become more prominent. While this approach provides good results in a fine grid, because of the problems mentioned in Section 3.1 with grids consisting of many points, the approach is not suitable. It is also inefficient – we need to compute a fine grid, which means computing a lot of stylizations in points that are close by and thus similar.

3.3.2 Interpolation of NNF

A different approach can be taken to the interpolation phase. Instead of interpolating the resulting stylized images, we can take the results of the stylization in the form of a nearest neighbor field and shift that. The nearest neighbor field holds the coordinates of the closest patch from the source image for each patch in the target image.

Shifting

The shifting itself is the same as when shifting just the images. Instead of pixel colors, we shift coordinates from the nearest neighbor field (NNF). For each pixel that is a patch center in the current image, we look where this pixel was in the view from a grid point with the use of a motion field. Then, we copy the coordinates from this position from the grid point to the current pixel. Again, we use masking to prevent shifting neighborhoods between different areas.

The use of NNF when shifting allows us to use a sparse grid. That is because when shifting NNF, one shifted patch can repair errors in its proximity, thus making this approach more robust against error. And because we have access to the NNF, we can modify it and improve the results.

In every point, we shift the four grid points in between which the current position is (as can be seen in Figure 3.4). This gives us more data to work with.

Producing the NNF

First, we need to construct a NNF for the current view. We can use shifted NNFs from the grid points between which we are (in 2D we have 4 of those shifted

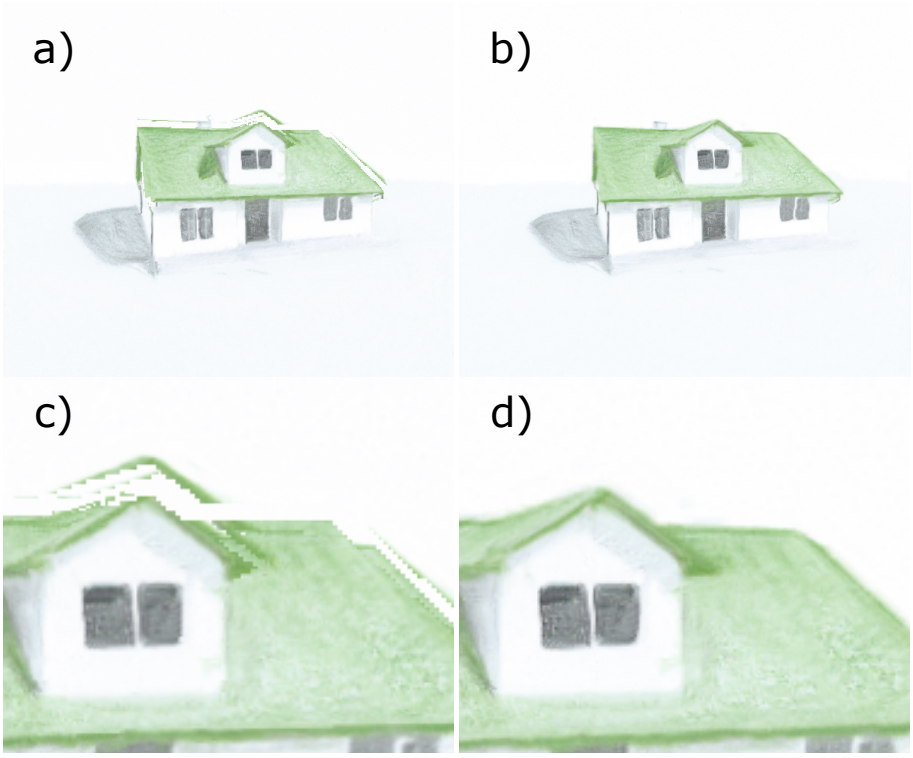

Figure 3.5: Interpolation modes
a) Image shifting b) NNF shifting c) Image shifting detail d) NNF shifting detail

NNFs). From these NNFs, we need to produce a new NNF that would represent the current view the best.

We take the coordinates from the closest grid point on which the current position is unmasked. So the new NNF is made as

$$\mathrm{NNF}_c(p) = \mathrm{NNF}_g(p - V^g(p)), \tag{3.3}$$

where g is the closest grid point in which the shifted pixel is not masked (in the case when all the grid points are masked, the closest one is used) and V^g is the motion field from g to the current position.

After producing the NNF, we run voting that is done by the simple average as in the original StyLit (see Chapter 2), which is fast enough to run in real-time. This voting produces current style image candidate (see Figure 3.6 a)).

Limited search iteration

After the voting, we get a stylized image that is already of good quality. But this can still be improved. Since we already have several shifted NNFs from the previous step, we can use them further. The patches in the shifted NNFs were good matches in their respective viewpoints so we can expect that their correspondence according to the guide channels is still good even after the shift. That holds if the distance of the viewpoints is not too excessive since the guide does not change much with movement. The only thing that changed is the style. Now, the current style from the previous iteration is influenced by the masking and by the choice of NNFs correspondences, thus making it blurry at certain places. Also, artifacts on places where the style exemplar was not exactly over

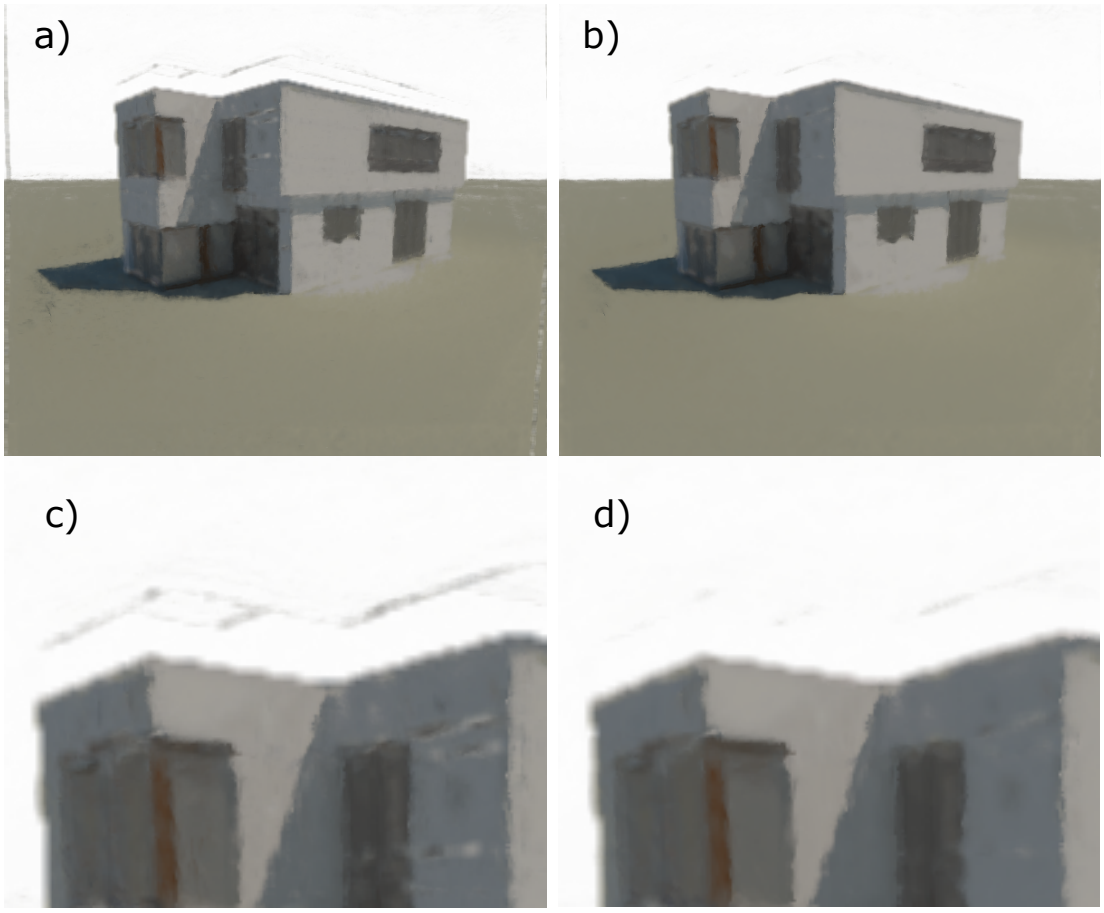

Figure 3.6: Limited iteration comparison
a) result without limited iteration b) result with limited iteration c) detail of result without limited iteration d) detail of result with limited iteration

the original image (as explained in image shifting) are visible (see Figure 3.6 a), c)). To address this problem, we run one refining iteration to improve the results.

This iteration is done as follows. For each patch in the target image, we look at all the correspondences from all the non-masked shifted NNFs. We compute error, but this time only on the style image, since we do not even have the guide channels and since at this stage we want to focus on improving the final style appearance. We then choose the best patch from those shifted NNFs. This does not involve much computation and can be easily run in real-time on GPU.

After this, we get a new, changed NNF, so another voting iteration must take place to get the final stylized image. The improvement caused by this limited iteration can be seen in Figure 3.6.

3.3.3 Conclusion

This approach still suffers from some problems. One of the prominent problems with this approach is the lack of interpolation. Because of this, there is a sudden change when the closest grid point changes, but thanks to the coherent generation from Section 3.2, the change is not too big. The importance of the coherent generation of the underlying data can be seen in Figure 3.7. There are also some artifacts that are caused from the shifting that remain unmasked, similarly

