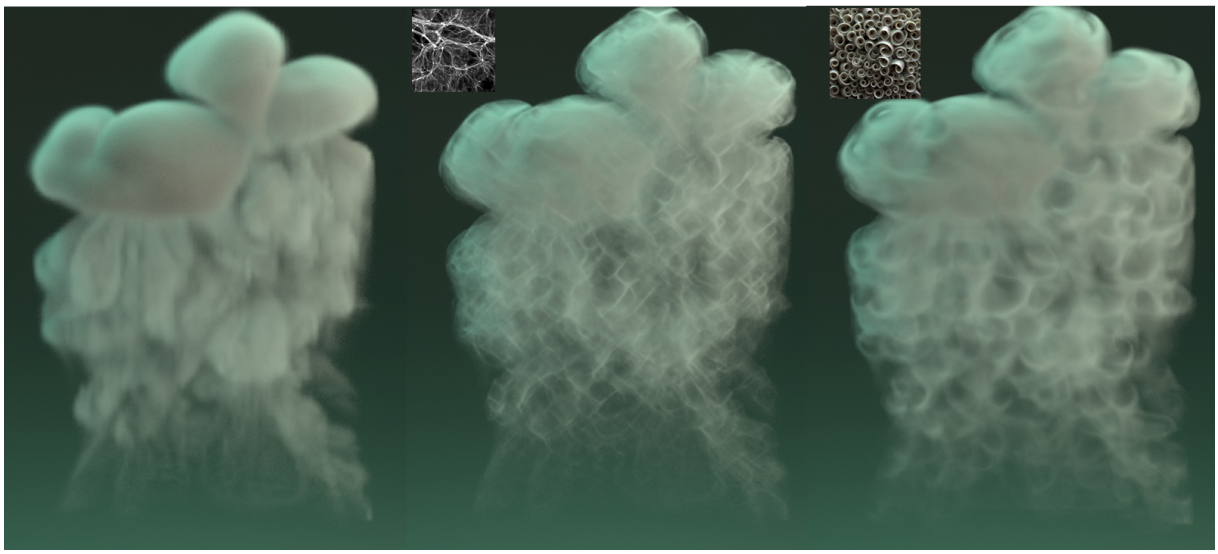
Efficient Deep CNN for Transport-Based Neural Style Transfer



Jesús Martín Berlanga

Master Thesis
March 2020

Supervisors

Byungsoo Kim
Dr. Vinicius da Costa de Azevedo
Prof. Dr. Markus Gross

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
computer graphics laboratory

Abstract

In this thesis, the speed and scalability of previous neural style transfer (NST) techniques for smoke are enhanced by training a feed-forward convolutional neural network (CNN) for stylizing smoke simulations. The feed-forward solution follows a patch-based method that considers spatial coherence of density fields while showcasing superior inherent temporal coherence compared to previous iterative NST techniques. Training in 3D space is possible by combining the CNN with advection and rendering differentiable modules.

Zusammenfassung

In dieser Arbeit werden die Geschwindigkeit und Skalierbarkeit früherer NST-Techniken (Neural Style Transfer) für Rauch verbessert, indem ein Feed-Forward-Convolutional Neural Network (CNN) zur Stilisierung von Rauchsimulationen trainiert wird. Die Feed-Forward-Lösung folgt einer Patch-basierten Methode, die räumliche Kohärenz von Dichtefeldern berücksichtigt und gleichzeitig im Vergleich zu früheren iterativen NST-Techniken eine verbesserte zeitliche Kohärenz aufweist. Das Training im 3D-Raum wird ermöglicht, indem das CNN mit Advektion kombiniert und differenzierbare Ansichten gerendert werden.

Acknowledgements

Firstly, I wish to thank Vinicius and Byungsoo for allowing me to work on an exciting project like this, with so much potential, and keeping me motivated. I greatly enjoyed discussing project-ideas with you, you made my thesis journey incredibly interesting during these six months.

Also thanks to Jingwei for providing me with a Pytorch fluid solver and helping me understand its components to integrate it on my project.

Lastly, thanks to my parents for encouraging me through my studies. And thanks to their sacrifice to support me both mentally and economically at all times. It would not have been possible without you. Thanks to my younger brother too for its contagious excitement on graphics and simulation topics, fun times skiing, and for wishing me the best on my studies.

Contents

List of Figures

List of Tables

1

Introduction

Artistically controlling fluids has always been a challenging task. Optimization techniques rely on approximating simulation states towards target velocity or density field configurations, which were mostly handcrafted by artists to indirectly control smoke dynamics. Recently, Neural Style Transfer (NST) techniques were used to artistically manipulate smoke simulation data. In this thesis, our goal is to enhance previous NST techniques for smoke by training a feed-forward convolutional neural network for stylizing smoke simulations much faster. The network has to adapt previous image-based 2-D solutions to consider 3-D density fields while considering the time-coherency that underlies the physical phenomena. The stylization can be used to enhance a dull low resolution volume with stylized details. We believe that a patch-based approach is key to generalizing Deep Convolutional Architectures for volumetric 3-D data for this reason we also explore a simple tiling method based on overlaps. Additionally, we explore architecture designs to compute stylization velocities in a multi-scale fashion, which could enable long-range correspondencies on the original style to be transferred to the target volumetric smoke.

Neural Style Transfer (NST) is a popular technique for artistically stylizing an image while keeping its original content. It computes styles by filter activations of pre-trained Deep Convolutional Neural Networks (CNNs) used for image classification, providing a rich range of styles that can model both artistic [GEB16] and photo-realistic [LPSB17, MSZM17] style transfer. Transport-based Neural Style Transfer (TNST) [KAGS19] extends image-based NST by indirectly manipulating fluid data through stylization velocity fields. The stylization velocity fields are optimized by minimizing differences between filters activations of a given target style and the style of a rendered smoke frame. Given an specified camera viewpoint, a differentiable volumetric renderer automatically enables the transferring of gradients computed in image-space to volumetric data. Temporary-coherent smoke stylizations are obtained by subsequently aligning and smoothing stylization velocity fields using the original simulation velocities. Their approach supports an unprecedented wide range of styles obtained from single 2-D images, ranging from simple artistic patterns to intricate real-images motifs.

However, the energy minimization solved by the Transport-Based Neural Style transfer is computationally expensive, taking up to 20 minutes per frame for a 3-D setup. Thus, in this work, we improve the efficiency of TNST by training a Deep Convolutional Neural Network. Our network takes as an input a patch of a smoke and outputs a velocity field that stylizes this patch given a predefined style. The patch-based approach allows our method to support general scenes, as we carefully choose a training data-set that has increased variability while also providing data augmentation in training time for better generalization. Subsequent patches have overlapping regions, which are combined together to produce seamless stylizations that are up to 2 orders of magnitude faster than the previous approach. Besides being computationally efficient, our Deep CNN patch-based approach allows stylization of arbitrarily large simulations on without exceeding a memory budget, making our method useful for production pipelines. Lastly, we benefit from the natural smoothness present on generative Deep CNNs to output an stylization that covers all smoke viewpoints with a single feed-forward approach, contrary to the original TNST that required multiple single-view passes. The contributions of our deep convolutional transport neuronal style transfer method, *DCTNST*, can be summarized as follows:

- The first feed-forward Deep Convolutional Architecture for Volumetric Stylization, enabling speed-ups up to 2 orders of magnitude faster than TNST
- A scalable and carefully fine-tuned patch-based approach that allows the stylization of large volumes of smoke (teaser) while providing computationally efficient stylizations
- Viewpoint-independent and temporally coherent smoke stylizations enabled by the inherent smoothness provided by our generative Deep CNN architecture.

2

Related Work

The artistic stylization topic has been studied for a long time, early method relied on placement of virtual strokes or image processing and filtering which suffered from either being complicated, produced limited quality results, or were not able to generalize to any style. Early style transfer methods were studied as a texture synthesis problem, and aimed at transferring pixel patches from a style image into the target image, however they were lacking of high-level style patterns. It was not until Neuronal Style Transfer (NST) methods appeared that the field was not revolutionized: They key idea of the success is to use a pre-trained convectional neuronal network to generate global style statistics that allow to change the appearance of a piece of work to look like any target image with a perceptual loss.

Style Transfer methods can first be divided into two categories: online and offline. Online neuronal style transfer rely on iterative optimization methods, for example gradient descent. However, computing the optimization process is computationally expensive and can require a lot of memory due to gradient propagation and storage, specially when considering 3D density fields, as in Kim et al. [KAGS19]. For this reason, Online neuronal style transfer is not suitable for interactive applications. On the other hand, modern offline methods move the computation burden to a training stage which can be executed more efficiently on a cluster designed for parallel computation and with more high-end hardware. Offline methods train a neuronal network in the task of stylization which can later compute stylized results much faster at test-time.

In addition, style transfer methods can be further categorized into Parametric Neural methods with Summary Statistics, and Non-parametric Methods. Whereas parametric modeling uses a global statistic, typically the gram matrix, to describe entire neuronal feature layers, non-parametric methods either try to directly aggregate pixel patches from the source image to the target image or extract neural patches from the features layers which are then compared or swapped to minimize a MRFs based loss. Non-parametric styles are capable to better capture local style characteristics allowing better preservation of fine detail, improving representation of texture styles (e.g. brick wall), and provide more flexibility to adapt to changing style across

the image: However they perform worse at capturing more structured styles (e.g. face style). For a thorough review of Neural Style Transfer methods, refer to Jing et al. [JYF+19].

Finally, Generative Adversarial Networks [GPAM+14] (GANs) have been used in many works to replace more simple pixel-losses, such as the l_2 loss, with a more complex loss learned by a CNN which ties do distinguish between ground truth and generated data. After the idea of perceptual losses were introduced and widely employed in image super-resolution, the combination of GANs and perceptual losses has shown specially impressive results: this combination is possible by using neuronal features as the input to the adversarial network. Ledig et al. [LTH+16] work was able to better capture high-frequency information compared to other approaches with results that, even though provide excellent peak signal-to-noise ratios, are often still lacking high-frequency details and are visually unsatisfying. Furthermore, taking inspiration from offline NST, GAN style-based face generators [KLA18] provide more intuitive control over latent spaces high-level attributes and stochastic variation while generating images of high quality.

2.1. Online Style Transfer

Online Neural Style Transfer algorithms perform feature matching by iteratively solving an unconstrained minimization problem through back-propagation, modifying values independently to approximate second-order statistics of a given input.

2.1.1. Parametric Neural Style Transfer with Summary Statistics

The seminal work of Gatys et al. [GEB16] enabled transferring styles between images, and since then, NST has been a topic of active research. The authors apply the gram matrix to each of several feature layers of a pre-trained VGG network to capture the style of an image. The gram matrices can be then used to define define a style loss between a target image style and an input image. Similarly, an additional content loss is used so that the objects and shapes in the output image are related to the input image. For the content loss the euclidean distance between one feature layer of original input image and the image being optimized is used. The idea is that the gram matrix can capture global statistics that make a particular style whereas the euclidean distance help preserve original high level spatially-variant structures. Feature layers at the beginning of the classifier are more suitable for the content loss because they encode high-level arrangement of objects. In comparison, information encoded in the features closer to the end of the network tend to reproduce pixel-values from the original image.

The gram matrix of a feature map G^l can be defined as in equation 2.1. Each entry ij of the matrix mesured the correlation between filter responses of two different channels of the feature map $F^l(I)$ (see fig.). After a image I is feed-forward into a classifier network then a response 2D filter map $F^l(I)$ is taken at the lth layer of size $H_l \times W_l \times C_l$ (height, width, number of filters) of the network and each filter $F_i^l(I)$ is flattened into a one-dimensional vector $\vec{F}_i^l(I)$. Then the inner product of two filters $\langle \vec{F}_i^l(I), \vec{F}_j^l(I) \rangle$ is used to provide spatial invariance.

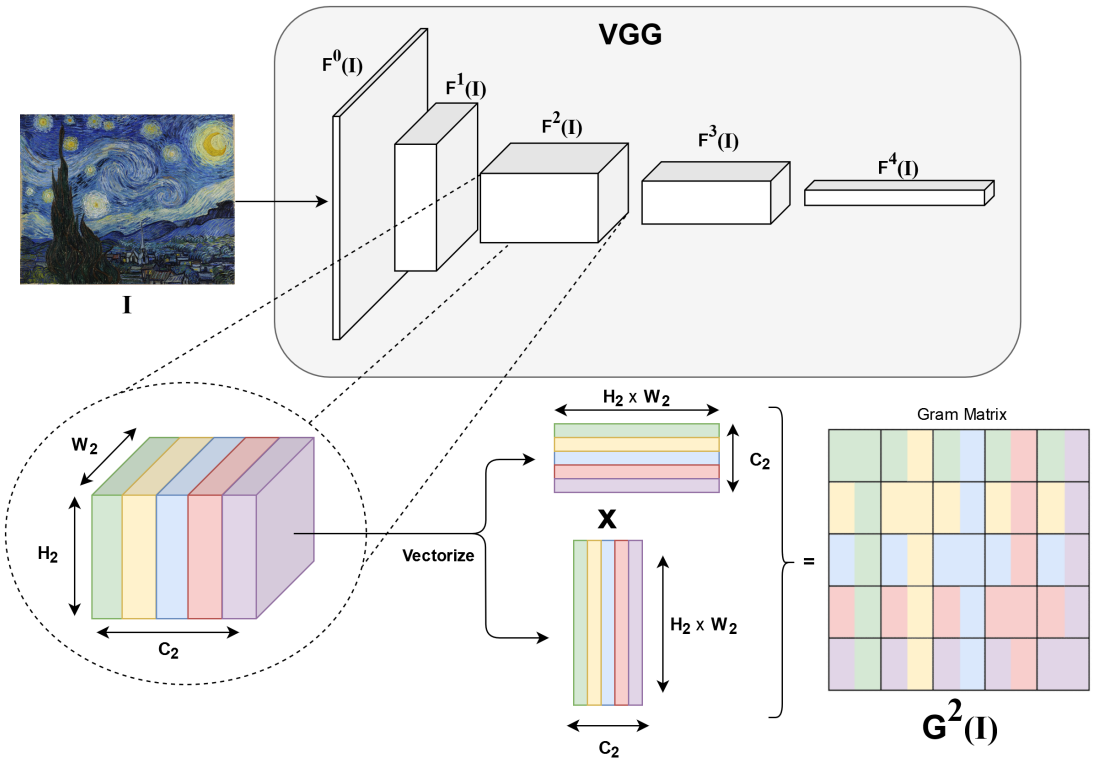


Figure 2.1.: *Computation of the gram matrix for the feature responses at layer $l = 2$ of VGG that capture style charasteristics of image I*

$$G^l_{ij}(I) = \sum_{k}^{H_l \times W_l} F^l_{ik}(I) F^l_{jk}(I) \tag{2.1}$$

The definition of gram matrix can be modified [Sne17] modified to include normalization (eq. 2.2). Optionally, normalization that can take channel dimension C_l into account [KAGS19].

$$G^l_{ij}(I) = \frac{1}{H_l W_l C_l} \sum_{k}^{H_l \times W_l} F^l_{ik}(I) F^l_{jk}(I) \tag{2.2}$$

Furthermore, since VGG was trained on Imagenet dataset, is common practice to also normalize the input image $I = (I_r, I_g, I_b), 0 \leq I_c \leq 1$ using Imagenet mean $(0.485, 0.456, 0.406)$ and standard deviation $(0.229, 0.224, 0.225)$: $I' = (\frac{I_r - 0.485}{0.229}, \frac{I_g - 0.456}{0.224}, \frac{I_b - 0.406}{0.225})$ This practice can be found for instance, on the fast neuronal style transfer implementation at pytorch examples repository.

The style loss $\mathcal{L}_s$ that measures how similar is the style of the image we are optimizing I_x to the image with the style we want to achieve I_s is in eq. 2.3. The definition uses the sum of squared differences of gram differences across different L layers, where each layer can be weighted differently w_l.

$$\mathcal{L}_s(I_x, I_s) = \sum_{l}^{L} w_l \sum_{i,j} (G_{ij}^l(I_x) - G_{ij}^l(I_s))^2 \tag{2.3}$$

And content loss $\mathcal{L}_c$ can be defined by using an unmodified version I_c of the image we want to stylize and its neuronal features on a single layer. Note that the equation corresponds to the original definiton by Gatys': Optionally the loss could be modified to take into account multiple layers and normalization.

$$\mathcal{L}_c(I_x, I_c, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l(I_x) - F_{ij}^l(I_s))^2 \tag{2.4}$$

Further improvements to Gatys et. al idea include: reduction of instabilities and artifacts by histogram [RWB17] and Laplacian [LXNC17] losses, tailoring stylization for portraits [SED16], multi-scale stylization (sec. 2.1.1), and enabling long term correspondences in video sequences [RDB16] (sec. 2.1.3). In addition, Li et al. [LWLH17] proposed an efficient way to compute style statistics by measuring discrepancies between two distributions, improving computational efficiency over the traditional Gram matrix [SZ14].

Outside the realm of images, transferring styles from images to meshes was enabled by differentiable rendering with approximate [KUH18] and analytic [LTJ18] derivatives. Closer to our work, Kim et al. [KAGS19] proposed a transport-based neural style transfer (TNST) to stylize volumetric smoke data. TNST supports complex styles generated from single images or from network activation maps, creating novel volumetric stylizations. Since their approach indirectly modifies fluids by computing stylization velocities in a fully differentiable setup, the amount of smoke present on the original simulation can be better preserved. Transport based approach does not suffer from ghosting artifacts and thinner smoke structures that the value-based approach used in traditional image stylization approaches produce when density sources are artificially created and removed to match targeted features.