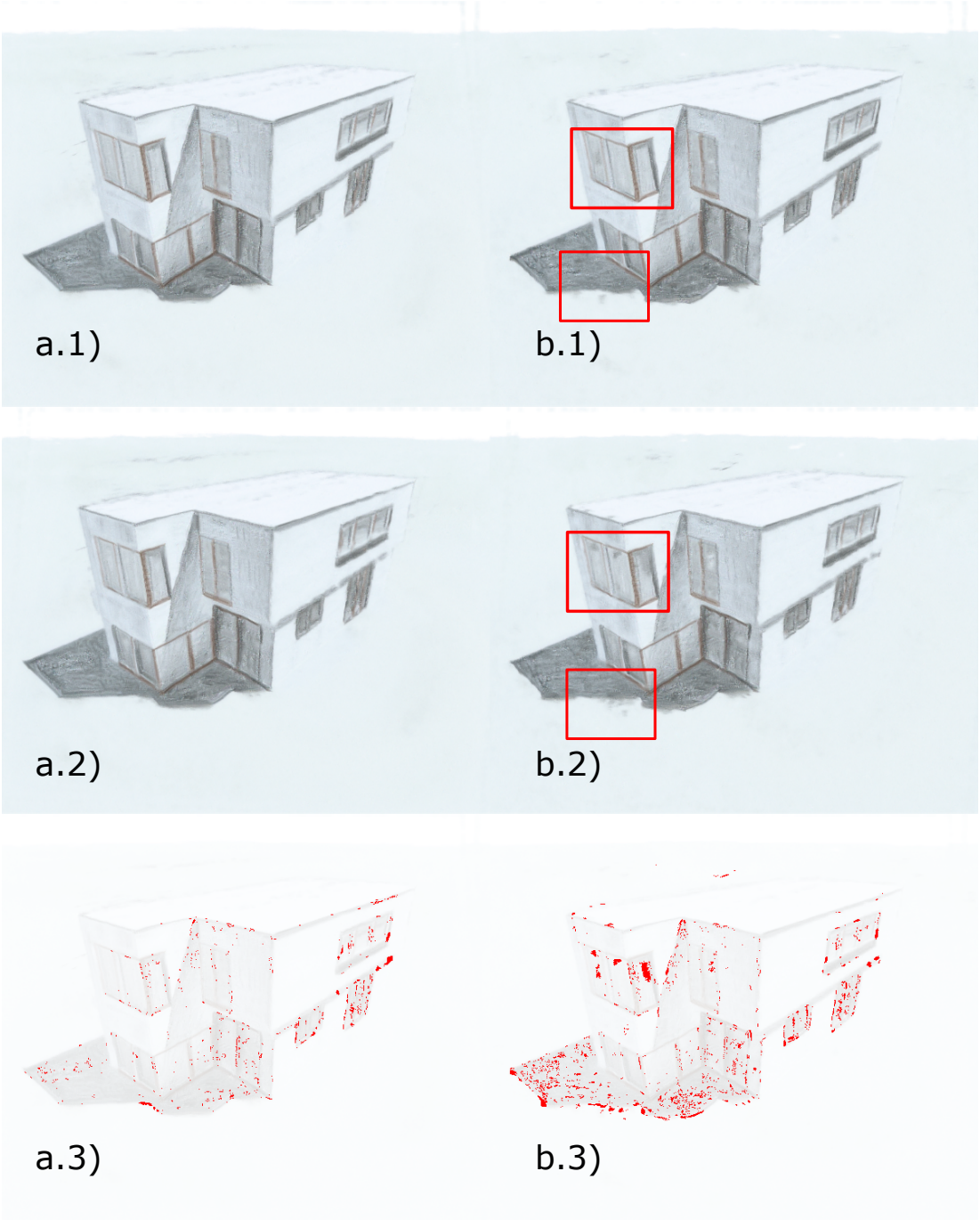

Figure 3.7: Coherence importance in interpolation
Images 1 and 2 are from almost the same view, but with changed closest point on the grid. In a) are results with coherence while b) without coherence. In 3) red pixels are pixels with a big difference between 1 and 2.

to the problem with image shifting. Also the outlines get partly masked and thus disappear on some places. Because of the limited stylization iteration, this problem is also less prevalent than in the image shifting. The quality of this interpolated result is just slightly worse than the result stylized with the StyLit algorithm from Chapter 2, see Figure 3.8.

Overall this approach outperforms the image shifting in almost all regards (comparison of these two approaches can be seen in Figure 3.5). The only downside is that it is slower to compute (for details, see the section about implementation, 4.2). But otherwise, it requires less precomputed data points, the results are of better quality and it produces fewer artifacts.

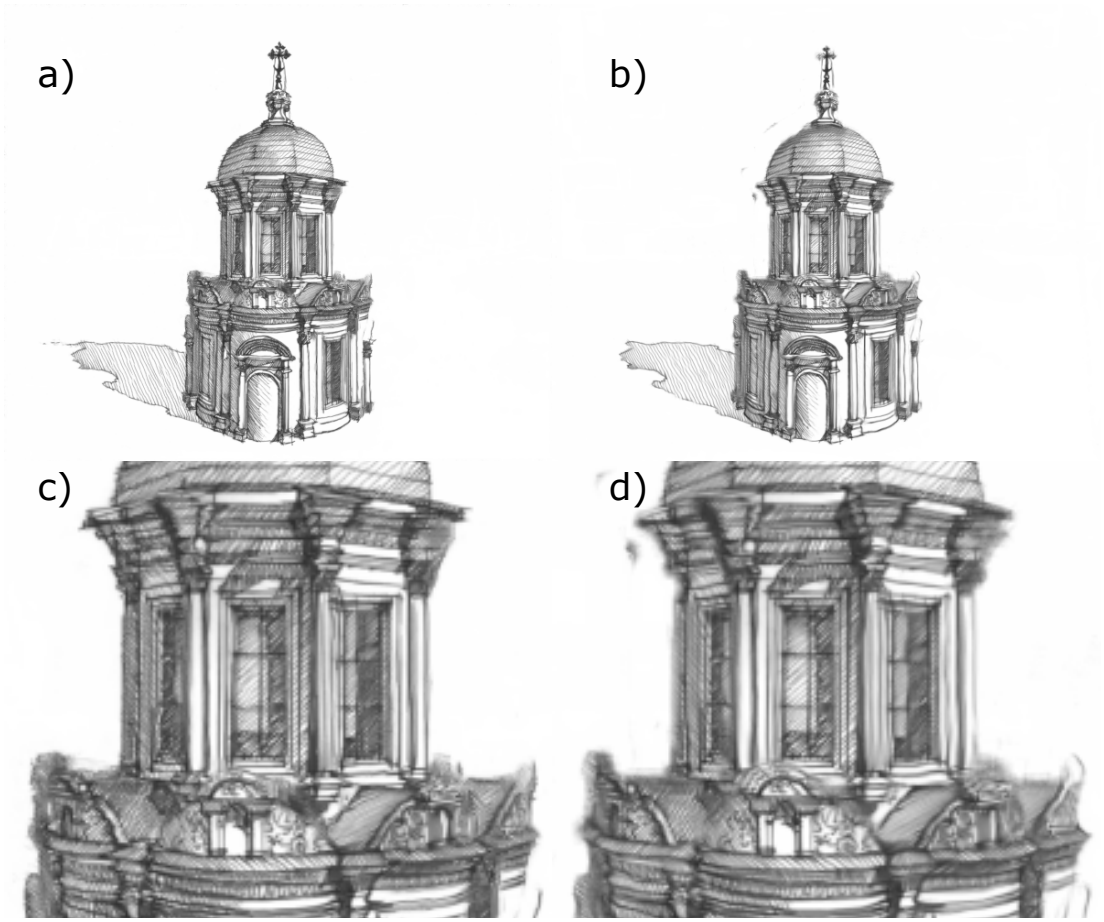

Figure 3.8: Comparison - interpolation and stylization
Comparison of results from the same view from the full stylization algorithm in a) and interpolated result in b).

4. Implementation

As part of this thesis, we implemented two independent applications. One is the style transfer algorithm based on StyLit (Fišer et al. [2016]) described in Chapter 2 with the expansion to spatially coherent style transfer as described in Section 3.2. The second application is the real-time viewer with limited stylization from Section 3.3. In this chapter, we will provide details on the implementation of those two applications.

4.1 Stylization algorithm

In this section, we will have a look at the implementation details of the stylization part of the thesis. This part was written in C++ and it has no user interface. The implementation is part of Electronic attachment A.1. It must be run with a configuration file that sets all the parameters of the style transfer.

4.1.1 Used libraries and technologies

In this part, we will introduce the libraries and technologies used in our implementation of the style transfer algorithm.

JetBrains CLion

An integrated development environment we used for the development of the C++ implementation.

CMake

A management tool for the build process offering multiplatform support.

CImg

A lightweight header-only library we used for basic image manipulation, including loading and saving images, rescaling and rotating.

OpenMP

A library used for parallelization. We use it for significant speedup, especially in the spatially coherent stylization, where all the frames can be run independently in parallel.

LibPNG

A library for manipulation with png files. Our implementation expects the guide channels and the style exemplar as a png file and saves the results in png files.

OpenEXR

A library for loading and saving exr files. This is a format for floating-point images. We need manipulating with floating-point format for motion vector fields for shifting described in Section 3.3.2.

4.1.2 Implementation details

We implemented the StyLit algorithm (Fišer et al. [2016]) with one slight change from the algorithm described in the paper and in Chapter 2. In the search phase (Section 2.4), we do not fit a hyperbolic function to get the error budget, instead, we compute the error budget as a percentile of the sum of all the errors. We did this for convenience and more control over the error budget. The percentile parameter can be chosen in a configuration file and we usually used 0.9. We also implemented all the changes proposed in Section 2.4.1 and the means to compute the style transfer in a spatially coherent way as described in Section 3.2.

Search phase

In the search phase of the algorithm (Section 2.3.1), we need to find the closest patch from B for each patch from A. This search happens several times during each search-vote iteration, which itself happens multiple times during the synthesis. Because of this, we need an algorithm that can establish the correspondences quickly, so a brute force search is too slow to use.

We decided to implement the PatchMatch algorithm by Barnes et al. [2009], which is the algorithm that was used in the original StyLit paper. This algorithm is used for approximate patch correspondence retrieval and is fast enough for our purposes. Even though the algorithm is only approximate, the results it provides are very good and usually close to the truth.

Input and output

The application takes guides of all frames as specified in Section 2.4.1. The algorithm expects all the guide channels to be in a png file format and the motion fields to be in and exr file format. More details can be found in Electronic attachment A.1.

The output is a series of png files. For each frame, the application saves the stylized result. It also saves the resulting NNF in a binary file, if NNF saving is enabled in the configuration. The application also needs to save the progress if we are performing the spatially coherent style transfer, so after each search-vote iteration, an intermediate result for each image is saved. These intermediate results can also be used to continue the stylization in case it needed to be stopped.

Performance

The style transfer using this algorithm is quite slow, which is caused mainly by the search phase. In this phase, the patch match algorithm (Barnes et al. [2009]) makes a lot of comparisons of patches from source and target images. This comparison consists of a lot of operations since it is computing error across all the guide channels and across all the pixels in the neighborhood.

We ran the algorithm on a six-core CPU Intel i7-8750H. For a single frame stylization without parallelization, the algorithm stylized a 600x500 image with six search vote iterations and six patch match iterations in 312 seconds on average, and when multiple images were run and stylization was parallelized on all the 6 cores, average time per frame was 61 seconds.

4.2 Real-time viewer

In this section, we will look at the implementation of the real-time viewer described in Chapter 3.3. We implemented both the image shifting and NNF shifting approaches. This part was written using C♯ and Cg. It is independent of the stylization application and if supplied data in a compatible form, it could run with data from any stylization algorithm.

4.2.1 Used libraries and technologies

In this part, we will introduce the technologies used for our implementation of the real-time viewer.

Unity

Unity is a 3D game engine. It is well known and has a comprehensive documentation[1]. Even though it offers a lot more functionality than we needed, we chose to use it for several reasons. The first is for its popularity. Because of its popularity, a lot of community support is present, even for things that users do not need often (in our case the generation and use of motion fields for example). Another reason is the simplicity of use. We needed to manipulate 3D geometry and camera settings and Unity offers everything we needed out of the box and well documented.

4.2.2 Implementation details

Because of the specific nature of our application, we did not need all of the things Unity can provide. We did not use any of the physics systems and we also disabled shadows and set the quality of light rendering to the lowest settings.

Input data

The application can load Unity AssetBundles[2]. In the case of a bundle for image shifting, it has to contain images with the results of the stylization. In the case of NNF shifting, it needs to contain the binary files with coordinates of the matched patches as floats.

Currently, the viewer is limited to one camera configuration. That is because transferring a camera from a 3D rendering software to Unity is not a straightforward process. The cameras cannot be just transferred one to one easily, so we resorted to the solution of fixed camera configuration. That means that to use

[1] https://docs.unity3d.com/Manual/index.html (Accessed December 21, 2019)

[2] https://docs.unity3d.com/Manual/AssetBundlesIntro.html (Accessed December 21, 2019)

the viewer, all the data supplied need to be generated using a camera with the same settings as we used and in a fixed resolution of 600x500 px.