Multi-scale style transfer

When using a pre-trained network to capture texture characteristics of a single high-resolution target image, the network might be only able to capture small details and bigger patterns can be missing because the receptive field of the pre-trained neuronal network is limited. Middle layers of the classification network have typically used for the style loss due to having better texture representation power whereas layers closer to the input can represent bigger-scale features. However, even when using layers at the beginning or using a combination of layers, the results are limited. Xavier [Sne17] found higher-quality results by extending the already studied idea of Gaussian pyramid for texture synthesis: his idea consists on feeding the image at many different resolutions into the pre-trained classifier rather than using many layers but only feeding the image at a single resolution.

Let I^n be an image at the n^{th} level of a Gaussian pyramid $P_N(I) = \{I^0, I^1, ..., I^{N-1}\}$ with N levels, and the corresponding gram matrix at one of the layers l be $G^l(I^n)$. Then, equation 2.3 can be modified to take into account scales as shown in equation 2.5.

$$\mathcal{L}_s(P_N(I_x), P_N(I_s)) = \sum_{n=0}^{N-1} \mathcal{L}_s(I_x^n, I_s^n) = \sum_{n=0}^{N-1} \sum_{l}^{L} w_l \sum_{i,j} (G_{ij}^l(I_x^n) - G_{ij}^l(I_s^n))^2 \tag{2.5}$$

Kim. et al. [KAGS19] followed the idea of using a Gaussian pyramid for multi-scale transport based density stylization. In their iterative optimization algorithm they first optimize higher levels of the pyramid (at lower resolution) and after a determined number of steps move to lower levels of the pyramid. In figure there is a comparison of multi-scale vs single-scale 2D density stylization implemented on pytorch that follows a similar approach. As can be seen multiple style scales play a big role to achieve visually pleasing stylization. The ability to reproduce bigger patterns is specially important to faithfully represent the style of images that are comprised of bigger objects such as flowers or spirals.

2.1.2. Non-parametric Style Modelling

Li and Wand [LW16a] were the first to propose a method to combine MRF and a DCNN for artistic synthesis, achieving improved texture coherence suitable for photo-realistic style transfer compared to parametric style transfer. They replace the Gram Matrix matching loss by a MRF regularizer for the style loss; and keep a content loss similarly as on Gatys work. For their style loss they use the higher levels of a pre-trained VGG network from which they extract local patches allowing the stylization to adapt to various local features.

$F(I)$ are the feature responses of a layer in the network after feeding the image I. The method extract local patches $\Psi(F(I_x)), \Psi(F(I_s))$ from feature responses corresponding to the image being optimized I_x and the target style image I_s. The local style loss is then defined as in equation 2.6: It measures the distance from each neural patch $\Psi_i(F(I_x))$ to its best matching neural patch from the target style image $\Psi_{NN(i)}(F(I_s)$.

$$\mathcal{L}_{s\_local}(F(I), F(I_s)) = \sum_{p=1}^{P} ||\Psi_i(F(I_x)) - \Psi_{NN(i)}(F(I_s))||^2 \tag{2.6}$$

On a more recent work from 2019, Zhao et. al [ZRL+19] combine both global and local approaches with a hybrid loss to achieve better quality results: their local loss is based on Li and Wand [LW16a] eq. 2.6 and their global loss on Gatys' eq. 2.3.

Another interesting recent work is from Texler et al. [TFL+19] which propose a hybrid global-local method which is able to efficiently process images of extremely high resolution. They use neural style transfer to generate a low resolution image used to guide a pixel-level patch-based synthesis which focus on the details.

2.1.3. Online Video Neuronal Style Transfer

If we directly apply previous algorithms to the frames of a video without more considerations temporal artifacts would appear, we would be able to see flickering as a result of drastic differ-

ences between the stylized results of two consecutive frames. Ruder et al. [RDB16] extends [**?**] algorithm to videos by adding a temporal consistency loss with a per-pixel loss between the current stylized frame and the previous stylized frame wrapped with optimal flow.

A more recent 2019 work from Jamriska et al. [JvST+19] produces video stylization with impressive quality with iterative optimization. Their method is based on non-parametric patch-based stylization. Temporal coherence is ensured with the help of two guiding channels: the patch selection algorithm have to minimize additional constraints imposed by the guides. The temporal guide for next frame is obtained by wrapping an already stylized frame with optical flow. An additional positional guide, which can be similarly forward wrapped, help further solve ambiguity between similar features that could be applied to a region, ensuring that the same particular features of previous frame will appear again on the same regions at next frame.

In the realm of 3D volume stylization, Kim et. al [KAGS19], are able to stylize density while being temporally coherent. In their method, stylized densities are obtained by optimizing a velocity field that transports the initial densities into stylization: they can achieve temporal coherence by smoothing the stylizing velocities with a given window size.

2.2. Offline Neuronal Style Transfer

2.2.1. Fast Parametric Neuronal Style Transfer

Johnson et al. [JAL16] and Unlyanov et. al [ULVL16] were the first works that improved test-time efficiency by the usage of a feed-forward approach. [JAL16] originally used batch normalization after convolutions, however on a later work Ulyanov et. al [UVL17] discovered that using the same architecture of Johnson' but using instance normalization can increase the quality of stylization.

Whereas Ulyanov' uses a feed-forward network with an architecture that takes an input image resized at different resolutions on a multi-scale CNN and a single style loss, Johnson' takes as input a single image on its original resolution and uses a simple sequence of layers, including five residual convolutional blocks at the same spatial resolution.

An architecture based on Jhonson' with instance normalization is often adopted as the basic network structure of many following works. In addition, deconvolutional layers based on strided convolution can introduce checkerboard artifacts and are often changed by nearest neighbor up-sampling and a non-strided convolution [UVL17]. Offline approaches were further improved by fine-control over stroke brushes [JLY+18], and capturing styles across distinct texture scales on a work by [WOZW16] wich uses multiple losses and sub-networks.

Haoyu Li et al. [LXC+18] use an architecture that takes the input content image at lower resolution and performs most of the operations at a lower spatial resolution space before up-sampling, allowing for even faster style transfer than [JAL16] and and specially suitable for real-time stylization of high resolution images (1024x1024 or more pixels) on embedded devices. They show images with comparable quality results to [JAL16] and show results with slightly lower style loss. To reduce the computational cost of nearest neighbor up-sampling followed by convolution in higher-resolution space while voiding checkerboard artifacts they use efficient sub-pixel

shuffle up-sampling [ALT+17]: which reorganizes (shuffles) the features after convolution in low-resolution space (cheaper to compute) to features into higher-resolution space. To further improve speed, dense convolutional blocks, which are highly parameter efficient, are used together with bottleneck layers (1x1 convolution) placed before each dense-block and which reduce the number of input feature maps. The densely connected convolutional layers use instance normalization after each convolution (post-normalization).

Driven by impressive results in image generation with adversarial training, a more recent field of study has found how translate a image from a collection (e.g. photograph with zebras) into another collection (e.g. photograph with horse) with GANs. This can be applied to stylization by for example defining a collection named Photographs and learn the mapping to, for example, Van Gogh Paintings. However, for training, the stylization then needs more than a particular image of the target style, it needs a collection of images resembling that style. CycleGAN is a popular architecture with a generator based on [JAL16] architecture. CycleGAN has recieved attention by newer papers with improvements for semantic information [LY19], color and edge constraints [ZYC+19]. Also it has been used as baseline for improved architectures: Gated-GAN [CXY+19] (multiple-styles, improved stability) and Artsy-GAN [LMZ18] (improved quality, diversity and performance). There has been some attempts to train a GAN on a 2x2 patches to blend patches borders but the result are not completely seamless [hdg].

2.2.2. Fast Non-Parametric Style Transfer

Li and Wand follow up previous work in based in MRF local stylization (sec. 2.1.2) with an approach based on a generative adversarial network to synthesize a particular style in real-time [LW16b]. The adversarial network take as input randomly sampled neuronal patches from VGG-19 layers. The texture/style loss is obtained by the adversarial network which learns to distinguish the neural patches from the style image from those extracted from the image that the generator network creates. The generator network, which decodes a latent variable to a stylized result, is trained as part of a variational auto-encoder (VAE) with a pre-trained VGG-19 encoder whose weights are kept fixed during training.

2.2.3. Multiple or arbitrary style fast style transfer

Previously discussed CNN architectures can do style transfer much faster than iterative optimization methods. However, these are still limited to the style of a single image and new models need to be trained to use a different style.

Newer extensions focus on multiple [DSK16] or arbitrary [HB17] [LLKY19] images per-model stylization. Dumoulin et al. [DSK16] use same architecture as [UVL17] but replace instance normalization with adaptive instance normalization, which includes additional scaling and translation parameters that can adapt to learn multiple styles.

Later, Huang et al. [HB17] follow up this idea with Adaptive Instance Normalization (AdaIN) to reduce model size and to be able to input an arbitrary target style to the network. Huang generator architecture use the first layers of a pre-trained VGG as a fixed encoder, AdaIN is used only once after the encoder, and its followed with a trainable decoder that mostly mirror

the VGG layers. Since the VGG encoder has many parameters the model can be slightly slower than previous single-style feed-forward approaches.

Finally Li. et al [LFY+17] improve the quality of [HB17] results by replacing AdaIN with whitening and coloring transformation. In addition, whilst [HB17] can be trained for arbitrary styles but cannot generalize to unseen styles, [LFY+17] can also work for any arbitrary unseen style. Both implementations and are much faster than Gatys' iterative optimization [GEB16], however the more expensive whitening and colouring transformations makes the model 17 times slower at 1024x1024 than previous single-style approaches. For this reason a more recent work [LLKY19] substitutes whitening and colouring transformations by a CNN transformation module.

Work of Shen et al. [SYZ17] took a different approach by using meta-networks to allow for arbitrary styles and a small model size, suitable for real-time arbitrary stylization on mobile devices. A small network is trained to produce the weights of a larger image transformation network that can stylize images. The meta-network can encode an arbitrary new style on 19 milliseconds and then be used to transfer the style to an image in 11 milliseconds on a modern GPU.

Looking into non-parametric style transfer methods, Li and Wand GAN (sec. 2.2.2) is only able to learn one style. Chen and Schmidt [CS16] propose to feed both input image and target style image into a convolutional network encoder and then that swap most similar input feature patches to the style patches. After the patch swap, the resultant features are feed into a deconvolutional network to produce the final stylized result efficiently. The deconvolutional network can be trained to produce images in multiple styles and can generalize well beyond its trained set of styles.

Finally, there is some GAN work that has been able to use a U-Net and adversarial losses to transfer a human-made sketch into coloured paintings with a dataset of paired examples [ZJL17]. Before the U-Net generator performs the decoding step, the latent code of the style image processed from VGG is added to the features of the U-Net encoder; this way the generator is able to incorporate the color style characteristics of any painting.

2.2.4. Time Coherent Fast Style Transfer

Gupta et al [GJAF17] uses a recurrent convolutional network for real-time video style transfer: at each time step the inputs are the current content image frame and the previous stylized frame, both inputs are concatenated along the channel dimension. The network architecture is similar as [DSK16] and, in addition to style and content loses, it uses the same temporal loss as [RDB16]. The optical flow is only used on training stage, facilitating speed at test-time. However, the network cannot take into account long-term consistencies where artifacts appear when objects gets occluded or new objects appear.

Chen et al. [CLY+17] produces real-time consistent stylization over longer periods of time using a feed forward network that takes as input two consecutive frames. It uses Jhonson' [JAL16] architecture but with additional operations between the encoder and the decoder: the input to the decoder is a linear combination of current frame features from the encoder and wrapped features of the previous frame. The linear combination depends on a mask that is

computed by an additional sub-network. It needs to use a optical flow sub-network and the mask sub-network at test-time.

Huang. et al [HWL+17] uses a feed-forward network who only takes a single frame as input and produces the stylized result. Optical flow is only used during training. Even though the network takes a single frame the authors claim how with sufficient training its capable to produce real-time temporal consistent video. The architecture of the stylizing network is similar to [UVL17], Jhonson et al. [JAL16] architecture with instance normalization, but uses a smaller number of channels to reduce the model size and allow faster computation.

2.3. Machine Learning

2.3.1. Deep Convolutional Network Architectures

Deep Convolutional Neuronal Network have shown great results in many field of Computer Vision, and Image Processing. Many of the recent improvements have to do with innovative changes in architecture: the idea of processing information on multiple-path has gained special attention. In addition, the idea of using blocks of layers as a structural units with build the architecture. Skip connections have been used successfully in many works to allow for easier flow of information through the network: easing training by alleviating vanishing gradients. In particular, ResNet [HZRS15] or DenseNet [HLvdMW16] have introduced and shown the benefits of convolutions blocks which include skip-connections between the layers that comprise the block.

Whereas feature reuse in a ResNet block is introduced with feature summation and requires sufficient depth in the network to improve performance, on a dense block, each layer has as input the features from all previous layers using concatenation. Feature concatiation allows to propagate and increase variation of the input on subsequent layers enabling to learn more useful features. The increased sharing of knowledge between convolutional layers in the block makes densely connected networks highly parameter efficient: meaning that they are able to obtain similar or better quality results with less parameters. Less parameters in consequence also ease computation, increase speed compared to wider or deeper architectures that yield similar results.

SRResNet and SRDenseNet are examples of network architectures based on ResNet and DenseNet and that have shown excellent results for super-resolution task. SResNet architecture has shown its best results when combined with perceptual losses and adversarial training (SRGAN) [LTH+16].

The popular U-Net autoencoder has also been used extensively on image processing. It also has been used for superesolution work both on images [HNW+19][LC19] where has been found that the skip connections between the encoder and the decoder are essential to recover details [LC19], and on 3D flow super-resolution (see tempoGAN in section 2.3.3). In addition, it has been used on sketch stylization [ZJL17] where the authors found better minimization of target loss by adding auxiliar output paths at different stages of the decoder and providing additional guiding losses to those outputs: they argue that in some cases it is possible that the network favours information from skip connections and that gradient do not flow through middle layers. Additional losses force gradients to also flow on low-level layers of the network.

Modifications of U-Net convolution blocks, originally comprised of two convolutional layers, has also been previously studied. It is common to use interpolation up-sampling instead of transposed convolutions to decrease checkerboard artifacts. In addition, the use of convolution blocks with one convolution instead of two has been used in super-resolution for increased speed [LC19]. Residual blocks have been also used on the encoder part for super-resolution [HNW+19] and dense blocks can be used both in the encoder and decoder for increased parameter efficiency [ZJX+18] [GKSC20]. On a work in the field of image reconstruction [GKSC20], U-Net has shown shown that even though a dense block has more convolutional layers this is offset by using less features per convolution, at the end yielding similar processing speed and better quality results.

Padding and border artifacts

Padding before convolution is used to preserve spatial resolution. However, convolutions can introduce border artifacts when a simple zero padding is used. Replication and reflection padding seem to slightly improve results but do not guarantee to get rid of all artifacts as it has been show that in some cases they can produce the same level or worse of artifacts [LSW+18]. Some patch-based works with overlap use padded convolutions and opt to simply crop the result if there are any border artifacts [LLWZ18].

The vanilla U-Net [RFB15] uses un-padded convolutions to avoid border artifacts, however each un-padded convolution lose some spatial resolution, and at the end the output image resolution is smaller than the input, and thus when used to process high-resolution images in multiple patches, it forces to use a minimum amount of overlapping on its tiling scheme, increasing the total amount of computation. In addition, network based on dense blocks, with considerably higher number of convolution layers, cannot opt to use un-padded convolutions because spatial resolution would be reduced excessively.

A more advanced method for padding are partial convolutions [LSW+18], originally designed for in-painting, they are able to mask their input so that invalid data holes in the input are not taken into account for prediction by the convolution weights. For padding, the idea is to treat the padding region as invalid data with the mask, so that they do not wrongly influence the prediction.

Other works aim at reducing or eliminating border artifacts by increasing context information for a patch: in the field of image compression, an additional network that takes as input neighboring patches has been used to generate a prediction within context that will be refined by the main network module to obtain the final result for a patch [MTC+18]. A multi-scale U-Net which takes sub-tiles at different resolutions as input has been used in segmentation to better capture contextual information in semantic segmentation [LSH+18].

2.3.2. Machine Learning and Volumetric Data

Volumetric data commonly used with Deep Convolutional architectures is inherently different than typical image-based datasets. Contrary to images, three dimensional volumetric data often relies in sparse occupancy voxels, scaling much worse than images due large regions

without information. Nonetheless, several methods employ machine learning for handling 3D data, specially for object classification [MS15, QSN+16, BLRW16] and human-pose estimation [GTHC19]. Volumetric Auto-Encoders were used to learn flow field deformations for object manipulation [YM16], synthesis of 3D volumetric hair occupancy data and flow fields [SHM+18], reconstructing 3D faces from single images [JBAT17], and combined with Spatial Transformer Networks to recover 3D structures from tomography data [YS18]. Recent approaches combine differentiable rendering [LB14] with voxelized data to learn richer features representations. Variational Auto Encoders [LSS+19] and Generative Adversarial Networks [STH+19] embed two dimensional data in a 3D feature space, enabling better view extrapolation when compared with screen space techniques. However, all aforementioned approaches treat the domain in a monolithic fashion, making it challenging to generalize for arbitrarily high-resolution data-sets.

Since fluid simulations easily grow to large resolutions due their volumetric nature, we have reviewed previous machine learning approaches supporting large datasets. Progressively growing network layers [KALL18, ZXL+19] [KLA18] as training advances speeds-up and increases Generative Adversarial Networks (GAN) stability in high-resolutions. Brock et al. brock2018 further improved scalability and stability when growing convolutional layers of previous architectures using self-attentive GANs [ZGMO19]. However backpropagating weight updates for architectures with large datasets might necessitate prohibitively large memory requirements. Thus, several synthesis methods subdivided the generative process in smaller patches that are combined to form a large final image. Früstück et al. [FAW19] modified generative latent codes by Markov Random Fields to avoid repetition and visual artifacts between patches, and GANs were employed for patch-based synthesis of non-parametric textures [ZZB+18] and image super-resolution [LTH+17]. However, to the best of our knowledge, no approaches combined direct CNN patch-based synthesis for high quality 3D volumetric data.

Facebook auto-encoder for volume reconstruction

Lombardi et al. [LSS+19] is capable of reconstructing a volume with color and opacity at each 3D position from a collection of viewpoints. The algorithm is specially suitable for interactive handling of video. Accounting for opacity allow to converge faster during training by widening for the discovery of good solutions whilst also makes the approach capable of reconstructing structures such as moving hair or smoke. The ability of the network to reconstruct smoke volume densities while handling temporal information is of special interest for this thesis work.

A variational KL-divergence loss to enforce smoothness on latent space enables the network to implicitly encode temporal information on latent space and learn dynamical temporal information when training the network with different images of same sequence. They are also able to generate volumes of a sequence by interpolating the latent codes of two key-frames.

The authors use a regularization term to avoid over-fitting and found good results with 3 viewpoints, and limited improvement by increasing the number of viewpoints. They are able to produce a volume of 128^3 at 90 times per second using a bottleneck architecture for a decoder, compared to the rate of 22 times per second using a convolutional decoder. They have to use background reconstruction and additional priors to help reduce artifacts that are common on challenging reconstruction problems.

An interesting part of the work of [LSS+19] is that they are capable of increasing the level of resolution of the final volume without increasing grid resolution and increasing memory requirements. They use a wrapping function that maps positions on a wrap volume to a template volume in order to sample it for ray-marching rendering.

2.3.3. Machine Learning and Fluids.

The feasibility of machine learning architectures to regress fluids representations was first demonstrated by Ladický et al. [LJS+15]. The authors approximated a Lagrangian fluid solver by Regression Forests, achieving impressive efficiency in SPH neighborhood computations. CNN-based architectures were employed in Eulerian-based solvers to substitute the pressure projection step [TSSP16, YYX16], to synthesize flow simulations from a set of reduced parameters [KAT+18] and to approximate steady-state velocity fields for predicting aerodynamic forces [UB18]. A LSTM architecture [WBT18] predicted changes on pressure fields for multiple subsequent time-steps, speeding up the pressure projection step, while matching low resolution simulations to higher resolution ones was enabled by a CNN for flow corrections [XWY19]. Differentiable simulations pipelines [SF18, HLS+18, HAL+19] that can be automatically coupled with Deep Learning architectures are recently a trend due their natural ability to interface with computer vision. GAN-based [XFCT18], slice-based [WXCT19] fluids super-resolution enhanced coarse simulations with rich turbulence details, while also being computationally inexpensive. While these approaches produce detailed, high-quality results, they do not support transfer of arbitrary smoke styles captured from single 2-D images. Additionally, their method shares the same restrictions of previous approaches to domains that fit inside the GPU. Data-driven [CT17] replaces smoke patches with data from a high detail fluid repository. Dictionary-based [Bai19] approach can process high-resolution smoke blending overlapped regions of patches using a Gaussian kernel, even though the work suffers from overfitting the tiling approach can be used for inspiration.

3D flow super resolution with GAN

The ability of tempoGAN [XFCT18] to generate surprisingly detailed flows while being temporally coherent is attributed mainly to the adversarial losses computed in feature space. They show results where adversarial losses greatly outperform manual losses on the final quality.

Furthermore, the authors show significant improvements on temporal quality by also inputting velocity in addition to the density field to the network. In addition, it is interesting that both the input velocity field and loss hyper parameter can be modified to produce different effects of stylized outputs.

They compare several temporal loss and found the best results by using a discriminator network that receives three aligned densities from the generator and from the training data set, the densities of previous and next frame are aligned to current frame by performing neuronal advection with known velocities.

For tempoGAN network architecture, the authors mention they have achieved high-quality results both with a generator based on two up-sampling layers followed by five residual blocks,

and with generator based on U-Net; the residual network gave slightly sharper results. They use nearest neighbor upsampling instead of deconvolutions to avoid checkerboard artifacts. They are able to generate a final 128^3 output in 2.2 seconds on average, with a x4 super-resolution factor and two GTX1080 Ti GPUs.

3

Method

Our work focuses on obtaining efficient, high-quality and scalable stylization for fluids, since previous offline approaches were never targeted for data-sets as large as ours. Thus, our architecture is currently restricted to single-style transfer.

Our stylization is based on global statistics and the gram matrix. Whilst non-parametric texture synthesis methods have shown great scalable and detailed results, especially when combined with GANs, our work requires that requires to transfer style from an image into a volume is incompatible with the aforementioned approaches: 2D patches, either at pixel space or feature space, from the style image cannot be directly aggregated into the higher-dimensional volumetric density field. In addition, our works aim for producing a stylizing velocity field and not directly densities to better be able to preserve the original amount of density (See TNST approach 2.1.1).

Finally, whilst adversarial training has shown great results in previous work, there is not a straightforward extension into stylization, since GAN approaches often require a collection of ground-truth data for the discriminator. GAN approaches can be more complicated and difficult to train due to instabilities and lack of control over the discriminator. For this reason, this work focus on establishing a first baseline of volumetric stylization with manually designed perceptual losses.

3.1. Single-Style Training Pipeline

This section explains the approach to train a model that learns to change an input density field to look alike a single target style image. To ease the understanding of the approach, a 2D method is first explained. Then the necessary changes to extend to 3D are explained. The 2D method can be used to compare how quality and other factors are affected after making changes in the pipeline to extend to three dimensions. In addition, it is very useful to have a 2D

implementation to train and make quick experiments before having to wait for longer training times of a 3D network. The 2D/3D sections help to get a high-level overview of how all the components are interconnected in the computation graph before diving into the details of the network architecture. Some of the illustrative diagrams use pictures from the results for a better understanding.

3.1.1. 2D

The input to the model and the deep convolutional neuronal network is a un-stylized density field d. The output of the neuronal network is not directly the stylized density field $\hat{d}$ but instead a velocity field $\hat{v}$ which can be used to transform d into stylization using the advection function $\mathcal{T}$. This follows the idea of transport-based stylization [KAGS19] which allows decreasing the artifacts on the final stylized density (see sec. 2.1.1) reaching higher-quality results. The velocity vectors that the network learns to produce $\hat{v_n}$ are normalized: in order to properly transport the density, it is required to de-normalize the velocity field before using advection. This is done by multiplying each vector in the velocity field by the size of the density field we want to transform.

$$\hat{d} = \mathcal{T}(d, \hat{v}) \tag{3.1}$$

Once a 2D stylized density field $\hat{d}$ is obtained it can be feed forward into a pre-trained 2D image classifier, from which gram matrix of filter responses at different layers are compared to those responses of the target style image I_s, forming a perceptual style loss $\mathcal{L}_s(\hat{d}, I_s)$ (Gatys' approach explained more in detail in section 2.1.1). More specifically, features are extracted from four convolution blocks conv1_2, conv2_2, conv3_3, conv4_3 of VGG-16 after ReLU activation functions have been applied to the features, and are weighted equally at the loss. The weights of the VGG network are kept fixed during training and are not updated. Note that VGG-16 expects 3 input channels so the single-field density field needs to be replicated before using it as an input to the classifier.

Furthermore, given shown importance of multi-scale stylization to achieve pleasing results (see sec. 2.1.1) a N-scale loss $\mathcal{L}_s(P_N(\hat{d}), P_N(I_s))$ (equation 3.2) can be used instead of single scale loss by computing the Guassian pyramid $P_N(\hat{d})$ of $\hat{d}$. The pipeline for this approach is shown in figure 3.1.

$$\mathcal{L}_s(P_N(\hat{d}), P_N(I_s)) = \sum_{n=0}^{N-1} \sum_{l}^{L} \sum_{i,j} (G_{ij}^l(\hat{d}^n) - G_{ij}^l(I_s^n))^2 \tag{3.2}$$

However, since the resolution of density patches is limited, further down-scaling stylized densities on a pyramid can result in unsatisfactory quality results. For this reason, alternatively only the style image (which can be high-resolution) can be downscaled on a Gaussian pyramid, this modification is shown in fig. 3.2 and equation 3.3. Note that since the size of the gram matrix only depends on the number of filters of the feature layer, gram matrices will have the same number of columns and rows even for differently sized images.

$$\mathcal{L}_s(\hat{d}, P_N(I_s)) = \sum_{n=0}^{N-1} \sum_{l}^{L} \sum_{i,j} (G_{ij}^l(\hat{d}) - G_{ij}^l(I_s^n))^2 \tag{3.3}$$

The advection operator must be fully differentiable so that the losses attached to the model can propagate backward through it. To achieve this requirement, a tensor-based PyTorch advection implementation which supports first order, and second-order MacCormack method is used. Whereas second-order advection provides a higher quality result, first-order advection is a simpler mapping than second order, it is faster to compute and should be easier to learn. The advection does not take into account boundary conditions.

Choice of pre-trained image classifier

VGG is used on Gatys paper and is typically adopted on many works but experimenting with different pre-trained networks and layers can result in very different stylizations for the same style image. For 3D density stylization Kim. et al [KAGS19] use inception for their stylization because the model has a smaller number of parameters and thus helps in their already memory-bound iterative optimization. However, the size of the image classifier is less of a problem for the feed-forward approach because the configuration of patch-size allows us to keep memory usage under control and the classifier is only needed at training-time. We have found that style loss based on VGG-16 is easier to tune (selection of feature layers and weights) to obtain good results since VGG-16 is able to produce richer feature responses [Inc].

Residual velocities and auxiliary losses

A more advanced approach to previously shown pipeline consists of adding additional output paths at several depths of the network together with additional losses for those outputs. This approach is inspired from [WOZW16] and [ZJL17]. [ZJL17] uses additional decoders and losses on U-net architecture, whilst [WOZW16] attach losses of increasing scales at different stages of their network. The additional losses can work as shortcuts for the optimizer to deeper layers of the network, allowing gradients to flow easier and ensuring different parts of the network update their weights.

The resolution of each of the output velocities $\hat{v}^0, \hat{v}^1, ...\hat{v}^N$ has a correspondence to one level n in a Gaussian pyramid with N levels and can be used to advect a pyramid of input densities $P_N(d)$ and obtain the stylized result at each level: $\hat{d^n} = \mathcal{T}(d^n, \hat{v}^n)$. The computation of the density field pyramid $P_N(d)$ is only required at training time in order to be used for advection with additional velocity fields and construct style losses per scale. At test-time only the highest-resolution velocity field $\hat{v}^0$ is used for advection. The auxiliary losses $\mathcal{L}_s^n$ are single-scale losses for a corresponding level of the pyramid $\mathcal{L}_s^n(\hat{d^n}, I_s^n); n \geq 1$ whereas a multi-scale loss that makes sure all scales information are taken into account for the final result is used at the highest resolution density field $\hat{d}^0$ (either equation 3.2 or 3.3).

In addition, inspired by incremental optical-flow architectures [HTL18, ZYCD18], the neuronal network outputs can be configured so that they correspond to a set of intermediate residual velocities v_+^n. The residual velocity at each level v_+^n is added to the bilinearly upsampled velocity

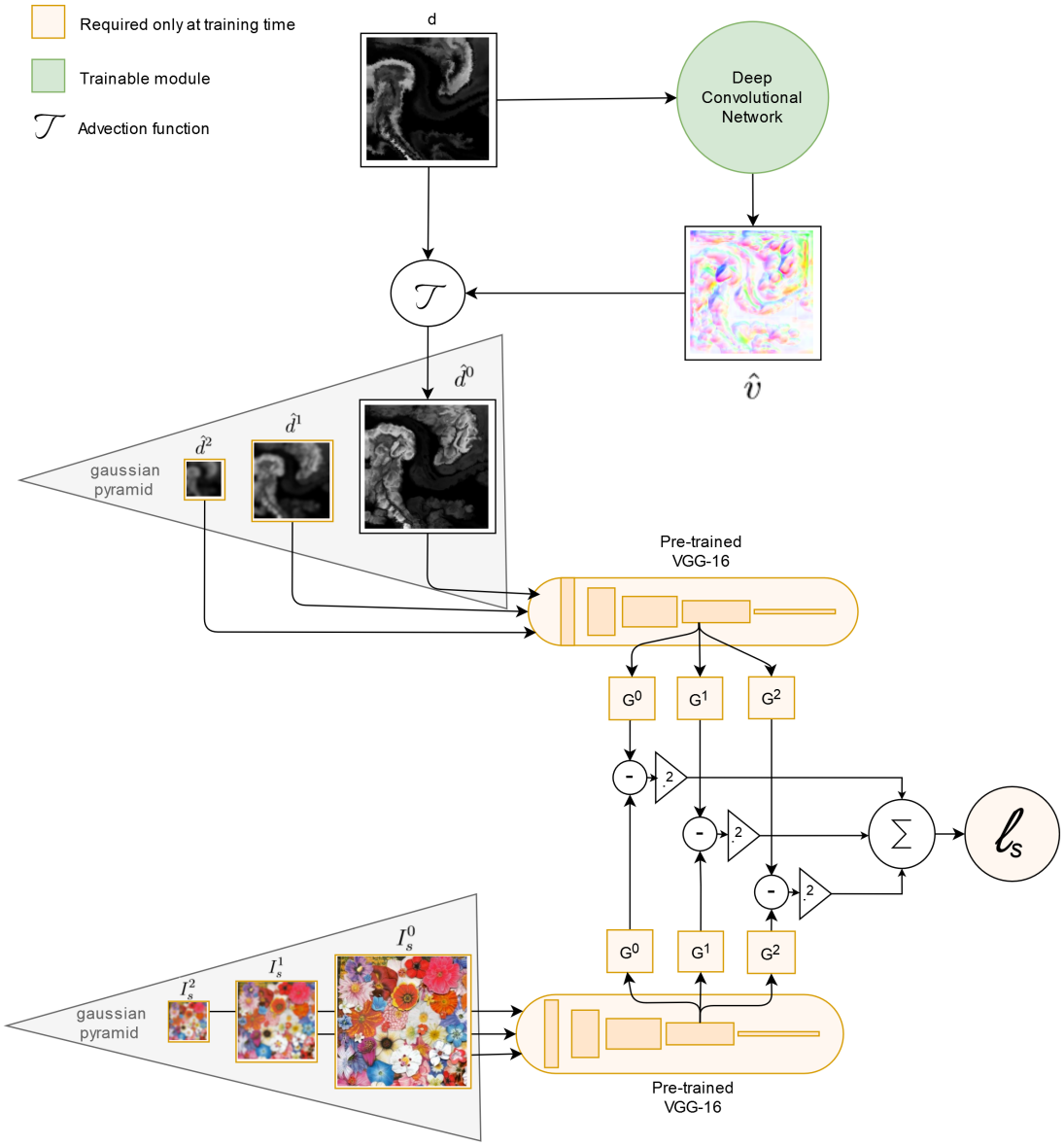


Figure 3.1.: *Basic Training Pipeline. After obtaining stylized density field $\hat{d}$ a Gaussian pyramid is used as part of a three-scale perceptual loss which targets style on image I_s. In this example only one of the layers of VGG is used for stylization. Note that this figure considers the neuronal network as a black-box to simplify the illustration. For details on how each network output is obtained refer to network architecture at section 3.2.*

found at one higher level $\hat{v}^{n+1}$ in the pyramid, yielding the next scale velocity in the pyramid $\hat{v}^n$. The learned residual velocity at each scale is normalized so the vector's magnitude does not need to be re-scaled before the addition. The idea of computing the final velocity field incrementally is that the problem can be decomposed in smaller easier steps. Furthermore, by decomposing the velocity field is possible to do additional post-processing on particular scales.

$$\hat{v}^n = v_+^n + upsample(\hat{v}^{n+1}) \tag{3.4}$$

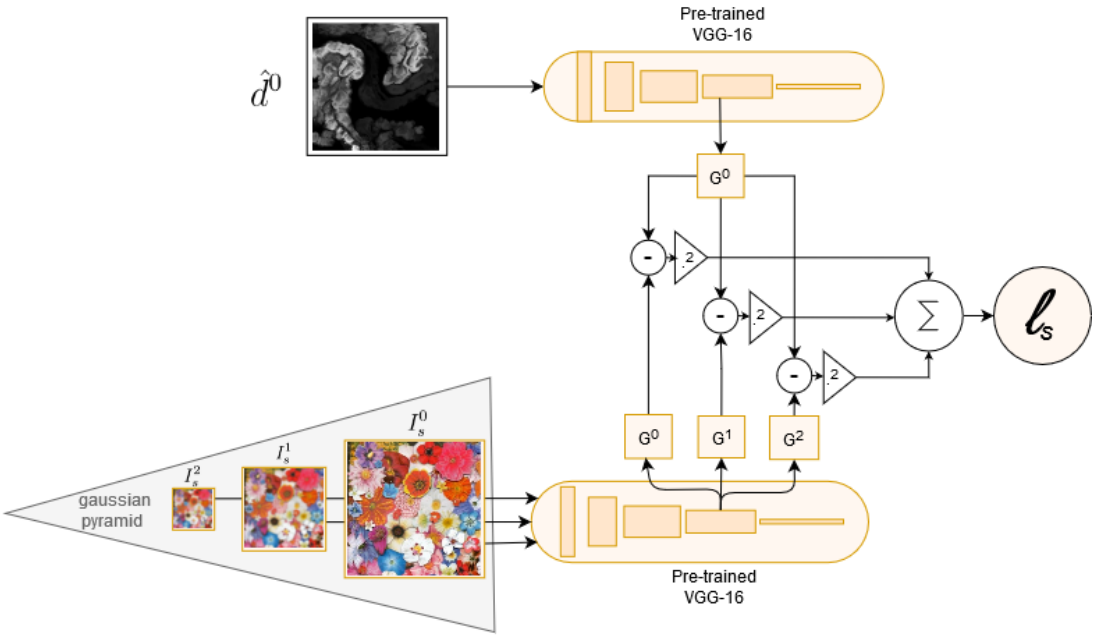


Figure 3.2.: *Original multi-scale loss idea from figure 3.1 is modified so that a Gaussian pyramid is used only on style image I_s.*

Figure 3.3 illustrates how both ideas, auxiliary losses and incremental computation of velocity field, can be combined.