Motion fields

Unity can generate motion vector fields automatically from cameras. These motion vector fields capture pixel position change from the last frame for that current camera. Because we need a motion field for four grid points, we must have a camera for each of them that moved from the grid point to the current position in the current frame. Because of this, we need eight cameras in total that are alternating between the grid points and current position. We also found out that rendering the motion field in lower resolution provides acceptable results.

Masking

For masking of the shifted NNFs, we use the same cameras as for the motion fields. Each of the cameras renders a motion field in the first two components of the four-component result. We then store the masking information to the remaining two components. The masking information comes from a simple color of a mesh that has no lightning applied.

Shader implementation

We implemented the whole algorithm in shaders. The shaders are supplied the grid point NNFs or images that are around the current position, and the current position. The shaders compute all the interpolation.

The NNFs are transferred to the shader as compute buffers that contain float coordinates of the nearest neighbors.

For the implementation of the NNF voting and limited stylization step (Section 3.3.2), we used a shader with multiple passes. The shader uses four passes in total, where in the first pass the first NNF is produced from the shifted NNFs and in the second pass a voting from the produced NNF is performed. In the third pass, the limited stylization takes place and in the fourth pass, another voting is performed.

4.2.3 Performance

We want to be able to run the viewer in real-time. Our goal is also to allow the viewer to be run even on lower-end devices. We tried to run the viewer on a low-end specification without dedicated GPU[3], on which the NNF shifting ran with about 30 FPS and the image shifting 60 FPS. The image shifting is much faster as expected since it does not perform additional limited stylization step as the NNF shifting does (Section 3.3.2). If the system we would like to run the viewer on had even lower specification than the lowest configuration, we could disable the limited stylization step in NNF shifting.

[3]intel i3-7100U + no dedicated GPU

5. Results

In this chapter, we will present and discuss the results of the implemented algorithms. The algorithm is hard to evaluate, because the quality of stylization is a highly subjective matter and there is no universal metric to measure the quality. However, faults such as blurring and artifacts are obvious.

Another problem with showing the results here is that the algorithm is implemented to provide smooth interpolation, which cannot be conveyed by still pictures. Because of that we also provide video of the results which can be found on the webpage `http://hauptfleisch.cz/ArchStyle/video.html` (the video is also included in Electronic attachment A.2.3). And we also provide the real-time viewer with all the bundles in Electronic attachment A.2.

The models we used to generate the results in this chapter are shown in Figure 5.1. We used three vastly different models that should represent different types of architectural models. The first one is a model of a chapel[1] with a lot of details and with a lot of complex geometry. This model is mostly rounded without many flat surfaces. The second model is a modern building[2] with a minimum of details and a lot of flat surfaces. The third one is a model of a more traditional house[3] that has some details and a lot of flat surfaces at the same time.

We experimented with multiple style exemplars for each of these models.

5.1 Chapel

The chapel model is a highly detailed model with a lot of curved surfaces. The only bigger flat surface that is interesting for the style transfer is the shadow on the ground. For this model, we have three style exemplars from artists and one computer-generated style (generated using FotoSketcher[4]). We used the

[1] https://www.turbosquid.com/3d-models/3d-orthodox-chapel-model/998158 (Accessed December 26, 2019)

[2] https://www.turbosquid.com/FullPreview/Index.cfm/ID/1204626 (Accessed December 26, 2016)

[3] https://www.turbosquid.com/FullPreview/Index.cfm/ID/795483 (Accessed December 26, 2019)

[4] https://fotosketcher.com (Accessed January 2, 2019)

Figure 5.1: The models used for producing the results in this chapter
a) Orthodox chapel model from Turbosquid by Brut1ch b) modern house model from Turbosquid by MartinDiavolo c) traditional house model from Turbosquid by 3D COR

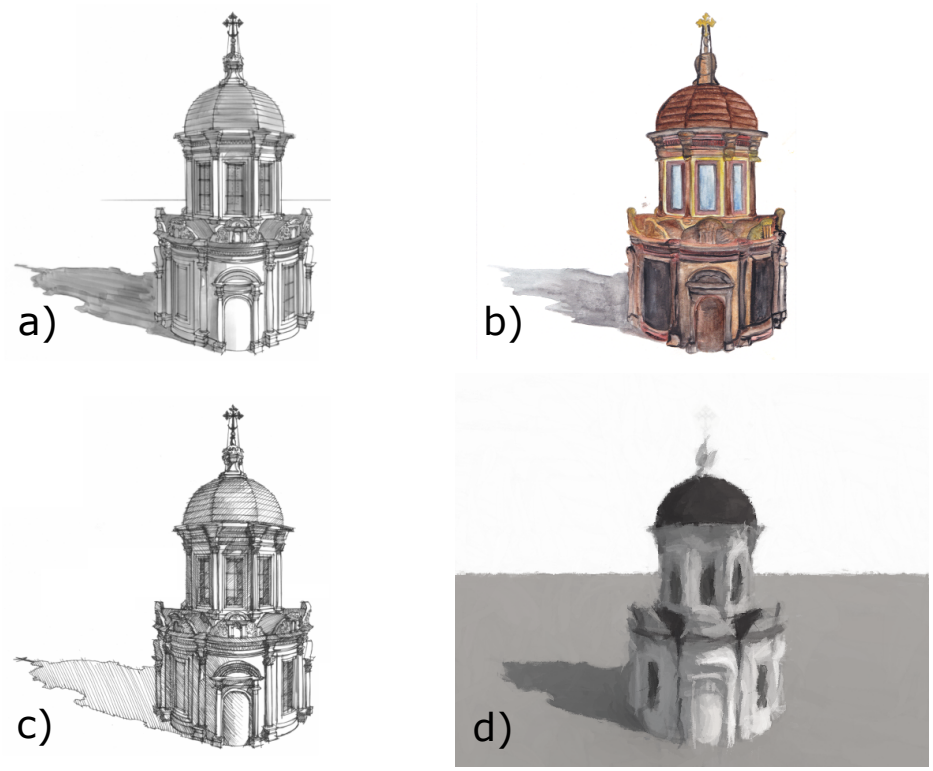

Figure 5.2: Styles for the chapel mode
a) Shade style by Jan Pokorný, b) Watercolor style by Štěpánka Sýkorová, c) Hatch style by Jan Pokorný, d) Paint style generated by FotoSketcher

computer-generated style in order to have one style that is the same for all the models. The styles used for the chapel model are in Figure 5.2.

This model also posed some problems when used in the real-time viewer. Since the highly detailed geometry has over two million vertices, it is not suitable for real-time use. To run the real-time viewer on this geometry, we had to simplify it first.

Two parts of the scene cause problems to the style transfer algorithm. These parts are the cross on top of the chapel and the cross in the shadow of the chapel. That is caused by the cross being visually such a prominent feature that the artists tend to exaggerate it (see a), b) and c) in Figure 5.2). This causes that it is very detailed in the artistic drawings, but without a lot of underlying geometry. This causes problems when shifting the results of one frame to another (from Section 3.2). That is because the geometry there is so thin that not much gets shifted. Because of this the coherence term then enforces error. Because of this, after the style transfer, the cross sometimes disappears partly or completely, or its overall shape is changed. See Figure 5.3 for these cross deformations.

A similar issue comes with the shadow of the cross. It is such a thin line that it is very hard for the algorithm to transfer it correctly in the guides. It is also not different from the rest of the shadow. This problem with the shadow of the cross can be seen mostly in c) from Figure 5.2, since the author put a lot of emphasis in the cross shadow.

Stylization results

In this part, we will discuss the results of the style transfer algorithm described in Chapter 2 with the space coherence expansion from Section 3.2.

All the results were generated on data taken from a uniform grid from a sphere around the model. Two neighboring points on the grid are 5° apart.

Results of the stylization on the chapel model can be seen in Figure 5.4 from multiple views without details and in Figures 5.5, 5.6, 5.7 and 5.8 in b) and d) for more detailed results.

In these results, there are a few interesting things to note. The view a) is the same view from which the artists drew their style exemplars. That means that this view should be fairly easy for the algorithm to replicate. This also means that here we have one view for which we have a ground truth to which we can compare the result. Most problems in this view can be seen in the hatch style, that is, in the image 2.a). The algorithm could not transfer the style on the shadow very well. The shadow of the cross is completely missing for the reasons mentioned before. But the algorithm is also struggling to keep the hatching lines straight in the shadow. That is because the guides in the area of the shadow are almost uniform. Thus, the algorithm struggles to decide which patch should go exactly where. This problem is less prominent in areas where the style is less sharp (the shadow of the chapel in the other styles) because in these parts small errors are not visible.

The viewpoint b) is a viewpoint from $(10°, 10°)$. In this view, there are already some areas that were not visible from the original point of view. These areas are mostly parts of the roof from above and some smaller parts of the wall on the left side of the chapel. Some areas get a little blurry, but the overall results are still satisfactory.

The view c) view is from $(-20°, -20°)$ where the style transfer is already starting to fail. The only result that does not exhibit severe problems is 4.c), although the structure of the strokes is lost as well. In 1.c) and 2.c), there is a lot of discontinuous lines or lines that do not retain their straightness. In 3.c),

Figure 5.3: Problems of the algorithm with the cross

the colors are getting mixed up.

Interpolation results

In this part, we will look at some results of the interpolation algorithm. Since still images cannot convey the interpolation results very well, we offer a video of the results on-line at `http://hauptfleisch.cz/ArchStyle/video.html` and in Electronic attachment A.2.3. We attach the viewer application in Electronic attachment A.2 together with all the data that we presented in this chapter.

Here, we will show several interpolation results to illustrate its quality and to show its problems. The results are in Figures 5.5, 5.6, 5.7 and 5.8 for the different styles. There is always a stylized result in a) (with detail in c)) and a shifted result in b) (with detail in d)).

In these figures, it can be seen that the the interpolated results are generally of slightly lower quality than the results from the stylization. But the difference is not very big. Some problems that can be seen are the artifacts from shifting as mentioned in Section 3.3.2. These artifacts can be seen in all the styles as splotches on the white background (see for example Figure 5.6 d) on the right side).

The most visible problem is that the edges of the drawings get washed out. That is because of the masked shifting. It is connected to the problem with artifacts, because the artist usually does not draw only in the places where the object is, but the outlines are partly outside of the object. These outlines are then masked and get washed out (this can be seen for example in Figure 5.5 d) on the outline in the center of the image, but it is visible on every style).