3.1.2. 3D Volume

In the 3D case, the input to the 3D network is a single-channel 3D density field $d : R^3 \rightarrow R$ and the output a collection of residual 3D velocity fields that can be used to reconstruct the final stylizing 3D velocity field $\hat{v} : R^3 \rightarrow R^3$ and stylized 3D density field $\hat{d} = A(d, \hat{v})$.

The pre-trained network used for the style loss takes as input 2D images so it is necessary to find a way to connect the output 3D field $\hat{d}$ with the 2D perceptual style loss. This is done by obtaining 2D renderings $R_{\Theta_i}(\hat{d})$ from multiple views $\Theta_0, \Theta_1, ...$ which can be then used as input to the pre-trained image classifier. The rest of the training pipeline is kept the same as in the 2D case.

The camera positions are parametrized by a three-dimensional polar coordinate system: $\Theta = (\theta, \phi)$ where θ represents the azimuthal angle, and ϕ the polar angle in a sphere.

Sampling new camera positions at every training step allow us to avoid biasing the solution into a particular view which would make the smoke showcase sharp patterns when the camera is aligned with a particular view but blurred features appear after the camera moves or rotates away from that configuration. Camera placement is done by uniform sampling of ϕ angles in $[-45^o, +45^o]$ range and choosing equidistant θ angles which are then all shifted with a randomly generated value θ_{offset}. Equidistant placement on azimuth ensures the whole volume is covered, avoiding to place cameras too close to each other in some regions while leaving gaps in others. Camera placement avoids angles near the poles $\phi = -90^o$ or $\phi = 90^o$ because in practice the

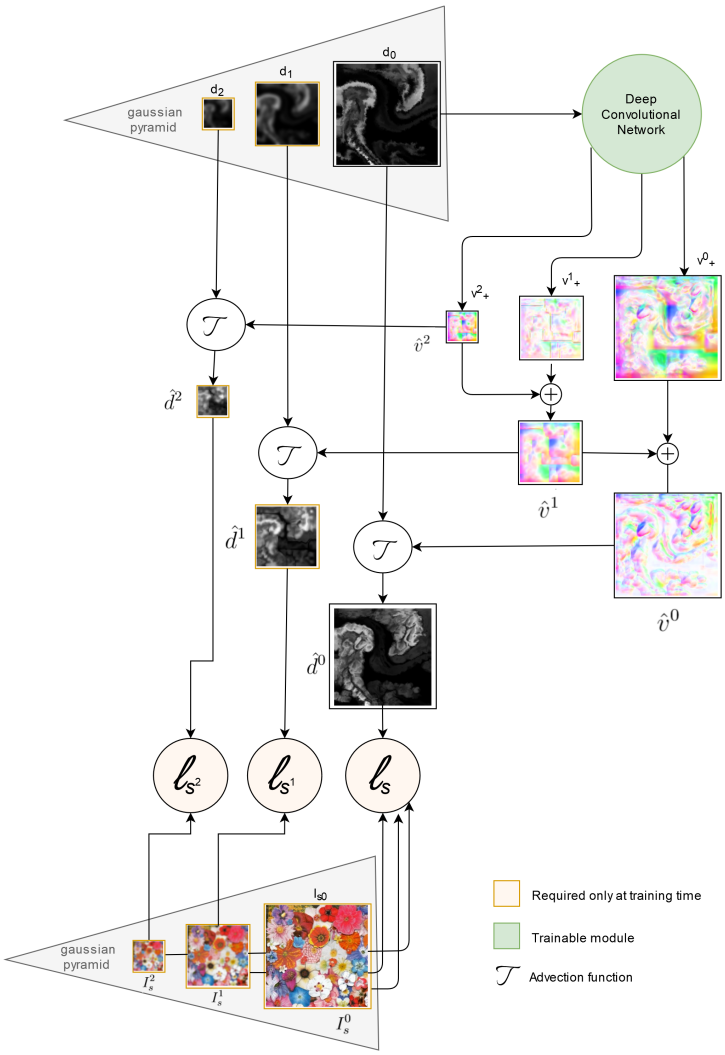


Figure 3.3.: *Residual Training Pipeline. Auxiliary losses target a particular style scale whereas the final higher-resolution output can be used as part of a multi-scale loss. Note that this figure considers the neuronal network as a black-box to simplify the illustration, for details on how each network output is obtained refer to network architecture at section 3.2.*

camera used on the final rendered animations looks at the smoke from the sides and not from high grazing angles. Using a high number of views allow to cover the whole volume more uniformly, for this reason, is recommended to use the highest amount of views as possible on training: on a high-end GTX 1080 Ti GPU, it has been possible to have up to 16 cameras for patch sizes 64^3 to 80^3 without running out of memory.

However, unlike [KAGS19] which uses iterative optimization to samples the camera positions at every step and slowly change the original density field into stylization, when using a neuronal network it is not as clear how well the neuronal network weights will be able to keep information from many views and how well will be the training convergence curve - note that after sampling new cameras and the optimizer updates the network weights it could be undoing previously stored information that was geared toward another view. For this reason, the implementation can be configured to use fixed positions for the cameras, the comparison of fixed views against

sampled views its discussed in the results section.

Similarly, as with the advection function, the renderer must be differentiable so that gradients can back-propagate during training. This is done by implementing a discrete direct volume rendering equation with tensor operations from the PyTorch library. For direct volume rendering $R(d)$, the volume can be traversed either forward or backward along depth D dimension, accumulating densities d_i along the depth axis with opacity factors o_i (eq. 3.5). The opacity factors are based on light transport (eq. 3.6), considering a beam of light that traverses along Z-axis the opacity is set according to the amount of radiance remaining after absorption events. The final values are normalized so that they are in a valid image range. A global extinction multiplier σ_e is used together with absorbance factors that are set to the density values, this way parts of the smoke with more density will absorb more light. The extinction multiplier is set to 0.075 for all experiments.

$$R(d) = \sum_{i=1}^{D} d_i o_i \tag{3.5}$$

$$o_i = exp(-\sum_{j=1}^{i} d_j \cdot \sigma_e) \tag{3.6}$$

The orthogonal rendering can be easily performed by applying the direct rendering equation on the depth Z-axis of the density tensor after it has been transformed into a particular view by using a 3D rotation matrix with $pitch = \phi_i$ and $yaw = \theta_i$. The yaw operation on the up y-axis is performed before pitch. Rotation can be done by encoding the rotation matrix on a spatial transformer layer that allows for differentiability [JSZK15].

However, when rotating the density patch, which is square-shaped, some gaps without smoke can appear on the viewport and in theory the network could learn to over-stylize the borders - generating artifacts that could difficulty a tiling approach. One idea that aims to solve this problem is to add replication padding to the stylizing velocity and patch density before the density is advected and while keeping the same rendering-canvas resolution - note that padding is only added after the network has processed the patch with its original dimensions. Figure 3.4 shows how renderings $R_\Theta(\mathcal{T}(pad(d), pad(v^*))$ looks like after the padding approach which extrapolates stylizing velocities into a bigger patch. However, in practice we have not noticed any benefits (e.g. reduction of artifacts) when using this idea and training can take longer to reduce the style loss; for this reason, we do not add padding/extrapolation of density/velocity on the experiments shown in the results section.

The simplicity of the renderer is a positive feature because it doesn't slow down computation and training. The main purpose of the renderer is to propagate gradients through the smoke, the method does not necessarily need to use a more advanced and realistic renderer which could also increase the need for fine-tuning.

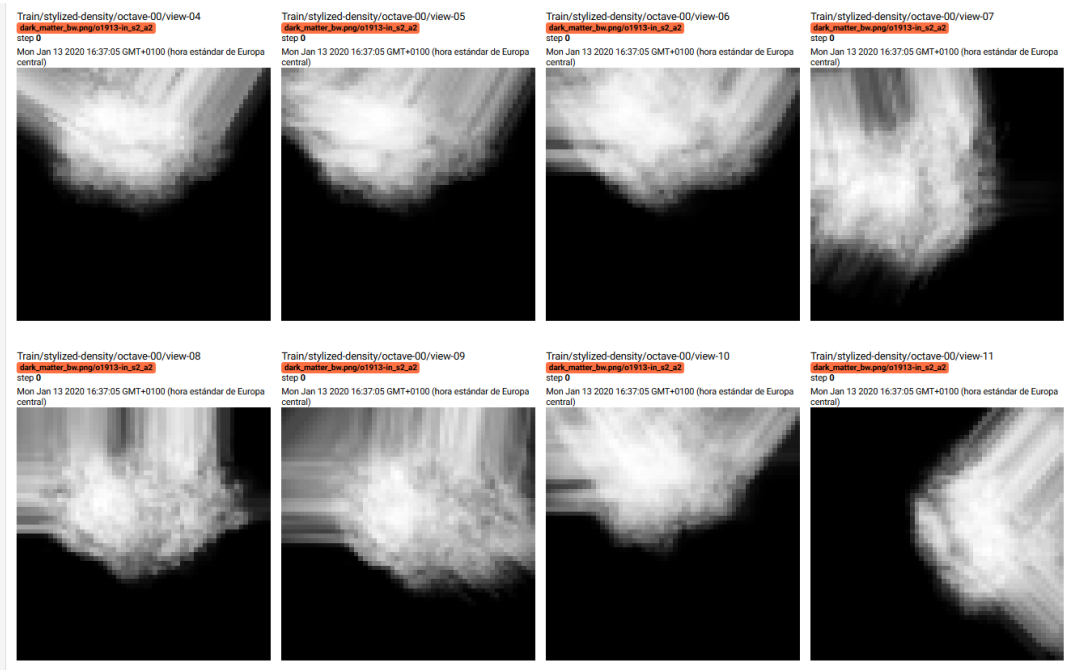


Figure 3.4.: *Views of stylized patch with density padding and velocity extrapolation*

3.2. Neuronal Network Architecture

The adopted network architecture is based on the U-Net design, which has been successfully used and studied in many works, typically for image segmentation but has also be used in works of super-resolution of both images and 3D densities (patchGAN). The network can be used for 2D by using 2D convolutions and bilinear interpolation operations and modified to work in 3D by changing the operations to their analogous: 3D convolutions and trilinear upsampling.

The U-Net design has several desirable architectural characteristics: First, it follows the shape of an auto-encoder where the receptive field varies in scale after downsampling/upsampling operations that occur following each conventional block - this is suitable for multi-scale stylization.

Secondly, the skip connections between the encoder and the decoder allow both faster training and have been proven to be effective in recovering not only high-level structures but also more fine-grained details [LC19].

In addition, the U-Net has been previously used in many works (mainly in image segmentation) to process in a tiling manner high-resolution images.

Finally, the U-Net model does not including any fully connected layer, which would be typically placed at the bottleneck of an autoencoder, making the network fully convolutional. This has the advantage that the network is not limited to process patches of a fixed resolution. This can be especially useful for the patch-based approach (explained in sec. 3.3) by allowing for a bigger receptive field that includes more context information when increasing patch-size at test time. During training the maximum allowed patch-size is more restricted than at test time due to memory limitations.

The customized design of the U-Net used in this work replaces the transposed convolutions, that the original UNet uses for upsampling, by interpolation upsampling: The 2D version uses bilinear interpolation and the 3D version trilinear interpolation. This way checkerboard artifacts are avoided. In the original U-Net decoder, the number of channels is decreased by a factor of 2 on the convolutional block and an additional factor of 2 on the transposed convolution used for upsampling. In our design, a convolution block in the decoder decreases the number of channels by a factor of 4 and then follows the bilinear/trilinear upsampling.

Hyperbolic tangent activation functions are used on the output layers so that the output values can also be in the negative range, allowing for negative components of velocities. The output activation function range is down-scaled so that velocity magnitudes do not create unreasonable transformations (too large transformations that can difficult training). We had success by using a coefficient of 0.128 for a patch size of 80^3 for the Tanh, since the velocities are normalized this means that the maximum allowed translation of density along each axis is of $0.128 * 80 = 10.24$ voxels. On 2D, a coefficient of 0.08 works well for a patch size of 128^2.

The architecture optionally allows for the computation of residual velocities by using additional 1x1 (1x1x1 for 3D) non-strided convolutions at different stages of the decoder as shown in figure 3.5.

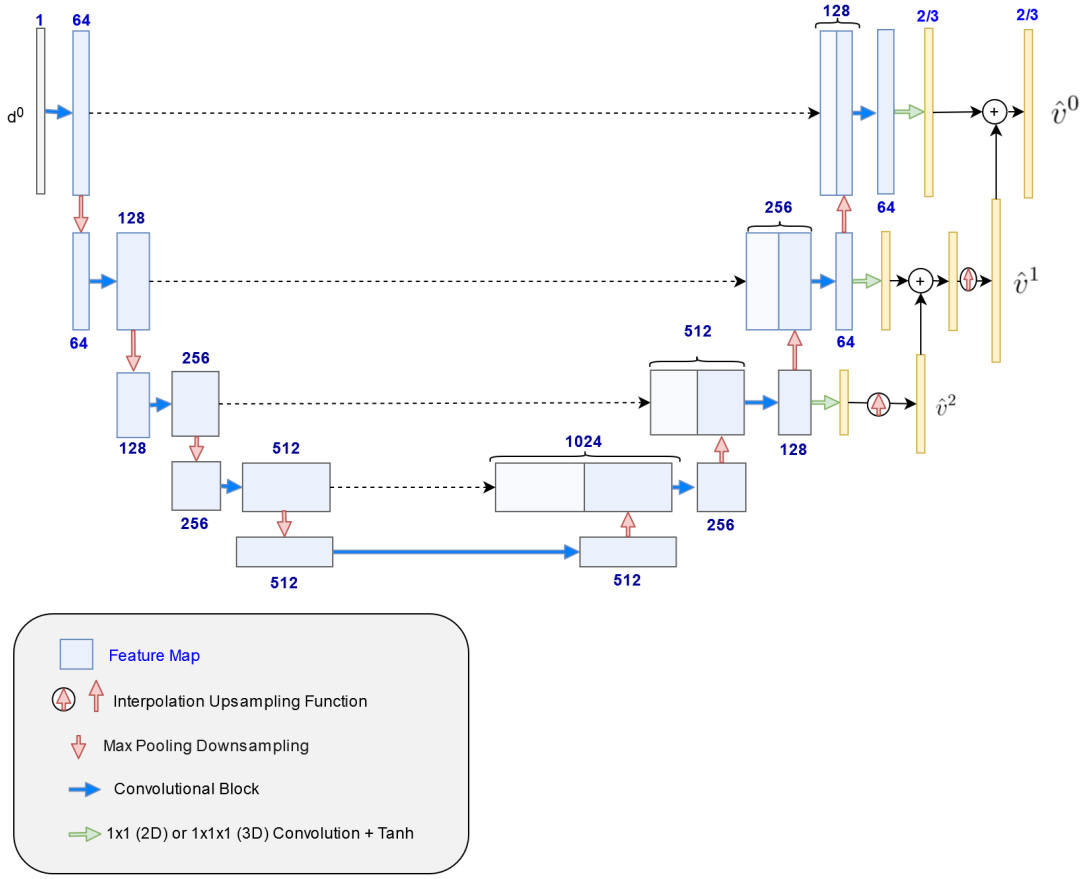


Figure 3.5.: *Customized U-Net model for the case of 3 residual velocities (U-Net$_{+3}$). Width of the boxes represent the number of channels whereas the height of the boxes represents spatial resolution.*

The implementation allows the customization of convolutional block and normalization type. In particular, it allows choosing between single convolution block, double convolution block (original U-Net), and dense block.

For training, we use Adam optimizer with 0.001 learning rate and gradient clipping. We have found good results by setting the weight of the style loss to a high value: $1e17$.

3.3. Patch Based Approach

Machine learning libraries such as PyTorch are heavily optimized to run on GPUs and perform computation in parallel. However, when performing stylization the neuronal network is required to store input data, weights for all the convectional layers and parameters of the trained model and multiple intermediate results in feature space. All of this can be high memory consuming especially when dealing with 3D data, and since GPUs memory is more scarce than host RAM, it results in a heavy limitation of maximum density resolution that can be processed in one pass.

A single-patch based method is impractical and too limited to be used in production, for this reason, a simple method based on overlaps that enable to process density fields of arbitrary resolution is proposed and studied.

First, the full-resolution density field of size $D_g \times H_g \times W_g$ is divided into of regular patches of same size P^3 (3D) or P^2 (2D). Regular patches can be conveniently stored in row-major-order in the batch dimension of a tensor, of shape [B,1,D,H,W] for 3D and [B,1,H,W] for 2D, instead of having to use another data structure. Where D is the depth, H the height, and W the width of a patch. Since we extract regular patches $D = H = W = P$. Patches can be overlapped, in this way, we can both introduce spatial coherence and allow for interpolation at the overlapped region to stitch the patches and reconstruct the full resolution stylizing velocity field $\hat{v}$. Whilst the strength of this approach is its simplicity, increasing overlap size O makes the number of patches needed to be processed to increase and reduce global efficiency and speed. The number n of patches necessary to cover the volume is shown in eq. 3.7. In section 3.3.2 we propose an alternative solution based on smoothing at feature space which does not require overlap.

$$\begin{aligned} n &= n_x + n_y + n_z \\ n_x &= ceil((W_g - O)/(P - O)) \\ n_y &= ceil((H_g - O)/(P - O)) \\ n_z &= ceil((D_g - O)/(P - O)) \end{aligned} \tag{3.7}$$

Once the full-resolution velocity field has been found advection is directly performed at full-resolution to obtain the stylized density field. Advection has a much smaller memory footprint at test-time (note that at train time it needs to store gradients) so unless the density field is huge there is no risk of running out of memory.