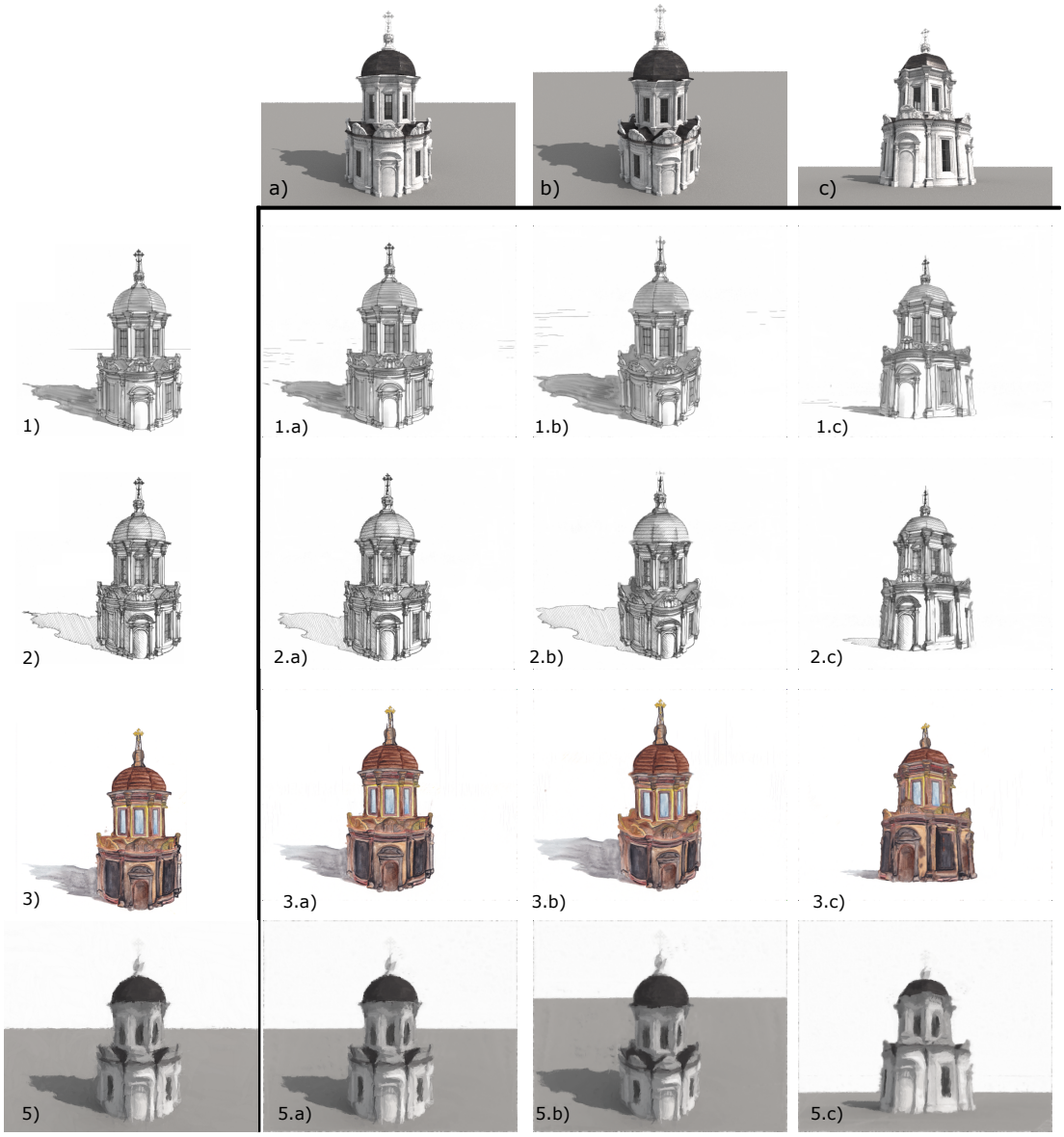

Figure 5.4: Results of the stylization algorithm from multiple views
Results on the modern house model on multiple views: a) the original view b) view from $(10°, 10°)$ c) view from $(-20°, -20°)$.

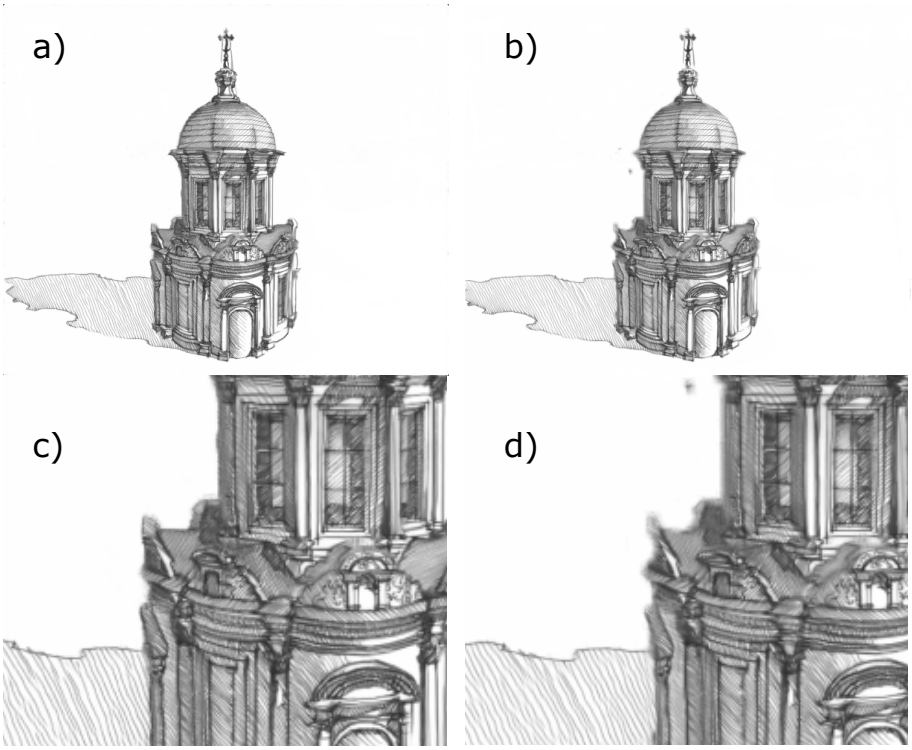

Figure 5.5: Results on the chapel model – hatch style
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

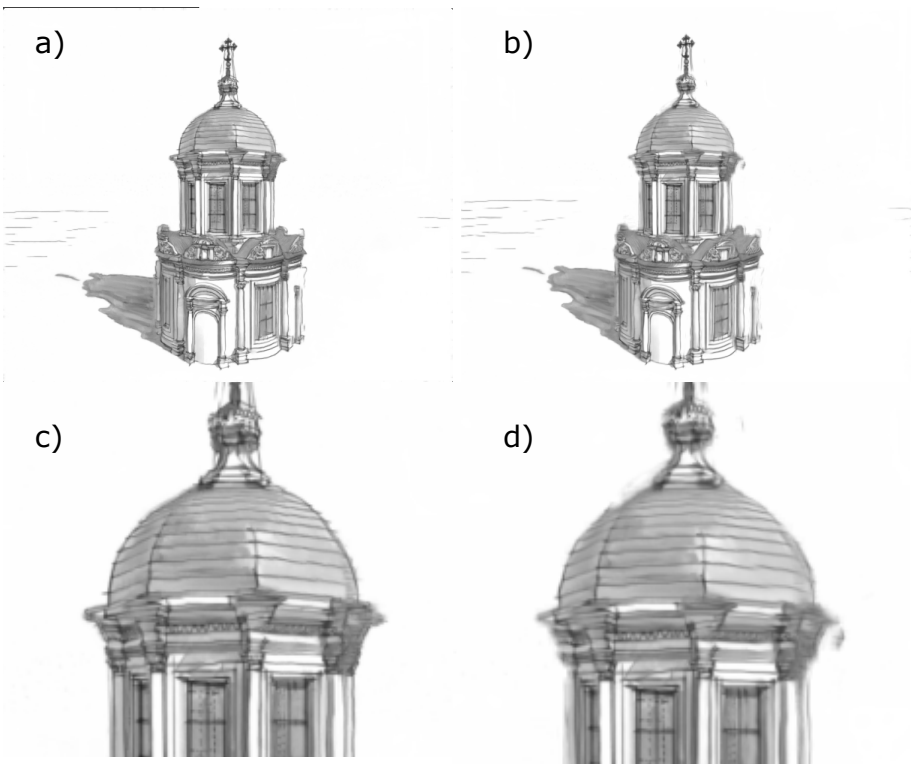

Figure 5.6: Results on the chapel model – shade style
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

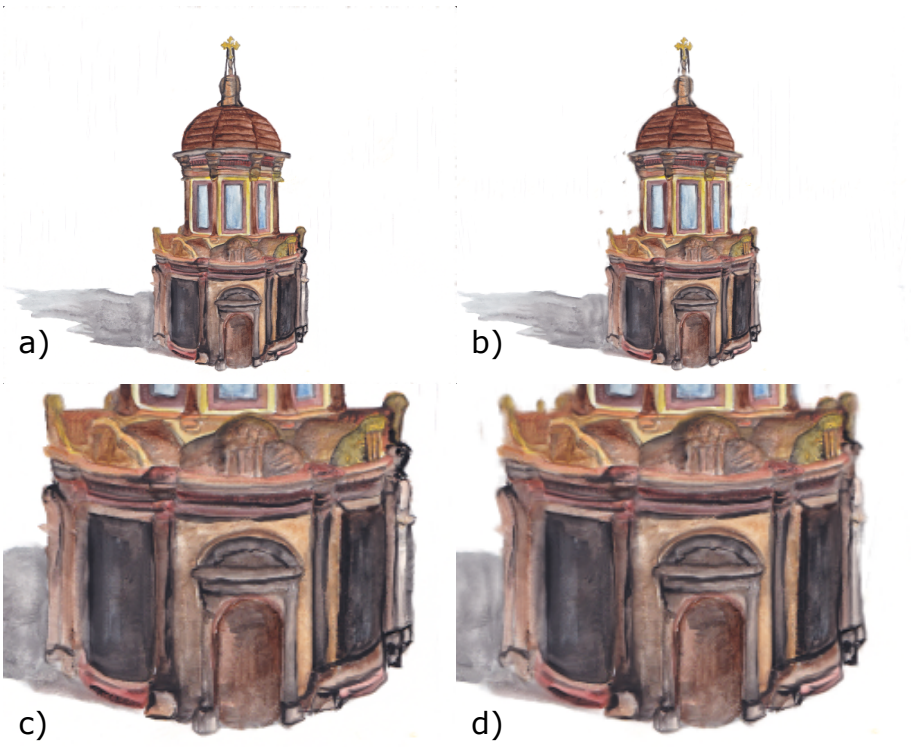

Figure 5.7: Results on the chapel model – watercolor style
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