Furthermore, it is difficult to cover exactly the input domain with a patch-size and overlap configuration. The input density field needs to be padded p_d voxels (3D) or pixels (2D) at each dimension as in equation 3.8. In addition, in order to avoid exposing further border artifacts,

more padding can be added to avoid situations where the border of a patch is aligned or too near with the border of the domain. This happens because the patches at the boundary of the domain do not include any stitching/interpolation on one of the sides and the border effects are more apparent. After the padded density field has been processed the padding is cut off from the output.

$$\begin{aligned} p_x &= n_x \cdot P - W_g \\ p_y &= n_y \cdot P - H_g \\ p_z &= n_z \cdot P - D_g \end{aligned} \tag{3.8}$$

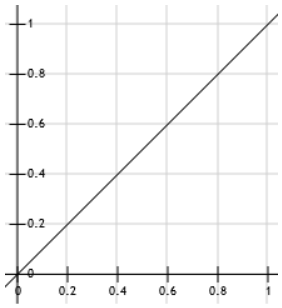
The divided patches are initially at CPU. The implementation uses an asynchronous approach to transfer data to the GPU and which allows for overlapping of computation and transfers. One thread pre-fetches a window of patches from pinned-memory into GPU and keep it on a queue, whereas the main thread issues processing operations with patches taken from the queue. Finally, another thread moves the results back to CPU so that GPU memory is freed and leaves more room for new patches. After all of the output patches have been retrieved then interpolation can be used to stitch them.

For the best performance and quality results, the highest patch-size that can be fitted into GPU should be used at a production stage. On one hand, this ensures most of the GPU computing power is being used. On the other hand, it increases the size of the receptive field of the network decreasing chances of artifacts that appear due to lack of context information which can make neighboring patches to be more incoherent. Besides, it also reduces the total number of overlapping regions that are susceptible to break due to border artifacts because the density field can be covered with less amount of patches. Note that GPU occupancy can also be increased with smaller patches sizes by batching multiple ones together into the neuronal network module however a bigger patch size with a batch-size of 1 is still preferred because of more efficiency due to reduced redundancy of overlaps.

3.3.1. Overlap post-processing: velocity interpolation

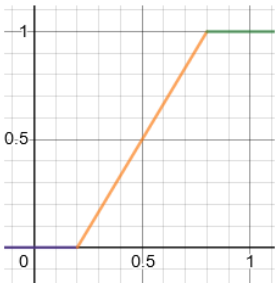
Regions with overlap where there are velocities from two patches can be blend together using a weighting function $f(x)$ for interpolation along the axis of the neighbors: the function will favor using information from the closest patch.

The weighting function can be for example a simple linear function $f(x) = x$. Furthermore, the weighting function can be directly modified as a piecewise function $f_c(x)$ to account for cropping, see figure 3.7. Cropping can be used to completely discard information nearest to the border that could contain border artifacts that can occur for example depending on the padding scheme used. Alternatively, the function can be based on a hyperbolic tangent that smoothly gives less importance to positions near the border (figure 3.8). Having different functions to choose from gives more flexibility to fine-tune interpolation at test-time for a particular style and density dataset.



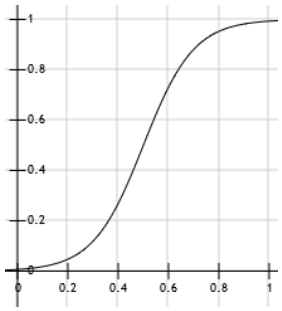
$$f(x) = x \tag{3.9}$$

Figure 3.6.: *Linear weighting function*



$$f_c(x) = \begin{cases} 0 & x \leq c \\ f(x-c)/(1-c) & c \leq x \leq 1-c \\ 1 & x \geq 1-c \end{cases} \tag{3.10}$$

Figure 3.7.: *Picewise weighting function*



$$f(x) = \frac{-tanh(-x\alpha + \frac{\alpha}{2})}{2} + 0.5 \tag{3.11}$$

Figure 3.8.: *Interpolation function based on Tanh*

The blended velocity value at a local position x, y, z on the overlap region of size O between two neighboring patches along x axis is:

$$\hat{v}_{blend_x}(x,y,z) = \hat{v}_{x0}(x,y,z)(1 - f(x/O)) + \hat{v}_{x1}(x,y,z)f(x)$$

Where v_{x0} is the velocity from the first patch at the overlapping region, and v_{x1} the velocity from the second patch - see figure 3.9 for a 2D visual representation. And similarly for the other two axes:

$$\hat{v}_{blend_y}(x,y,z) = \hat{v}_{y0}(x,y,z)(1 - f(y/O)) + \hat{v}_{y1}(x,y,z)f(y/O)$$

$$\hat{v}_{blend_z}(x,y,z) = \hat{v}_{z0}(x,y,z)(1 - f(z/O)) + \hat{v}_{z1}(x,y,z)f(z/O)$$

For simplicity we do not treat the special case of corners which would involve 4 additional diagonal neighbors.

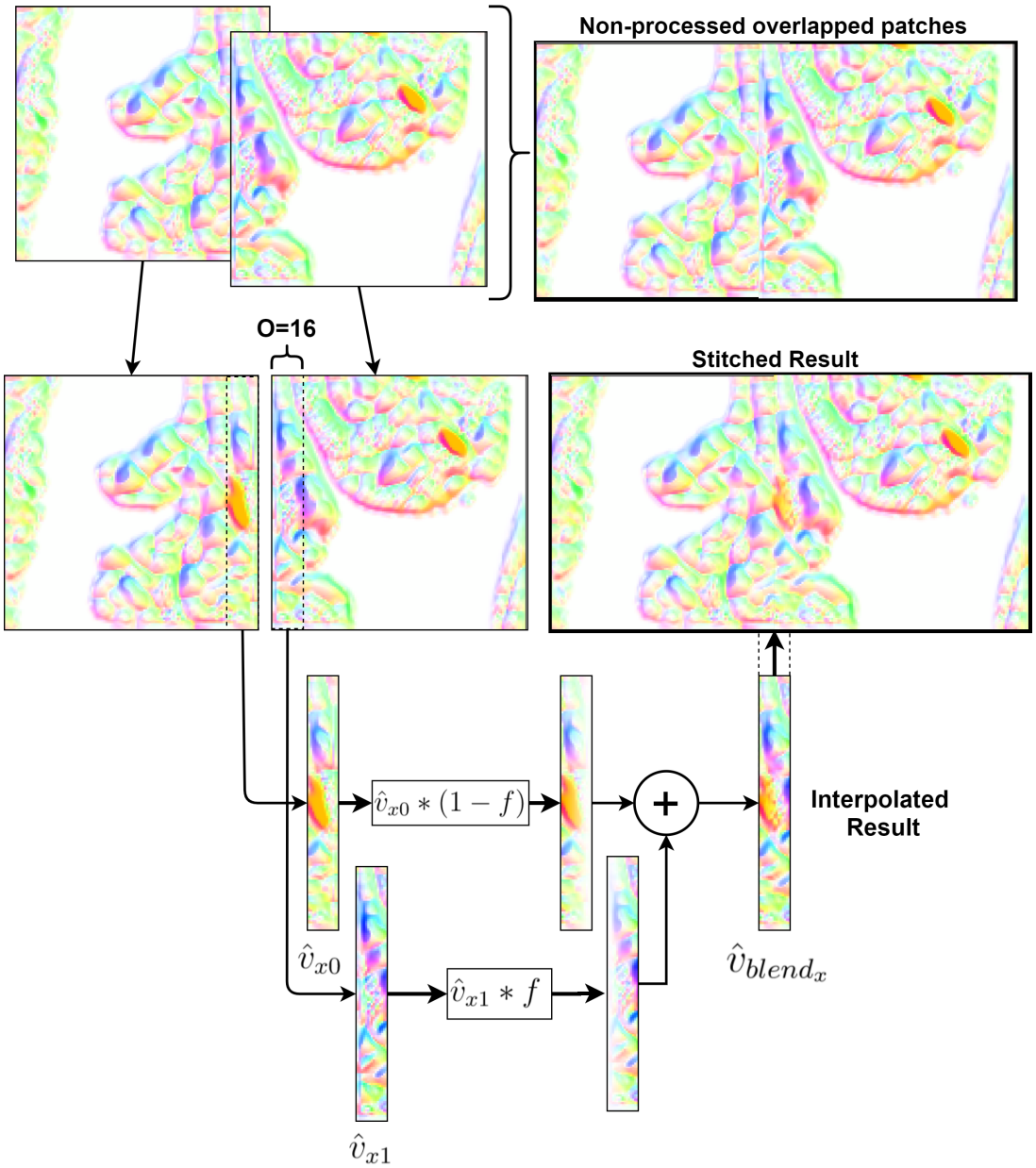


Figure 3.9.: *Patch overlap velocity interpolation for one of the axis on two dimensions. Example for patch-size 128^2 and 16 pixel overlap corresponding to "Peace" style used in the results section figure 4.7*

3.3.2. Stitching at feature space

A more advanced approach, compared to only interpolating the final output of the neuronal network, is to instead interpolate feature responses after different layers of the network: since interpolation is placed between convolution layers the stitching increases its non-linearity, allowing for more complicated variations to connect neighboring velocity fields.

Note that when using an auto-encoder with skip-connections it is not enough with interpolating at the latent space output of the encoder because the final decoded values will also depend

on the information that comes from the additional skip-connections. One option is to store intermediate results after each stage of the decoder, and process data in multiple-passes each one which computes the next stage of the decoder for each one of the patches. However, storing all the features requires a large amount of memory that can even exceed available host RAM. This makes the approach memory bound and poorly scalable, for large simulations if scratch storage is required then speed is decreased considerably. When using a network based on dense-blocks with less feature maps the memory requirements are decreased.

A more efficient approach consists of dividing the full-resolution field into cells that form a checkerboard and doing two processing passes. The following approach is compatible with non-overlapped tiling scheme:

- The first pass processes patches corresponding to white cells and stores the borders of size one in feature space at different stages of the decoder (figure 3.10). The center of feature cells can be discarded because they are only required for the computation of the final velocity field of that particular patch: this is a huge memory gain compared with the multi-pass idea that stores all intermediate results for feature interpolation.
- The second pass processes patches corresponding to yet unprocessed densities in the black cells. In this pass, the network computation graph can be modified to allow for interpolation at each stage of the decoder. When processing each patch the four (on 2D) or six (on 3D) neighbor borders are accessed and used to replace values that are near the border with weighting functions - see figure 3.11.

3.4. Temporal Coherence

The stylization of a sequence of frames can be obtained by individually processing the density of each frame with the convolutional neuronal network and the patch-based approach. However, changes in the input density field that happens from frame to frame can cause the network to vary its output and consequently make smoke style patterns to abruptly vary their shape or appear and disappear.

If a particular trained model exhibits these problems (as seen in results section depending on the style image our network can already produce temporally coherence results so no further action is not required), temporal coherence can be enforced at a post-processing phase, after the stylizing velocities $\hat{v}^0, \hat{v}^1, ...$ corresponding to each frame have already been obtained. The velocity at a frame can be temporally smoothed by looking at velocities in a given a frame neighborhood of size $W = 2w + 1$, following an approach similar to [KAGS19].

The method requires the velocities from the original smoke simulation $U = \{u^0, u^1, ...\}$ each one that can take the un-stylized density at ith frame to frame $i + 1$: $d^{i+1} = \mathcal{T}(d^i, u^i)$. The method uses U to transport velocity fields instead of densities from one frame to another: $\mathcal{T}_i^j(\hat{v}_i, U)$ is defined as a function that can transport an stylizing velocity from frame i to j by recursively doing advection with the simulation velocities. When the destination frame comes before the origin frame (j < i) the advection is performed backwards by negating the simulation velocities. For example, equation 3.12 shows how to transport a velocity field two frames forward and equation 3.13 two frames backwards.

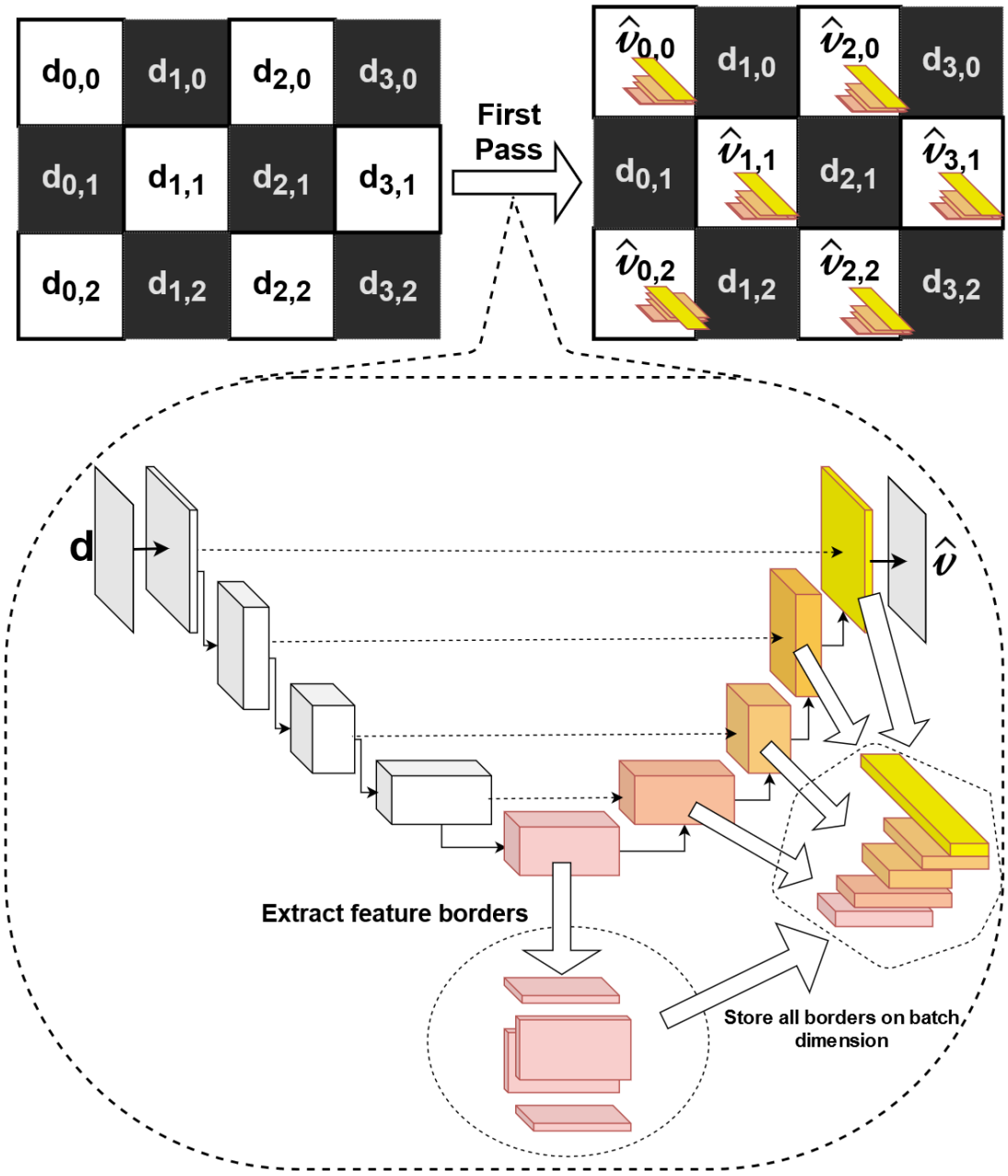


Figure 3.10.: *Simplified view of how feature responses borders are extracted at each depth of a 2D decoder. 3D colored boxes represent features after they have been concatenated with the skip-connection and processed by the convolution block.*

$$\mathcal{T}_0^2(\hat{v}_0, U) = \mathcal{T}(\mathcal{T}(\hat{v}_0, u_0), u_1) \tag{3.12}$$

$$\mathcal{T}_2^0(\hat{v}_2, U) = \mathcal{T}(\mathcal{T}(\hat{v}_2, -u_1), -u_0) \tag{3.13}$$

The velocities in a window centered at i=t, can be aligned together by forward-advecting w previous frames neighbors with indexes i < t and backwards-advecting next w frames with