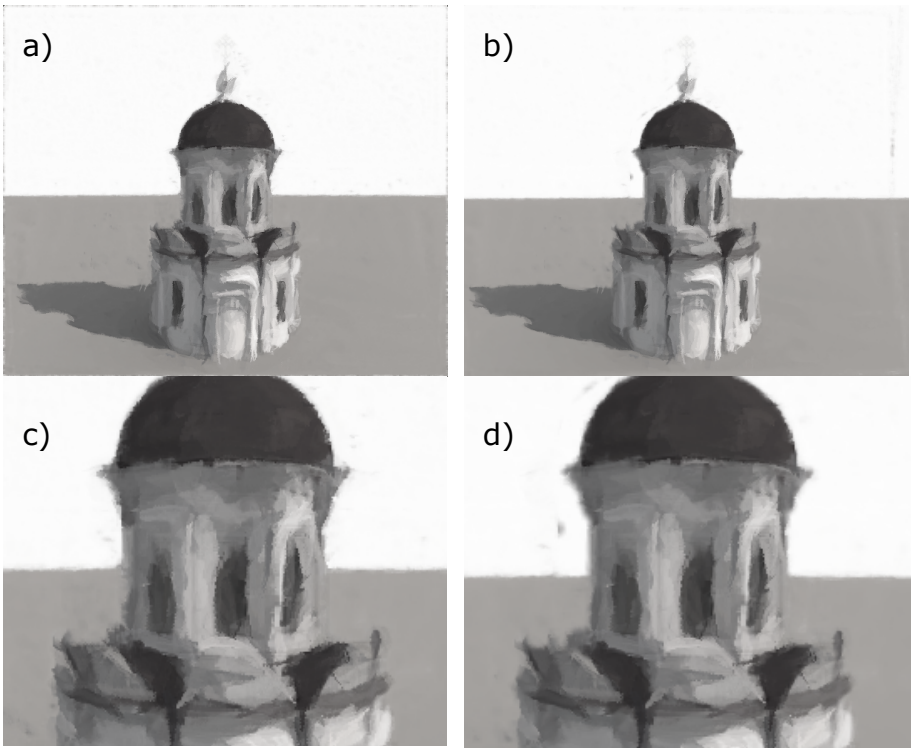

Figure 5.8: Results on the chapel model – paint style
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

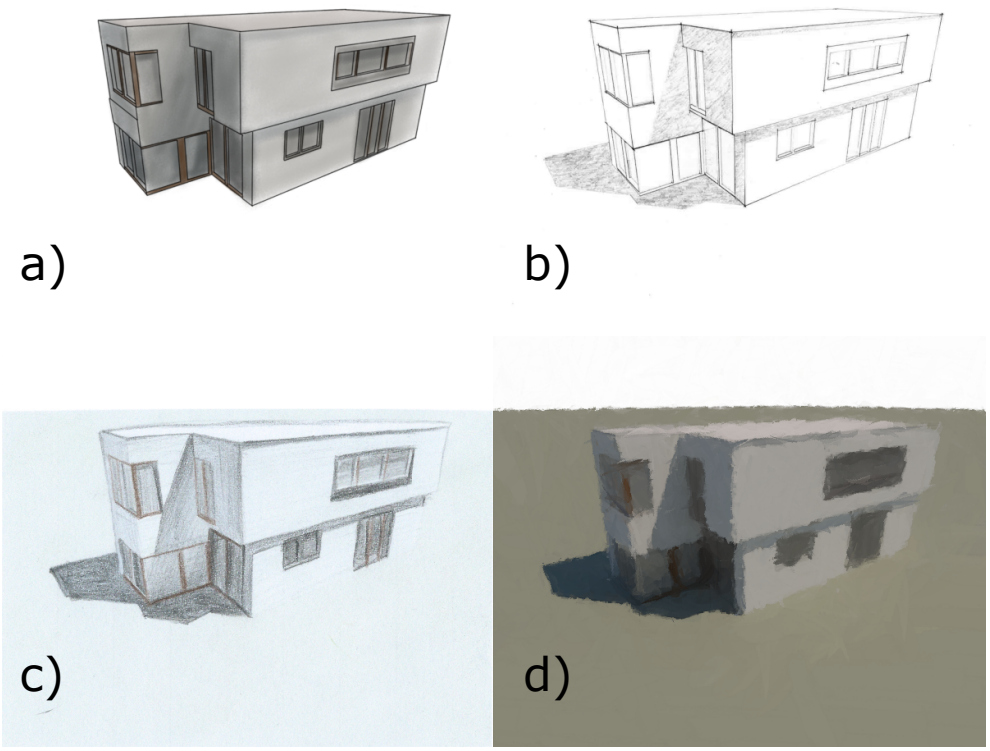

Figure 5.9: Styles of modern house model
a) Illustration style by youssrayd on Fiverr b) Sketch 1 style by nimshan12 on Fiverr c) Sketch 2 style by Barbora Kociánová d) Paint style generated by FotoSketcher

5.2 Modern house

This model is much simpler than the chapel. It is also much closer to what an architect might have when producing a sketched visualization. That is because of two reasons. First, the model is much closer to what is being designed nowadays. Second, the phase in which the architect does a visualization using a hand-drawn sketch is usually when the design is not yet finalized. Instead, it is done with just a crude and simple model of the building without a lot of details. This corresponds to this model. The styles for this model are shown in Figure 5.9. There are multiple interesting things to note about the styles. In a), the artist chose to ignore the shadow on the ground. In b), the most important part of the style is the outlines. In c), there is a lot of texture which should be retained. Finally, d) is the same computer-generated style as before. The results for this model and the styles can be seen in Figures 5.16, 5.20, 5.18, 5.19 and 5.17.

Stylization results

The stylization itself works well for all the styles of this model. The space where the style transfer works well is similar to the one of the chapel. One slightly problematic style is the paint style (Figure 5.9 d)). In this style, a lot of details in the texture are lost (see Figure 5.11). That is because there is a lot of detail

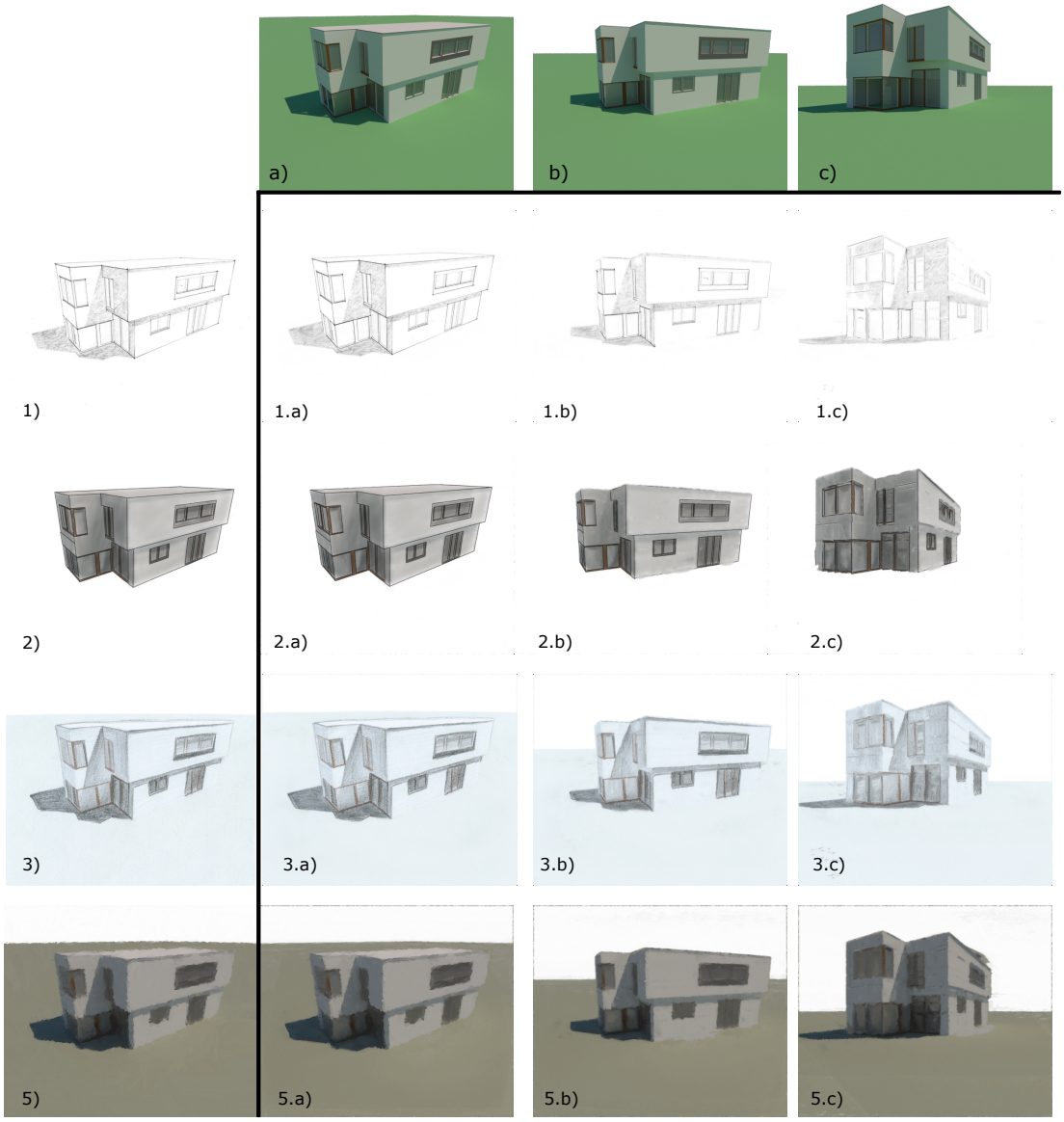

Figure 5.10: Results on the modern house model
Results on the modern house model on multiple views: a) the original view b) view from $(10°, 10°)$ c) view from $(20°, -20°)$.

in strokes that go beyond the original part to which they belong (for example the dark strokes of the window go over parts of the wall). This gives the original style a somewhat messy but sharp look. This look is lost a little in the stylized result since these strokes going over different parts of the model are not preserved. The reason why this is not so significant on the same style on the chapel model is that the chapel does not contain a lot of flat surfaces, thus the guiding is much less uniform. Because the chapel does not have uniform guides, the patches have a clearer placement, so even the strokes that are over different parts of the models than the ones they belong to may get placed correctly. With the more uniform guides, these patches do not have a clear placement and thus may not get used.

One thing to note in the sketch style (Figure 5.9 b)) in the Figure 5.12 is the left camera-facing wall. There are artifacts of darker areas intervening with white areas. The cause of this is in the original style exemplar. The two walls facing camera in the new view have almost the same values in the guide channels, they are in shadow and have the same color. But the artist chose to stylize them differently. One of them in dark shade, one in white color. The algorithm then has problems deciding patches from which of the two walls to use and produces these artifacts.

Interpolation results

With this model, problems with the removal of outlines mentioned in the chapel model section are even more visible. That is because on the modern house model there are a lot of straight outlines that bear a lot of information. This is most visible in Figure 5.14, where in b) and in the detail d), the outline in the back of the roof almost completely disappears. In other results of the interpolation, the results are a little more blurry than the original (as can be seen in Figures 5.12, 5.11 in b), d)), but otherwise of comparable quality.