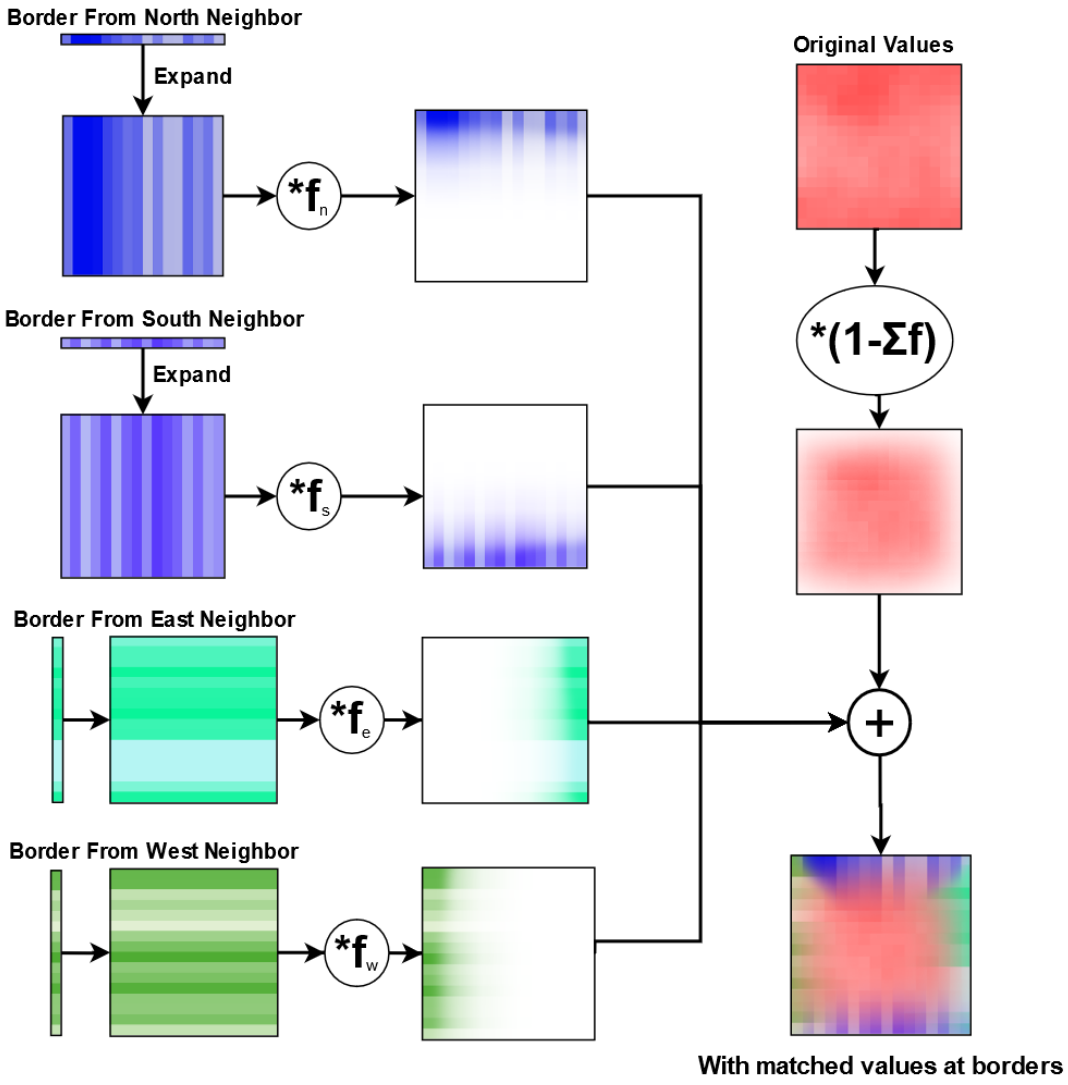


Figure 3.11.: *Illustration of an strategy that smoothly transitions to match values at borders with weighting functions on 2D.*

indexes i > t. Then the velocities are combined together into the smoothed styling velocity $\hat{v}^*$ by using some weights w_i: see equation 3.14.

$$\hat{v}_t^* = \sum_{i=t-w}^{t+w} w_i \mathcal{T}_i^t(\hat{v}_i, U); \tag{3.14}$$

The weights are set in order to compute the mean, which in non-special cases is $w_i = 1/(2w + 1)$. But note that when doing smoothing and the center frame t is at the beginning $t < w - 1$ or end of the sequence $t > F - w$ the smoothing take into account less velocities because of the missing frames before or next to t.

Whilst [KAGS19] apply temporal smoothing as part of every step of their iterative optimization, the method only needs one pass over the sequence with our work. Whats more, the window size W at [KAGS19] is more bounded by memory because their temporal coherence enforcement is

coupled with optimization: computed intermediate velocity fields on equation 3.14 need to be stored together in order to back-propagate gradients.

3.5. Masking

Some of the style's results have shown artifacts where the network learns an stylization that deforms the original smoke shape excessively and/or halo artifact's appear.

Artifacts can be reduced by constraining velocities to only into areas where is density by using a mask $m = clamp(a * g(d), 0.0, 1.0)$ for which a Gaussian g is optionally used to have soft edges and a coefficient a can be used so that velocities on regions with density at least $\frac{1}{a}$ are left completely unchanged. The mask can be applied to the stylizing velocities either at test-time $\hat{v}_{masked} = \hat{v} * m$ or as an additional loss so that the network already learn to produce masked velocities and no additional post-processing is required (eq. 3.15). However, post-processing mask has the advantage that can be more easily fine-tuned to the particular density field we want to process and training is easier.

$$\mathcal{L}_m = \frac{1}{DHW} \sum_{i}^{D,H,W} \sum_{c}^{C} (e^{(1-m_i)*|\hat{v}_{i,c}|} - 1) \tag{3.15}$$

3.6. Training Data

Patch-based approach benefits improves generalization but does not mean is inmune to overfitting. For this reason we have taking care of generating a diverse dataset combined with data augmentation.

3.6.1. 3D dataset

We follow a similar procedure to FluidNet paper to generate synthetic data for training: the velocity field is initialized as a random turbulent field, geometry is randomly placed as obstacles or density inflow, localized input perturbations are added in a procedural and randomized way. Each simulation includes randomly chosen gravity, density, and vorticity confinement of varying strengths.

We have reused FluidNet dataset generation source code with some modifications. Instead of always using 3D models as obstacles, sometime we use the 3D model's shape to initialize an inflow density with the same shape. We use a subjset of 1000 NTU 3D Model Database. Each simulation performs 128 solver steps and stores 64 frames of the sequence (every two steps density is saved). We compute our simulations different higher voxel resolutions: In particular we compute:

- 320 sequences at 64 voxel resolution

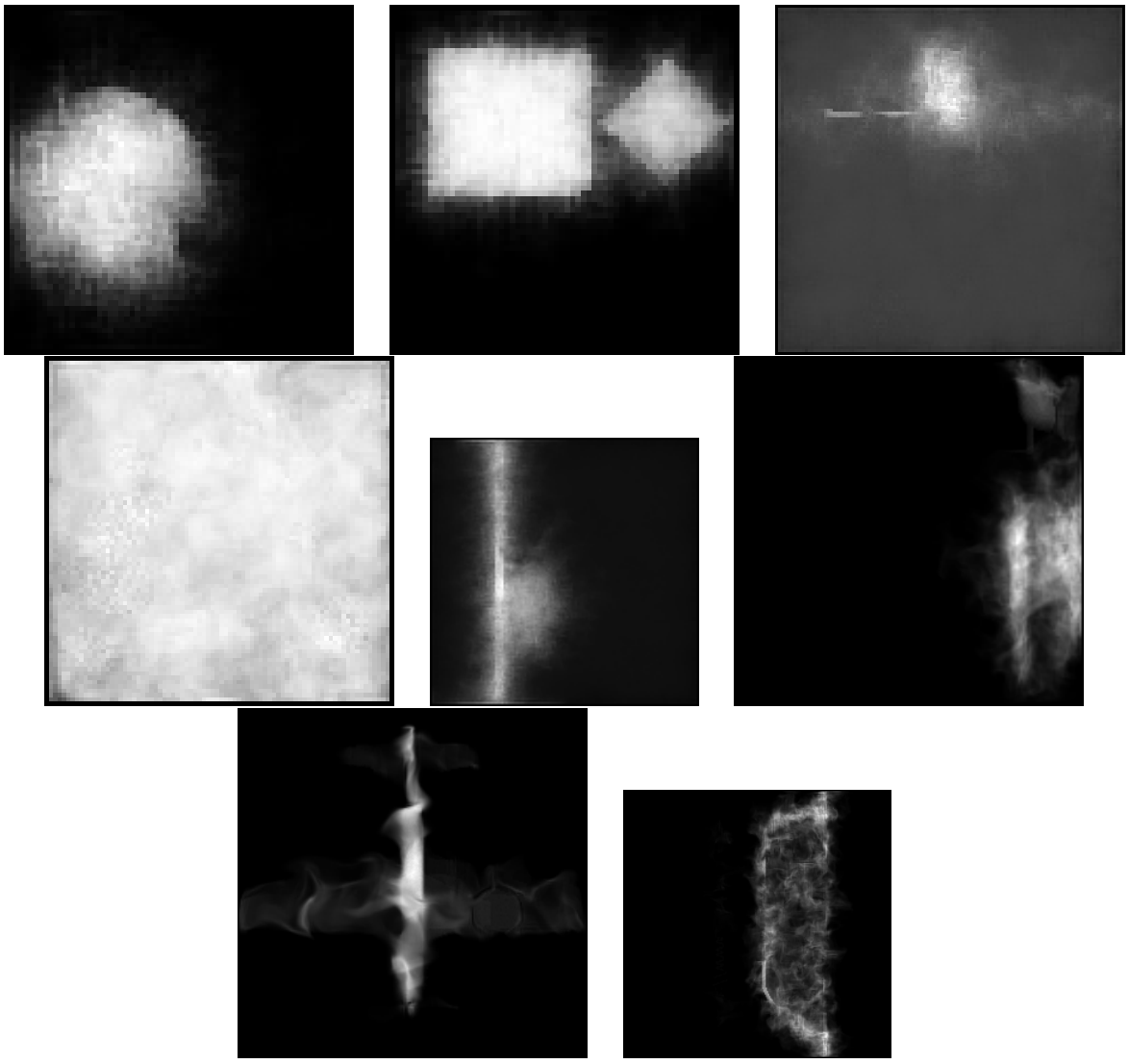


Figure 3.12.: *Sample frames from the 3D dataset*

- 160 sequences at 80 voxel resolution
- 40 sequences at 128 voxel resolution
- 20 sequences at 160 voxel resolution
- 10 sequences at 192 voxel resolution
- 10 sequences at 225 voxel resolution

Simulations at higher voxel resolution are more costly so we reduce the number of simulations. However there is also more space available for patch extraction.

We combine synthetic data generation with augmentation both at simulation resolution and patch resolution. We apply random re-scaling into $[P_e^3, 300^3]$ voxels to each simulation smoke sample and make sure density values are normalized in [0-1] range. And after that we sample multiple random patches that are used for training, those patches are also augmented with rotations and random turbulence by advecting the patches with random velocity. $P_e = P * \sqrt{2}$

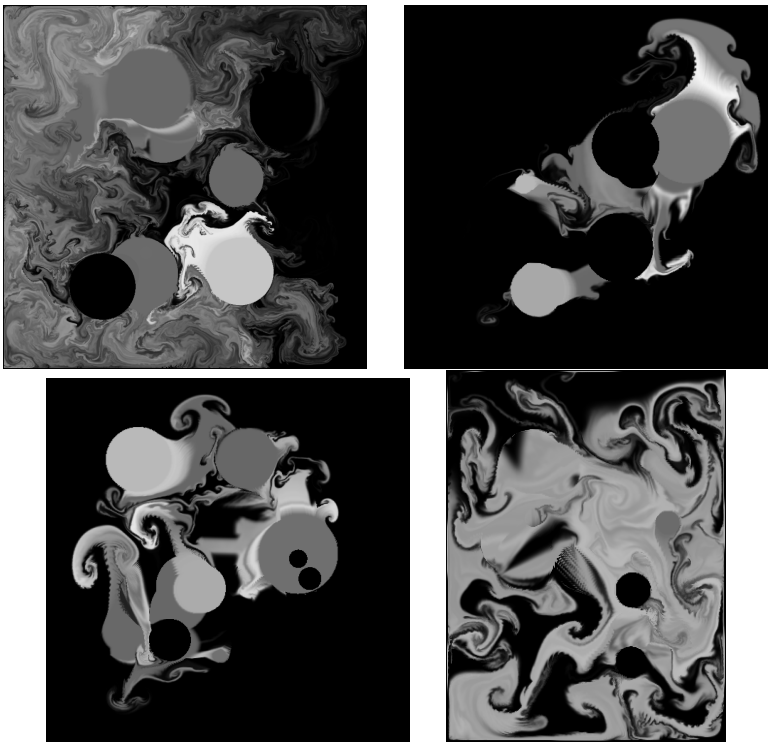


Figure 3.13.: *Sample frames from the 2D dataset*

is the patch extraction size and is slightly bigger than training patch size P so that there are no discontinuities after rotation used in augmentation, after augmentation the patch is cropped into a size of P^3. The number of patches extracted from each $[P_e^3, 300^3]$ voxels smoke varies with the smoke resolution: we extract more patches from higher resolution smokes.

3.6.2. 2D dataset

The 2D dataset follows a similar approach to the 3D dataset , the main difference being that the simulations can be done at higher resolutions. In particular there are:

- 75 sequences at 768x768 pixel resolution
- 100 sequences at 640x640 pixel resolution
- 125 sequences at 512x512 pixel resolution
- 250 sequences at 384x288 pixel resolution

To generate the 2D dataset, on each simulation a random gravity (direction and maginutude) and vorticy confinement strength is chosen. A simplex noise generator is used to initialize the density field inside one of the randomly placed silhouettes from the MPEG-7 Core Experiment CE-Shape-1 dataset (contains 70 shape classes, and for each class there are different silhouettes to provide intra-class variance). Randomly sized spherical objects are positioned on the domain: they can act either as density sources, velocity impulse generators, or obstacles. The ammount of density generated each frame by the sources varies according to a randomized wave shape

and can have decay. After simulation step 50 frames start being saved every two steps.

As part of augmentation, at each training step an image corresponding to the density of a frame in a sequence is randomly flipped. Also, after patch extraction at random positions of the frame, the patches densities can be randomly advected.

3.6.3. Patch queue

On both 2D/3D datasets, patch extraction and augmentation is performed on separate worker threads each one assigned with a different cuda stream, then the extracted patches are enqueued so that the main thread can use them for training. Each worker extract patches from multiple simulation data samples and then shuffle the patches to increase variability on local window of patches used on training.

3.7. Other approaches

This sections mentions some changes that might be done in the previously explained method and discuss their effects - some of the alternate configurations will be further discussed in the results section.

3.7.1. Architectures based on residual layers

Jhonson et. al proposed a popular network architecture [JAL16] that when used with instance-normalization [UVL17] has been shown to produce high-quality results on image style transfer. However, most of the (residual) convolution blocks on the network are at the same spatial resolution after which might cause the network to only be able to properly learn features at a single scale. Adittionally, longer encoder-decoder skip-connections are missing. A model based on this architecture has been implemented so that it can be compared with UNet architecture, the only required change to work with transport-based stylizing is to use a Tanh output activation function.

A more advanced architecture, also based on residual layers, is SRResNet, which has been used on image super-resolution. Compared to Johnson' it is comprised of more residual blocks (increased depth) but each one with fewer filters. On top of that, it has a longer range skip-connection and uses efficient sub-pixel up-sampling at the end. This architecture can be modified for transport-based stylizing by also using a Tanh output activation function and by adding two down-sampling blocks at the beginning (similarly to Johnson') so that the network can accept input at its full resolution. See the table A.7 in the appendix for more details.

3.7.2. Slice-based 3D approach

The method section 3.3 describes a simple patch based approach with we have used to create a first baseline for fast volumetric stylization. However, an alternative slice-based approach has

also been implemented with limited results

The slice based 3D approach uses a 2D neuronal network to process a 3D volume. It can be seen as an special form of patch-based approach which allow to process arbitrary depth volume overcoming memory limitations. First, an axis on which we want to process the volume is selected, then the network takes as input one of the density slices along that axis, and the previously processed stylized density slice to produce a 2D style velocity field. The individual 2D velocity slices are then stacked into a volume.

The benefits compared to patch based 3D-network, is that it can take into account stylizing across the whole density depth whereas patch-based only take into account density that can see on that patch. Moreover, the perceptual loss can be directly connected to the output 2D result instead of rendering from multiple sampled camera positions.

However, a limiting problem of this approach is that it requires to rotate the input density field, perform multiple passes along different axes, and then finally reconstruct the final 3D velocity field from the multiple 2-component velocity outputs. Limited results have been obtained by doing two passes along two orthogonal axes and averaging the y components (up-axis) of the velocity vectors: however more research is required to find a good strategy for combining more velocity fields, or even combine this approach with the regular-sized 3D patch approach.

3.7.3. Stacked Multi-net model

We tried a 2D multi-scale approach where we stack multiple networks, each one takes the stylized density field of the previous network and adds stylization according to a different scale in a Gaussian pyramid. However, each new pass amplifies errors of the previous network and the final quality is undesirable. For this reason, we decided to focus the rest of the research on a single-network method.

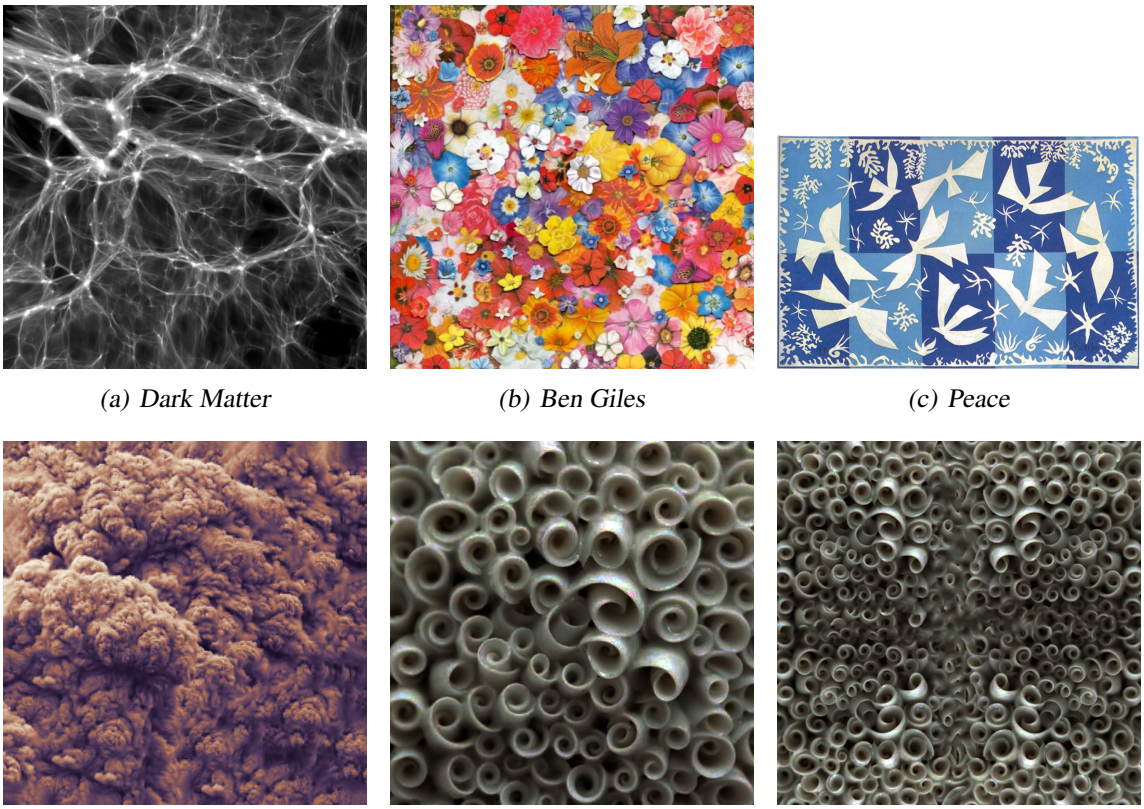
3.7.4. Multi-view 2D-encoder to 3D-decoder

An approach based on Facebook neuronal volumes has been considered [LSS+19]. The idea is to use multiple 2D encoders instead of a 3D encoder, each of the encoders would take as input one rendering of the volume from a different view. This approach would be able to encode more efficiently, especially if the weights of the multiple encoders are shared. The downsides are that skip-connections between encoder and decoder are not possible, at least without additional processing (it could be possible to reconstruct 3D features from multiple 2D features according to an orthographic camera model and then use the 3D features for the skip connection), and makes the problem harder since as shown in [LSS+19] the output of the decoder is not completely artifact-free.