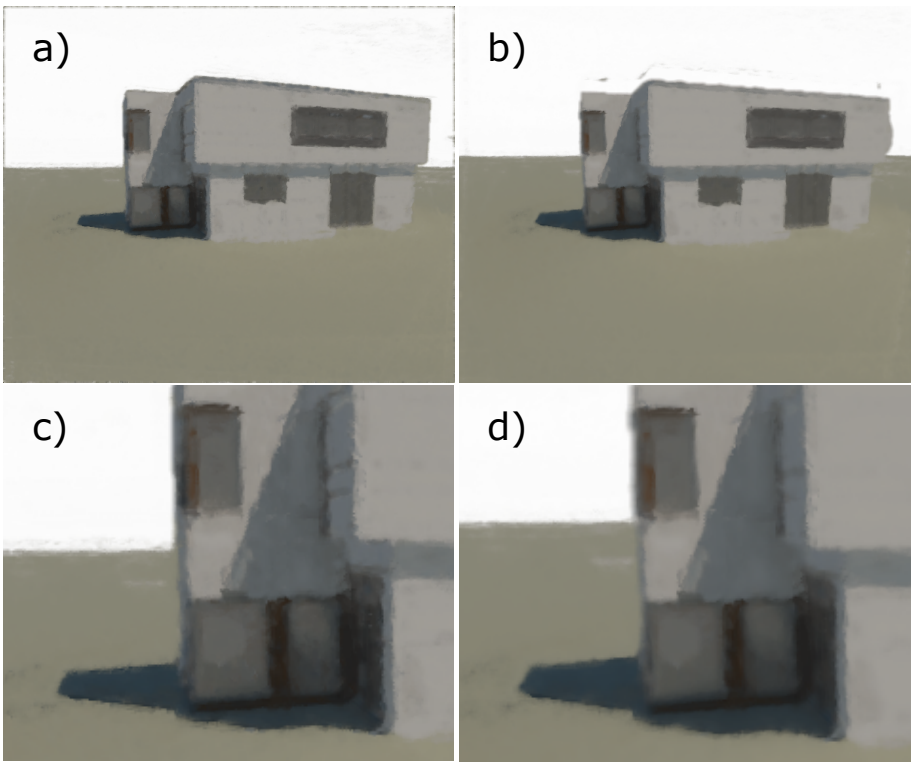

Figure 5.11: Result on the modern house – paint
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

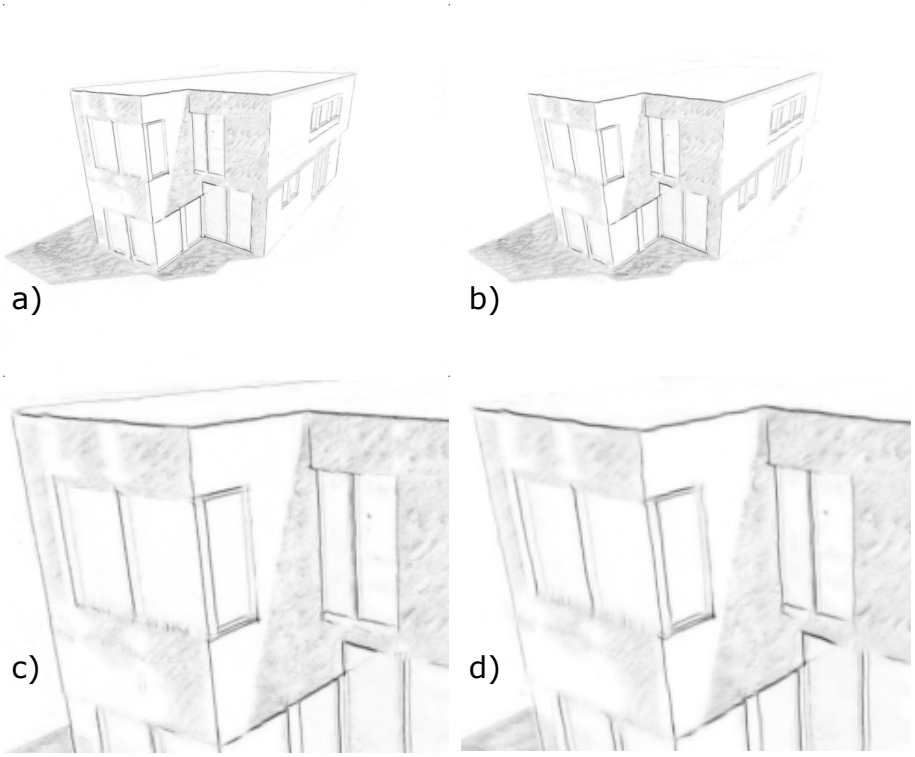

Figure 5.12: Result on the modern house – sketch
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

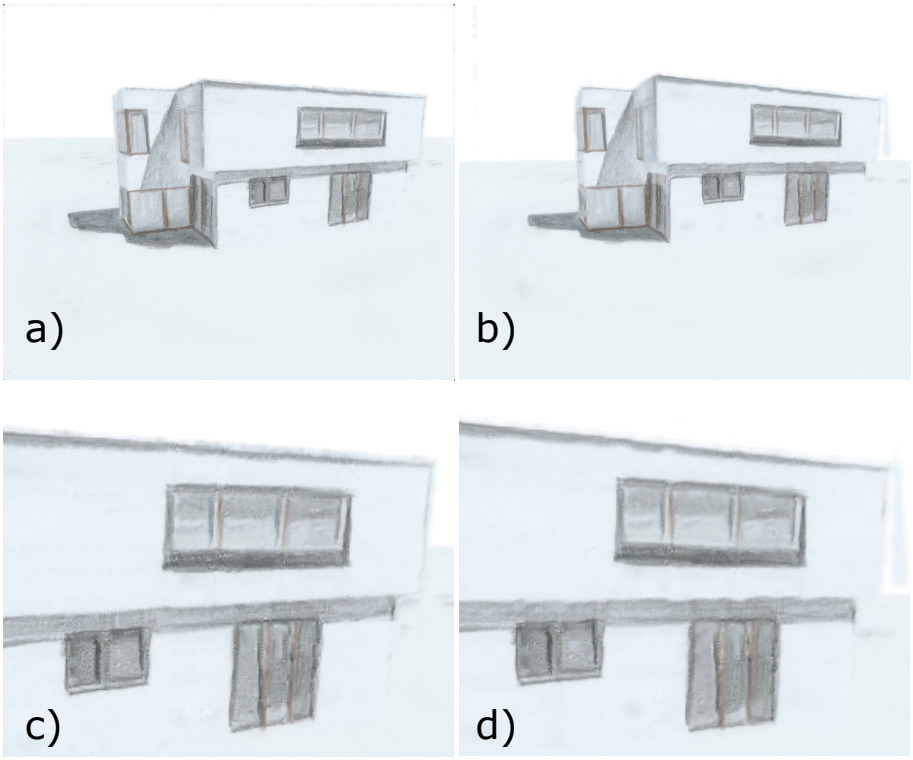

Figure 5.13: Result on the modern house – sketch 2
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

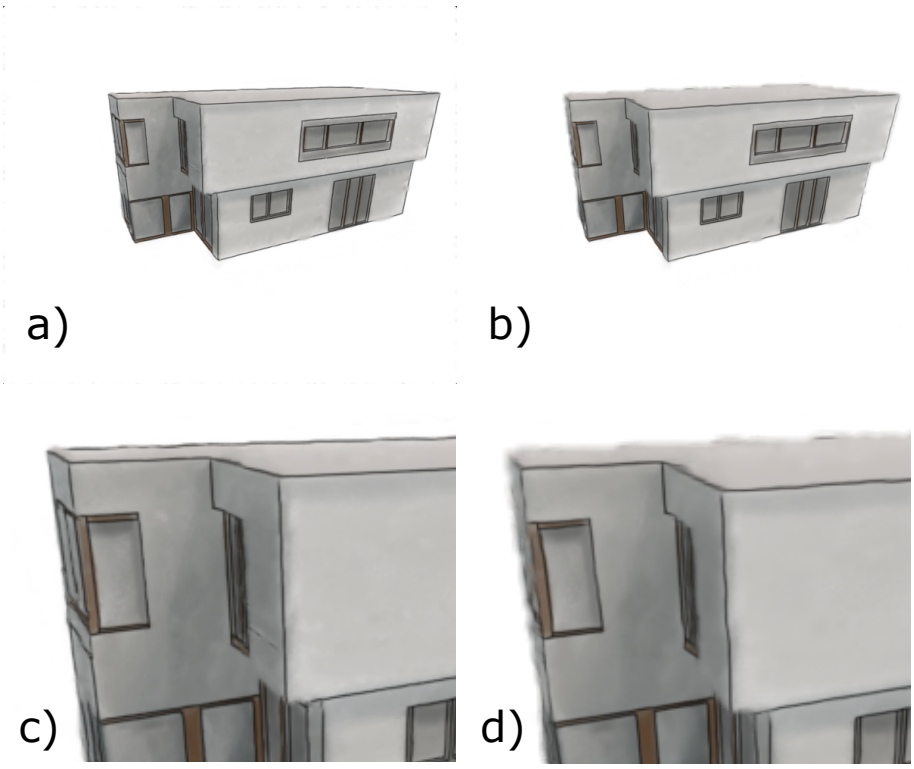

Figure 5.14: Result on the modern house – illustration
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

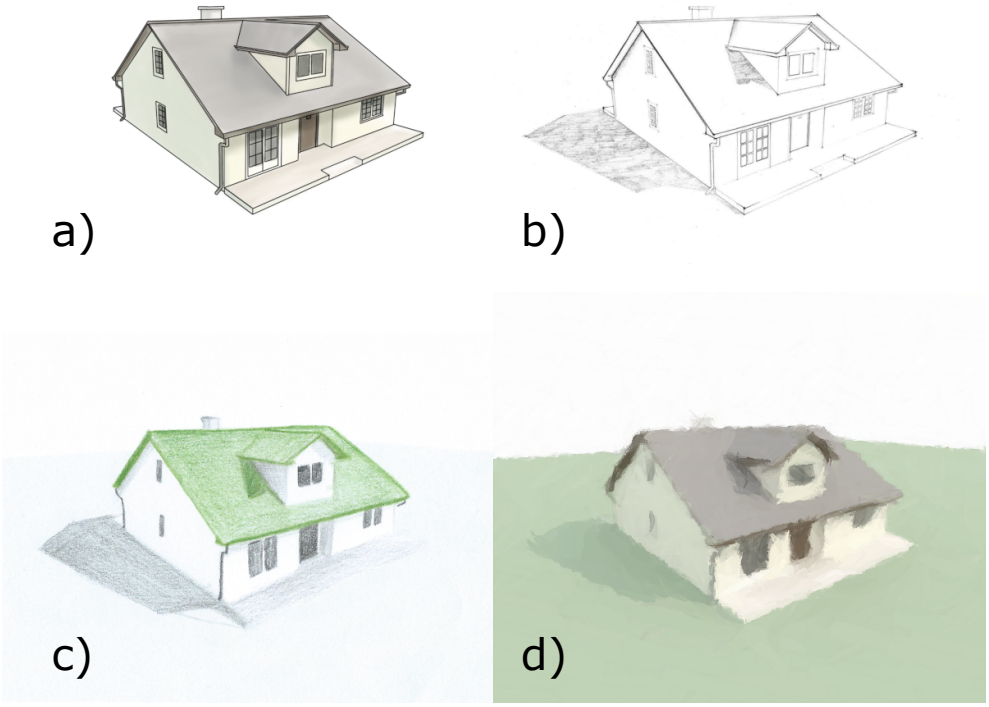

Figure 5.15: Styles of the normal house model
a) Illustration style by youssrayd on Fiverr b) Sketch 1 style by nimshan12 on Fiverr c) Sketch 2 style by Barbora Kociánová d) Paint style generated by FotoSketcher

5.3 Traditional house

The traditional house model is simpler than the chapel model, but with more details than the modern house model. It is also something that may be present in nowadays architecture. The styles for this model are in Figure 5.15.

Stylization results

Results of the stylization on the traditional house model can be seen in Figures 5.16, 5.17, 5.18, 5.19 and 5.20.