A 2D autoencoder that followed their architecture has been implemented. However, we obtained unsatisfactory 2D results: the network outputs velocities lacking on high-frequency details that are not of enough quality for transport-based density stylizing. For this reason, further research has not been continued on this idea. The auto-encoder was hard to tune for satisfactory training, the use of ELU activation layers was particularly useful to avoid vanishing gradients.

4

Results



(a) Dark Matter *(b) Ben Giles* *(c) Peace*

(d) Volcano *(e) Spirals$_{1.0}$* *(f) Spirals$_{0.5}$*

Figure 4.1.: *Style images obtained from different sources*

The collection of style images used to train models in this section is on figure 4.1. Adam optimizer with gradient clipping and a style weight of 1e17 is used in all experiments. The learning rate is 0.001 for all U-Net experiments. U-Net model uses default original convolutional blocks comprised of two convolutional layers unless otherwise stated. For the loss graphs, the standard deviation is shown in a shaded area, and the main colored line represents the mean. The mean and standard deviation can be computed by considering a moving window over the data saved at different optimization steps.

4.1. 2D

The test losses are evaluated over a final full-resolution stylized density field after all patches are assembled. The three reported test-loss scales do correspond to the multi-scale style loss as described in section 2.1.1, where a three-level Gaussian pyramid is used for the density field as well as for the style image. Since the end-goal of the patch-based approach is to apply stylization at high-resolution images, evaluating test-loss at a higher resolution than patch-size is preferable. The image 4.2 is used for the test-time loss evaluation.

We follow the same approach as [JFA+15] to visualize 2D velocity fields. 2D velocity fields can be shown as a color map where the direction is represented by hue values and magnitude by saturation. To keep visualizations comparable, hue values are computed relative to the same maximum magnitude of 16, so that the same color values in two different images correspond to velocities of the same magnitude.

Some of the feed-forward results are compared with an iterative TNST method also implemented on PyTorch and which uses the same configuration of VGG layers and weights for the perceptual loss and also uses Adam optimizer.

On the practical side, on models trained with batch normalization, we have found that slightly sharper results can be obtained by allowing to keep updating mean and variance with test-data patches. However, this seems also to generate more unstable results.

All 2D models use first order advection. We did not found significant differences by using second order advection but results look slightly more sharp on first order advection and more smooth with second order.

4.1.1. Single-scale U-Net stylization

Using a patch of size 128^2 for training, after only 2-3 hours of training, 50k optimization steps, on a single high-end GPU the model is already able to produce stylizations of good quality. After 3-5 hours of training, and 75k optimization steps performed, is difficult to see any additional visual improvements by doing more iterations. Although it is possible to train with smaller patch sizes, both train and the test-loss show worse values in that case (see figure 4.3 for 80^2 patch-size). All 2D U-Net models used on the the following experiments use a convolution block comprised of two convolutional layers with reflection padding.

Instance normalization on previous works has shown dramatic improvement in image-based

Figure 4.2.: *Un-stylized test image used to evaluate the different models in the experiments.*

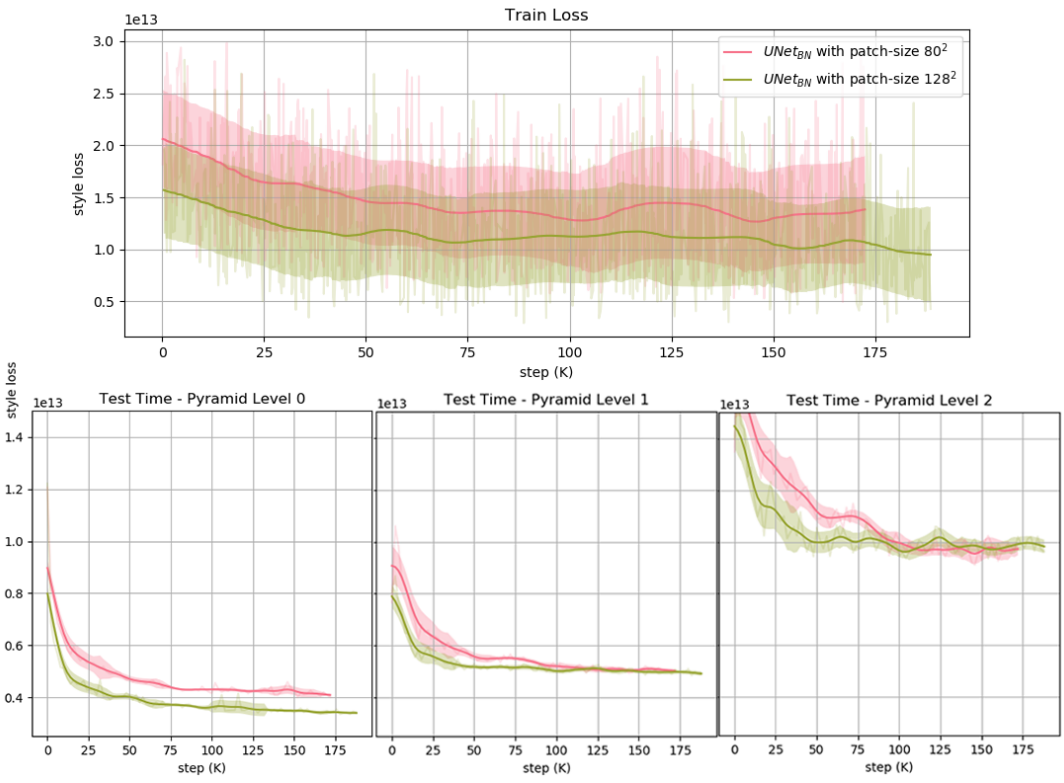
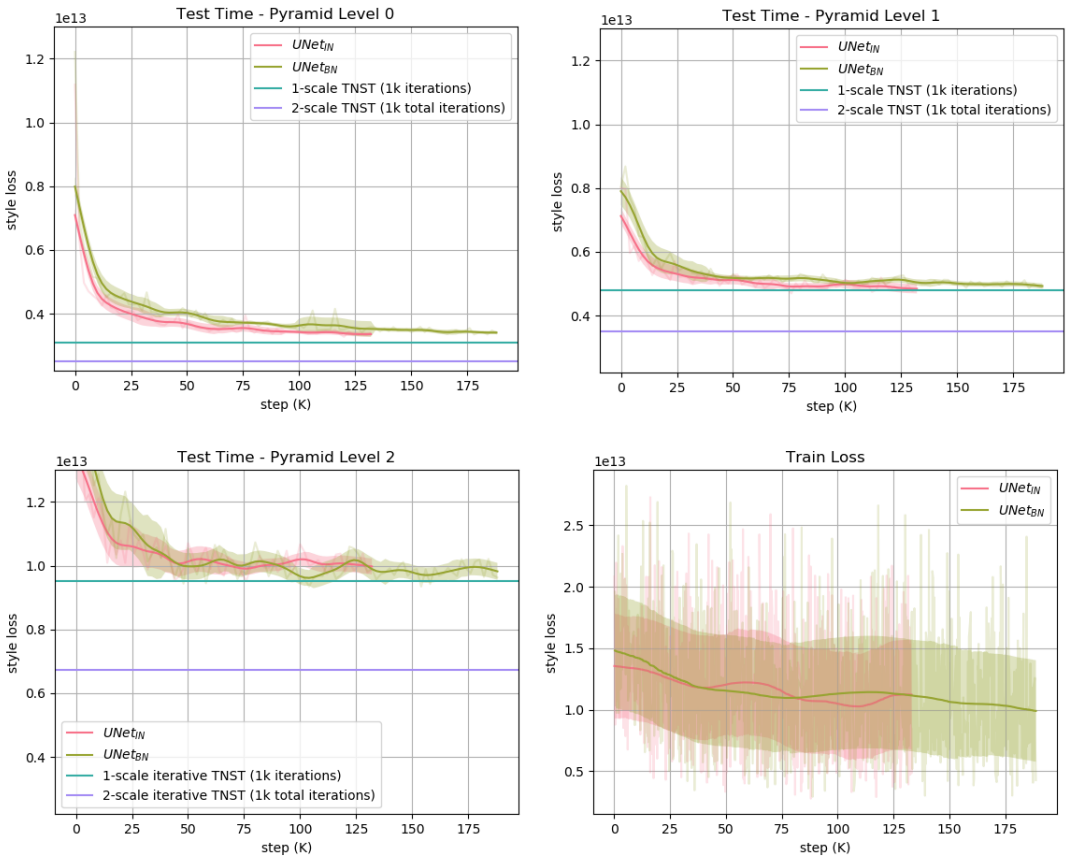


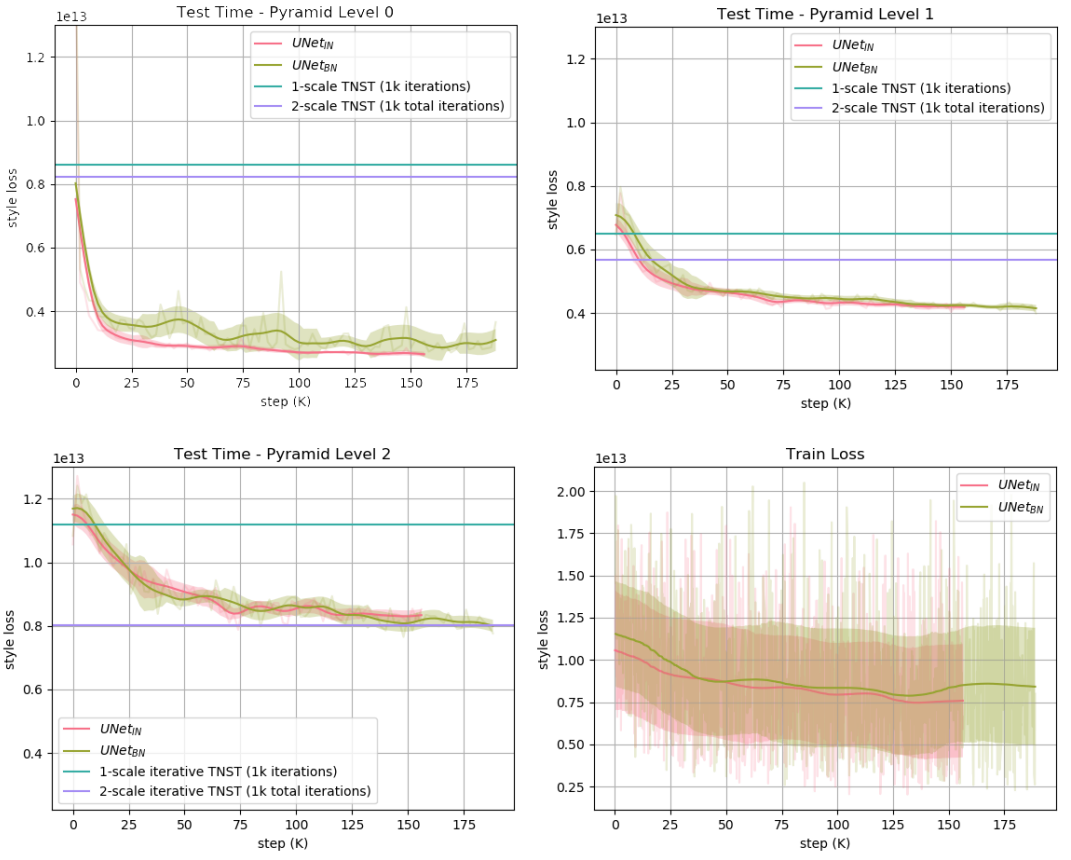
Figure 4.3.: *Train and test loss for different patch sizes. A Tanh factor of 0.128 is used for the model with 80^2 patch size, and a factor of 0.08 for 128^2 patch size. Both models use batch normalization.*

stylizing methods. However, in our results for the U-Net architecture, we only found a slight improvement in style loss (fig. 4.3) and not any significant visual changes. Out of 4 style images tested, instance normalization has shown a slightly better test-time loss on three of the styles (Ben Giles, Dark Matter, and Peace). A detailed description of the 2D architecture used in the experiment with instance normalization can be found in appendix table A.2.

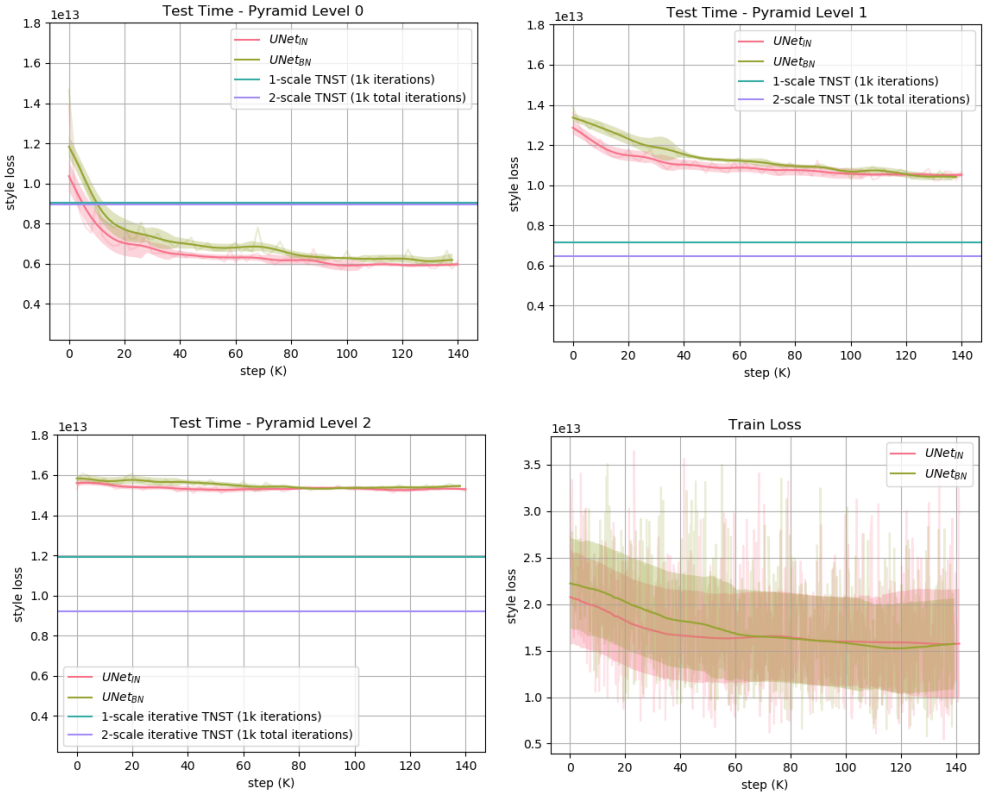
Compared to iterative TNST, on Wave and Peace the feed-forward approach is able to have a lower test-time loss on the first level of the pyramid than the iterative TNST method (fig. 4.4(d), and 4.4(c)), however, performs worse in next-levels: This indicates that the feed-forward approach perform better on smaller-scale shapes but worse on bigger patterns. Surprisingly, the feed-forward approach can outperform TNST test-loss on the Dark Matter style on two levels and similar loss on the third level (fig. 4.4(b)): this again can indicate that the CNN is more suited for style images with textures that have smaller details. In contrast, the CNN is not able to surpass TNST on any level for Ben Giles which is an image with a lot of medium-sized flower objects (fig. 4.4(a)). Note also that the TNST configuration was set to the same parameters as the feed-forward approach to enable a reasonable comparison, however, it might be possible to obtain better results by fine-tuning the TNST approach: for example, [KAGS19] TensorFlow implementation uses more pyramid levels with smaller resizing factor, Inception network instead of VGG-16 and Laplacian gradient descent. Figures 4.4, 4.5, 4.14 and 4.7 show a side by side comparison of TNST and feed-forward methods.