In Figure 5.20, one issue can be seen in areas that are newly visible in the view. In this model, such areas are for example under the roof. In the upper part of the window under the roof in Figure 5.20 c), the stylization visibly fails, because the algorithm cannot decide what patches to copy there. Another issue visible in Figure 5.20 is that lines that should be straight are slightly curvy, which can be seen in the line under the window.

Interpolation results

In the results, interpolation performs well again with the achieved quality comparable to the stylization itself. But a few issues are still visible.

One issue can be noted in Figure 5.18. After shifting of the NNF, if an area becomes much larger than the original area, it becomes blurry. That can be seen in the shadow in d), which becomes blurred and starts to lose the texture of the shadow in c).

Another slight issue is present in Figure 5.20. Some straight lines may become slightly curved or disconnected after the shifting. This can be seen in the image d) on the line of the dormer, or on the line on the roof of the dormer.

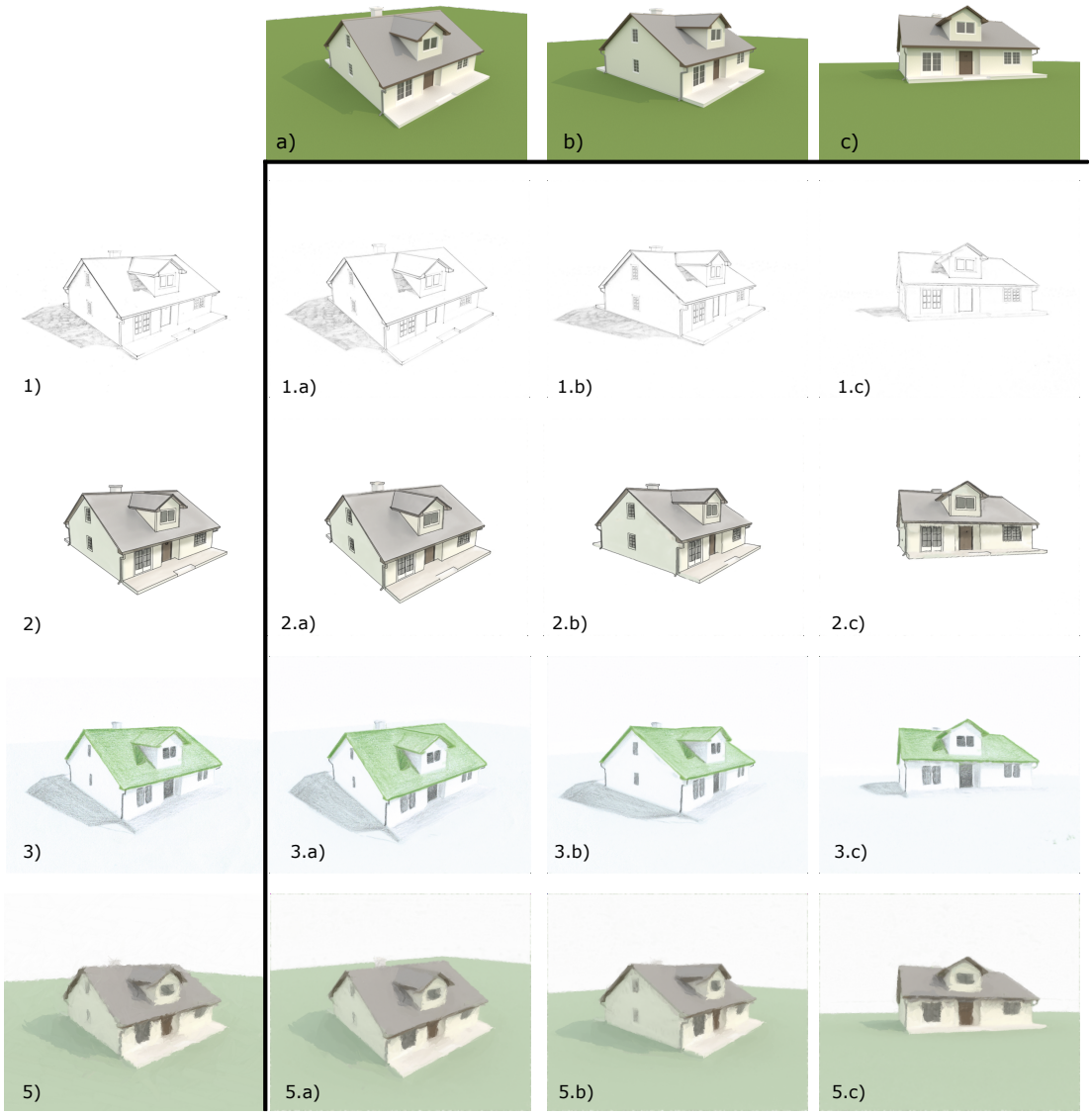

Figure 5.16: Results on the normal house model
Results on the modern house model on multiple views: a) the original view b) view from $(-10°, 10°)$ c) view from $(20°, 20°)$.

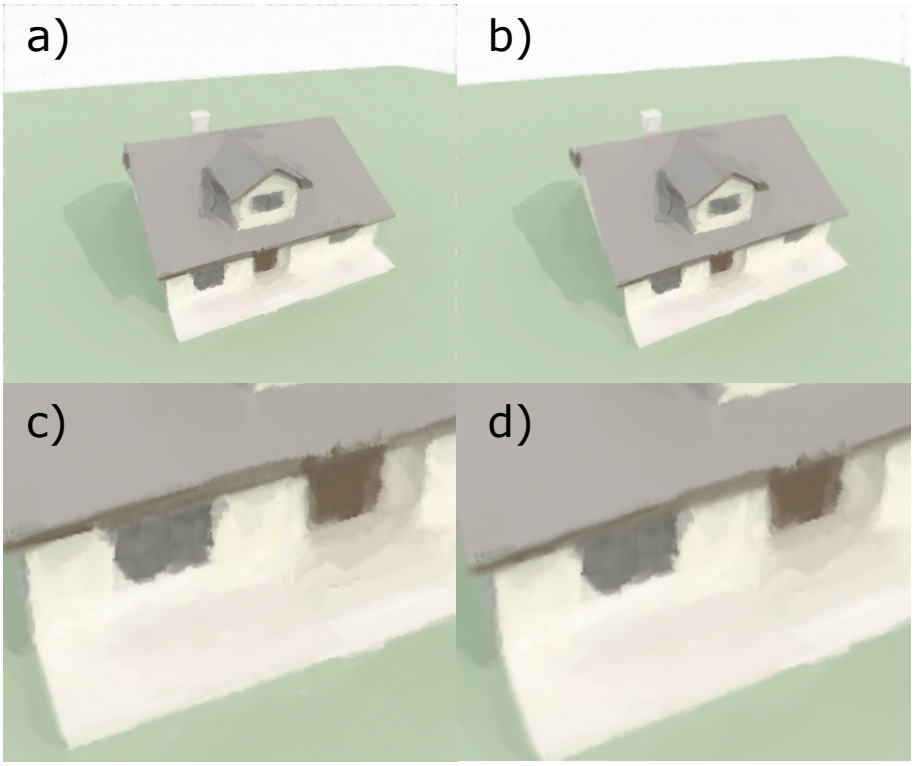

Figure 5.17: Normal house – paint result
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

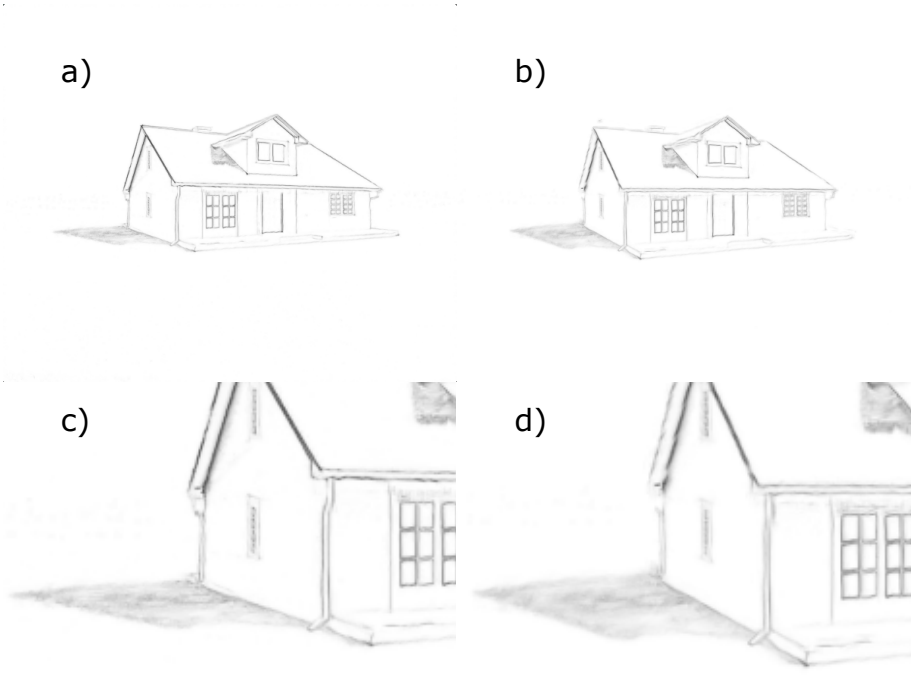

Figure 5.18: Normal house – sketch result
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

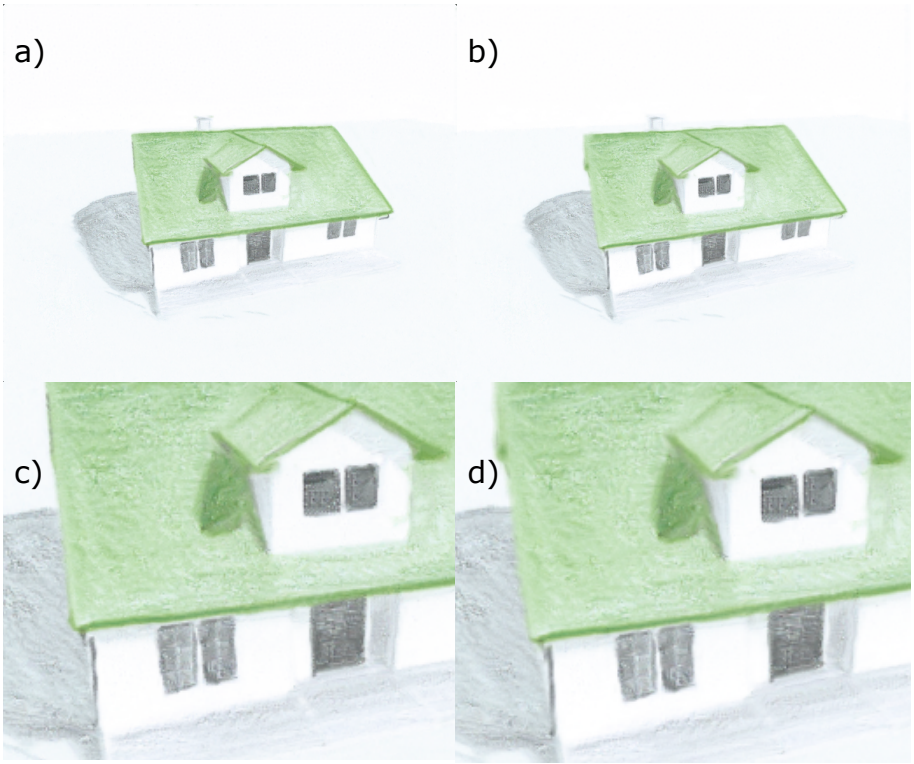

Figure 5.19: Normal house – sketch 2 result
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

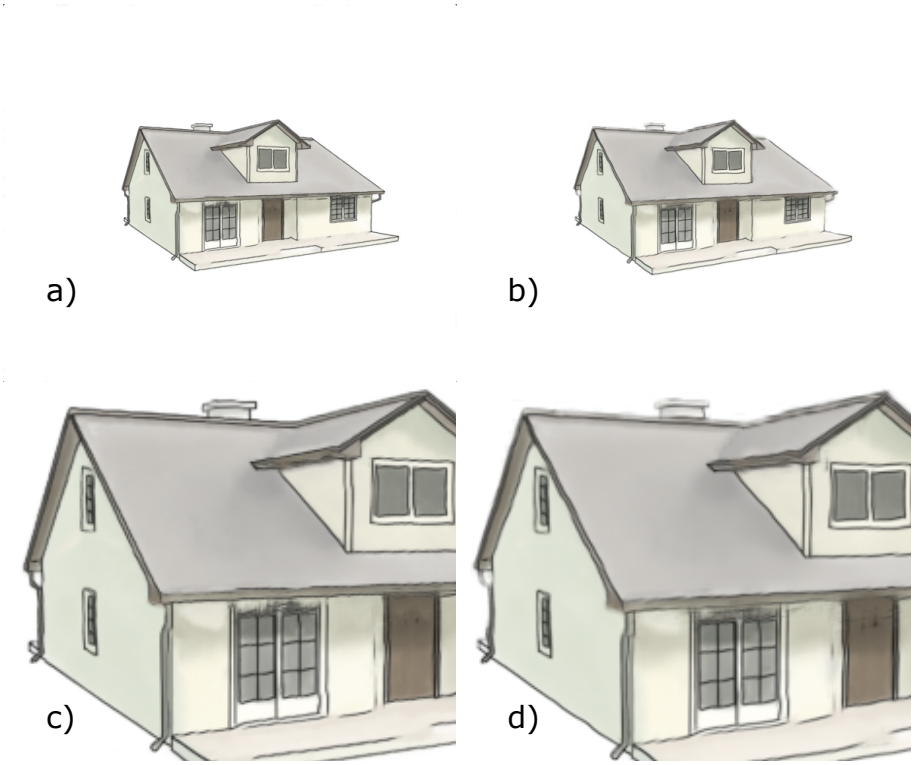

Figure 5.20: Normal house – illustration result
a) result of the stylization on grid point, b) result of shifted NNF from the a) result, c) detail of the stylization on grid point, d) detail of the interpolation result

Conclusion

In this thesis, we explored the problem of style transfer from one view to multiple different views and then provided a way to smoothly interpolate between those stylized viewpoints.

First, we explored the possibility of using certain algorithms to our problem of style transfer in the area of architectural sketches. We decided to use the StyLit algorithm (Fišer et al. [2016]) as a base for the thesis because of its results for style transfer on 3D scenes. We then introduced a few minor changes to the algorithm to improve its results on architectural sketches. We also introduced a modification to the algorithm that enabled a spatially coherent style transfer to multiple viewpoints in a parallel way. With these stylized results, we introduced an algorithm to smoothly move between them, while preserving a high quality of the stylized result.

We also believe that the approach described in this thesis could have real-world use. The use-case would be as follows. An architect would like to provide a more interesting visualization than just a sketch to a client. But the architect cannot expect the client to have a powerful computer to run the stylization and for the client to wait until the stylization finishes. Thus, the architect runs the stylization and then sends the results to the client. The client can then run the real-time viewer on their machine.

Limitations

The main limitation is the inherent limitation of the underlying StyLit algorithm. It cannot work on views that differ too much from the original view. This limits the position of the views the stylization works on.

Another problem we were not able to solve is the problem of the inaccurate style exemplars. Artists usually use strokes that go slightly over areas the strokes do not belong to and with the use of masking in the real-time viewer this causes removed outlines.

Future work

Since the field of style transfer in a spatially coherent way is an open problem that is not very widely explored, there is a lot of different ways to expand this work.

One of the areas we feel would deserve more attention are rotations in the style transfer. The angle of lines carries important information, especially in our case of architectural sketches, where hatching is a quite popular style. The angle of hatches is not arbitrary and has a purpose, but our current implementation uses only the original angle. There are many unexplored ways of addressing this problem, especially taking into account that we are working over a 3D scene, where data like principal curvature directions can be extracted.

Another area that could certainly be improved is the quality of the real-time viewer. Currently, the viewer runs in the same resolution as is the output of

the synthesis. We believe this could be improved even with the lower resolution synthesis. The available NNF could be leveraged to produce higher resolution results.

The third possible way is a straightforward expansion of the methods we introduced. For the spatially coherent stylization and for the real-time viewer, we were working just in two-dimensional space (rotations). This could be expanded to include many interesting problems. One could just expand the space by including also zoom into the rotations. Other expansions in this could area could include smooth changes in geometry. That would allow stylized animation that could be viewed from several angles.

Bibliography

Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. In *ACM Transactions on Graphics (ToG)*, volume 28, page 24. ACM, 2009.

Pierre Bénard, Forrester Cole, Michael Kass, Igor Mordatch, James Hegarty, Martin Sebastian Senn, Kurt Fleischer, Davide Pesare, and Katherine Breeden. Stylizing animation by example. *ACM Trans. Graph.*, 32(4):119:1–119:12, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461929. URL `http://doi.acm.org/10.1145/2461912.2461929`.

Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (5):603–619, 2002.

Jakub Fišer, Michal Lukáč, Ondřej Jamriška, Martin Čadík, Yotam Gingold, Paul Asente, and Daniel Sỳkora. Color me noisy: Example-based rendering of hand-colored animations with temporal noise control. In *Computer Graphics Forum*, volume 33, pages 1–10. Wiley Online Library, 2014.

Jakub Fišer, Ondřej Jamriška, Michal Lukáč, Eli Shechtman, Paul Asente, Jingwan Lu, and Daniel Sỳkora. Stylit: illumination-guided example-based stylization of 3d renderings. *ACM Transactions on Graphics (TOG)*, 35(4):92, 2016.

Jakub Fišer, Ondřej Jamriška, David Simons, Eli Shechtman, Jingwan Lu, Paul Asente, Michal Lukáč, and Daniel Sýkora. Example-based synthesis of stylized facial animations. *ACM Transactions on Graphics*, 36(4), 2017.

Oriel Frigo, Neus Sabater, Julie Delon, and Pierre Hellier. Split and match: Example-based adaptive patch sampling for unsupervised style transfer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 553–561, 2016.

Oriel Frigo, Neus Sabater, Julie Delon, and Pierre Hellier. Video style transfer by consistent adaptive patch sampling. *The Visual Computer*, 35(3):429–443, 2019.

Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Advances in neural information processing systems*, pages 262–270, 2015.

Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

Agrim Gupta, Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Characterizing and improving stability in neural style transfer. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4067–4076, 2017.

Aaron Hertzmann, Charles E Jacobs, Nuria Oliver, Brian Curless, and David H Salesin. Image analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340. ACM, 2001.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

Ondřej Jamriška, Jakub Fišer, Paul Asente, Jingwan Lu, Eli Shechtman, and Daniel Sýkora. LazyFluids: Appearance transfer for fluid animations. *ACM Transactions on Graphics*, 34(4), 2015.

Ondřej Jamriška, Šárka Sochorová, Ondřej Texler, Michal Lukáč, Jakub Fišer, Jingwan Lu, Eli Shechtman, and Daniel Sýkora. Stylizing video by example. *ACM Transactions on Graphics*, 38(4), 2019.

Nicholas Kolkin, Jason Salavon, and Gregory Shakhnarovich. Style transfer by relaxed optimal transport and self-similarity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10051–10060, 2019.

Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Trans. Graph.*, 24(3):795–802, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073263. URL `http://doi.acm.org/10.1145/1073204.1073263`.

Jing Liao, Yuan Yao, Lu Yuan, Gang Hua, and Sing Bing Kang. Visual attribute transfer through deep image analogy. *arXiv preprint arXiv:1705.01088*, 2017.

Ce Liu, Jenny Yuen, and Antonio Torralba. Sift flow: Dense correspondence across scenes and its applications. *IEEE transactions on pattern analysis and machine intelligence*, 33(5):978–994, 2010.

Alasdair Newson, Andrés Almansa, Matthieu Fradet, Yann Gousseau, and Patrick Pérez. Video inpainting of complex scenes. *SIAM Journal on Imaging Sciences*, 7(4):1993–2019, 2014.

Hisko Hulsing Raphael Bob-Waksberg, Kate Purdy. Undone. Amazon Video, 2019. URL `https://www.amazon.com/Undone-Season-1/dp/B07SVHR2KH`.

Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. Artistic style transfer for videos and spherical images. *International Journal of Computer Vision*, 126 (11):1199–1219, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

Peter-Pike J. Sloan, William Martin, Amy Gooch, and Bruce Gooch. The lit sphere: A model for capturing npr shading from art. In *Proceedings of Graphics Interface 2001*, GI '01, pages 143–150, Toronto, Ont., Canada, Canada, 2001. Canadian Information Processing Society. ISBN 0-9688808-0-0. URL `http://dl.acm.org/citation.cfm?id=780986.781004`.

Daniel Sýkora, Ondřej Jamriška, Ondřej Texler, Jakub Fišer, Michal Lukáč, Jingwan Lu, and Eli Shechtman. StyleBlit: Fast example-based stylization with local guidance. *Computer Graphics Forum*, 38(2):83–91, 2019.

Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis. *arXiv preprint arXiv:1808.06601*, 2018.

Ting-Chun Wang, Ming-Yu Liu, Andrew Tao, Guilin Liu, Jan Kautz, and Bryan Catanzaro. Few-shot video-to-video synthesis. *arXiv preprint arXiv:1910.12713*, 2019.

Yonatan Wexler, Eli Shechtman, and Michal Irani. Space-time video completion. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 1, pages I–I. IEEE, 2004.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

List of Figures

A. Electronic attachments

A.1 Stylization algorithm

The source project of the stylization described in Chapter 2 is provided. The implementation is tested on Linux only. Instructions to run the program and requirements can be found in *ReadMe.txt* in the project folder. Specifications of the configuration files are in *ConfigInstructions.txt* in the project folder.

A.2 Viewer

A.2.1 Project

The source project files are included. In this project, there is only one assetBundle for each interpolation mode. The other assetbundles can be copied from the build from *ArchStyleViewer_Data/StreamingAssets/AssetBundles*.

The most important scripts for the NNF interpolation (3.3.2) are:

- *Assets/shaders/ShiftNNFAndVoteExpandedMasking.shader*, which performs the interpolation itself
- *Assets/scripts/NNFVotingScript.cs*, which feeds data to the shader

A description of the expected AssetBundle structure is provided in *AssetBundleStructure.txt*.

A.2.2 Build

The provided build is for Windows only. Instructions to running the application are provided in *readme.pdf*.

A.2.3 Video

We provide a video showing the results of the interpolation.