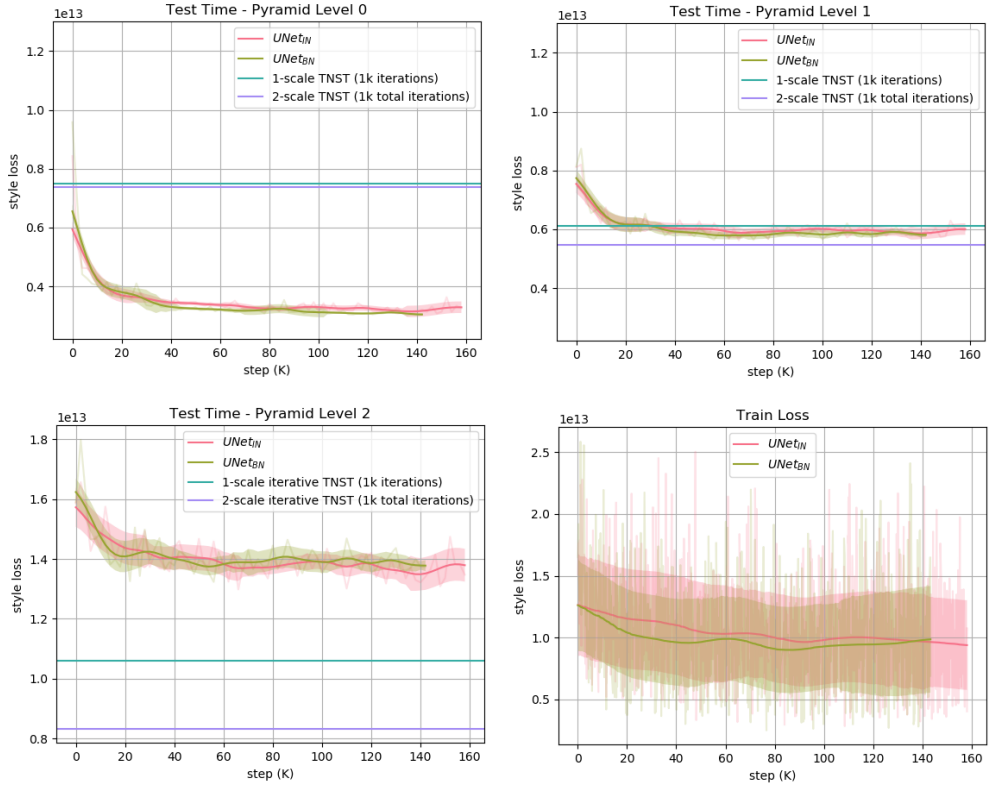
(a) Ben Giles Losses



(b) Dark Matter Losses

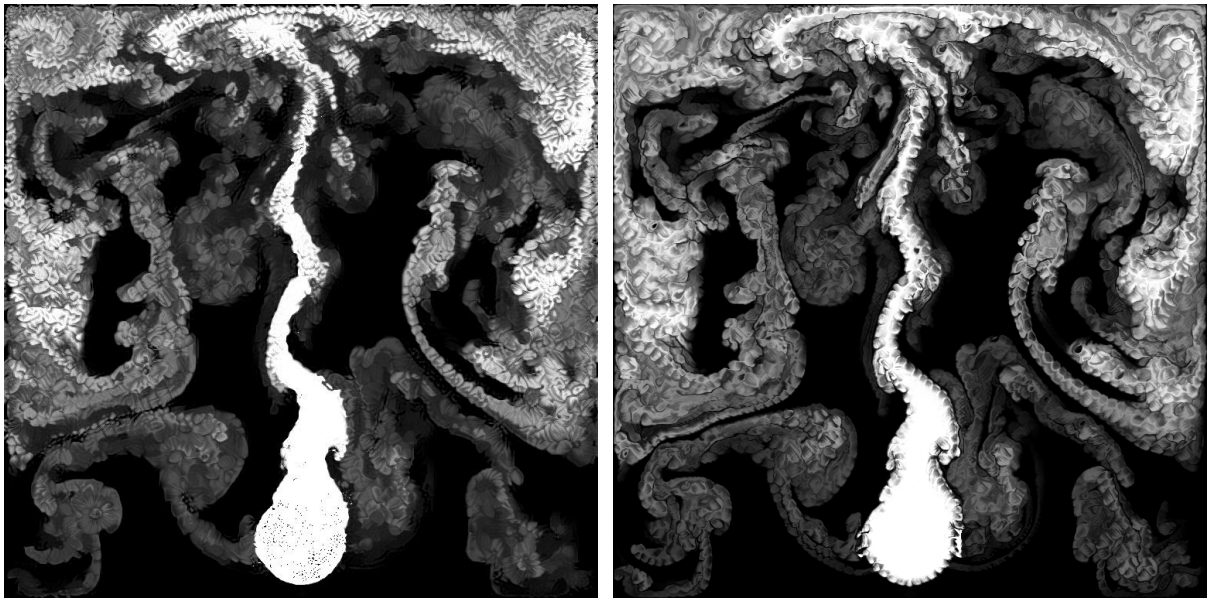


(c) Peace Losses



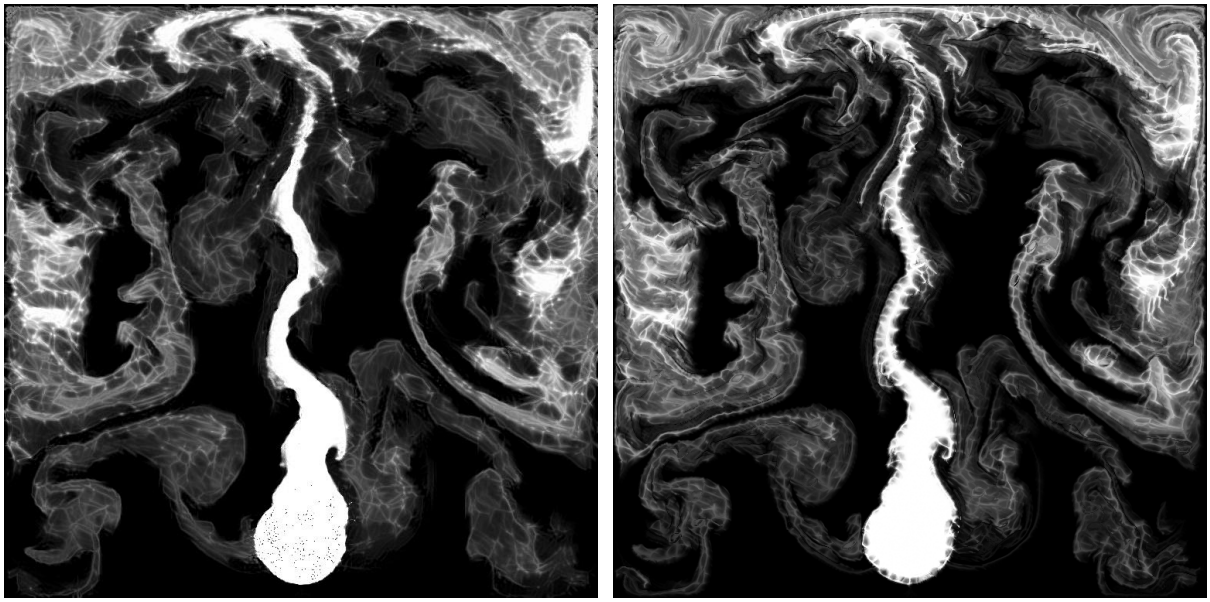
(d) Wave Losses

Figure 4.3.: *Single-scale loss U-Net feed-forward approaches compared to iterative TNST. U-Net$_{IN}$ (shown in red) uses instance normalization and U-Net$_{BN}$ (shown in green) uses batch normalization with batch size 4.*



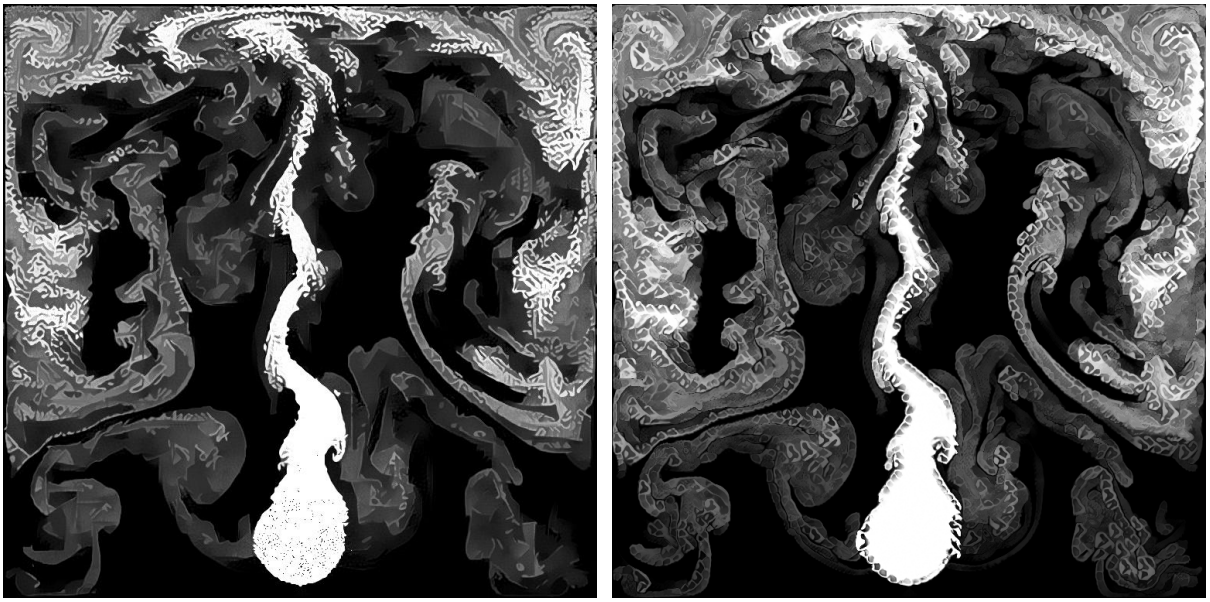
(e) Iterative TNST (1k iterations) (f) Feed-forward

Figure 4.4.: *U-Net feed-forward with instance normalization compared to iterative TNST for single-scale loss Ben Giles style. 16 pixels overlap and velocity interpolation is used in the CNN result.*



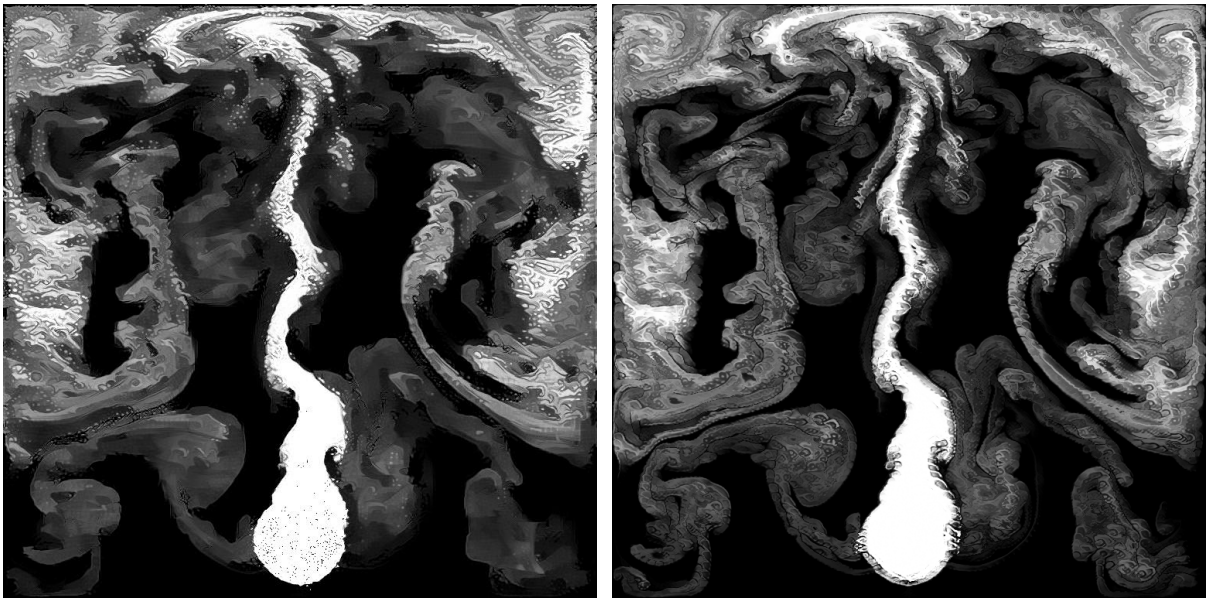
(a) Iterative TNST (1k iterations) (b) Feed-forward

Figure 4.5.: *U-Net feed-forward with instance normalization compared to iterative TNST for single-scale loss Dark Matter style. 16 pixels overlap and velocity interpolation is used in the CNN result.*



(a) Iterative TNST (1k iterations) (b) Feed-forward

Figure 4.6.: *U-Net feed-forward with instance normalization compared to iterative TNST for single-scale loss Peace style. 16 pixels overlap and velocity interpolation is used in the CNN result.*



(a) Iterative TNST (1k iterations) (b) Feed-forward

Figure 4.7.: *U-Net feed-forward with instance normalization compared to iterative TNST for single-scale loss Wave style. 16 pixels overlap and velocity interpolation is used in the CNN result.*

Patch Stitching

Fig 4.8 shows that the patch-based approach can be applied to a high-resolution 768x768 image in three different ways: without overlap, with overlap but no interpolation, and combining overlap and interpolation. Only adding overlap, (and cropping overlapped regions), solves most of the incoherence between neighboring patches, interpolation then can be successfully applied to smooth the remaining smaller incoherences. At least 16-pixel overlap between patches needs to be used to achieve smooth transitions.

Velocity Mask

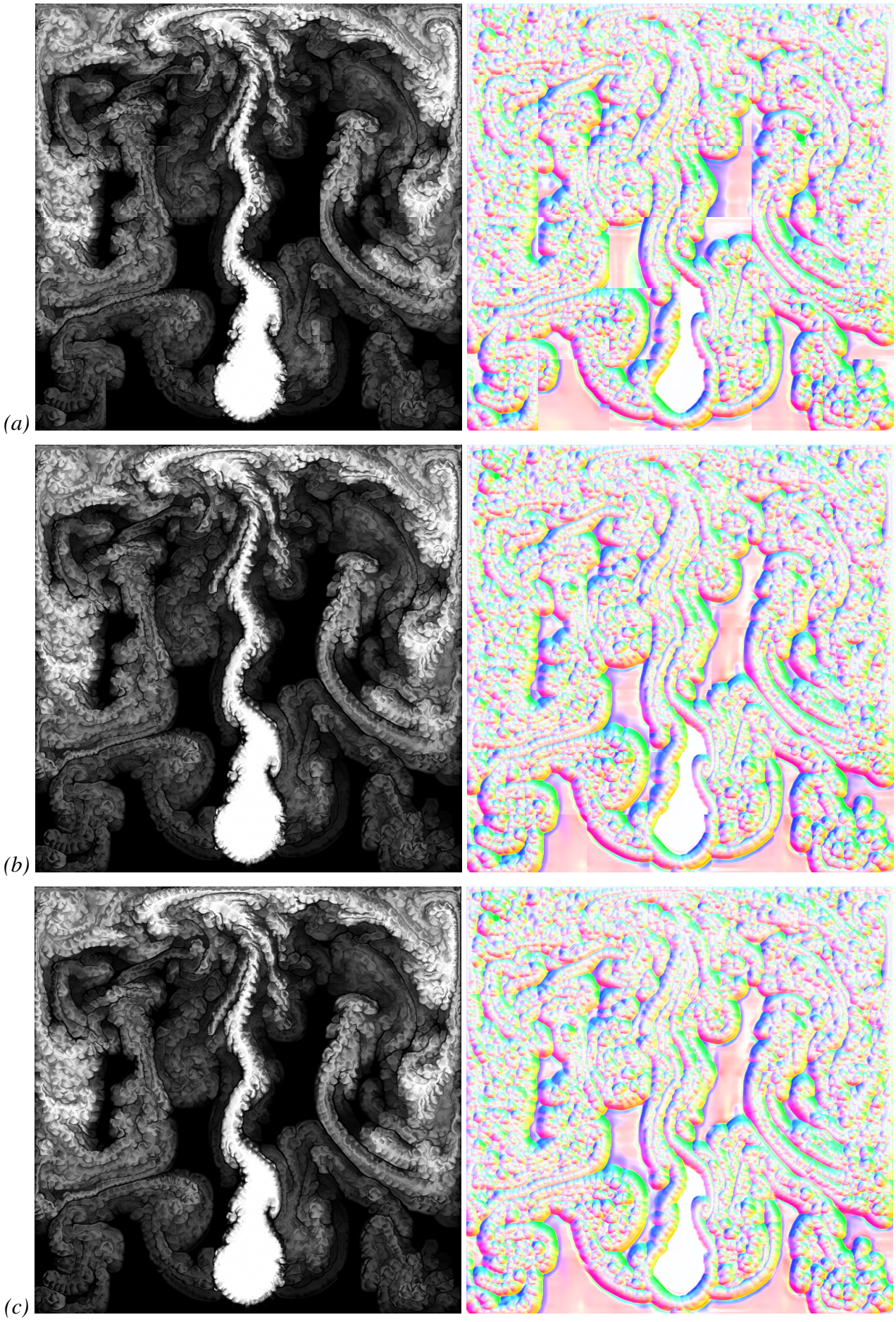
Figure 4.9 shows that the model stylizes more aggressively when there is no velocity mask loss. However, it also expands the original shape of the smoke excessively. A model with two-scale losses and with velocity mask 4.14(b) is able to showcase similar similar style patterns without compromising to deform the original shape of the smoke.

4.1.2. Multi-scale U-Net stylization

For the multi-scale loss applied a patch during training, better results with more details and decreased style loss are found by not using a Gaussian pyramid for the density and only using the Gaussian pyramid in the style image (as in figure 3.2 from method section). A reason for this behavior is that since the resolution of the patch is already limited, further down-scaling decreases the ability of the classifier to produce features of sufficient quality due to the lack of resolution. The idea of [Sne17] of using a Gaussian pyramid was originally intended for high-resolution images. The Gaussian pyramid is only used on the density patches for figure 4.10, all the other experiments only use the pyramid for the style image.

For some styles, a slight improvement on higher scales of the test-time pyramid loss can be found when using a model U-Net$_{+2}$ with 2-level multi-scale loss, with residual velocities, batch normalization, and not using a Gaussian pyramid for density patches. In this case, the results show slightly more aggressive pattern creation (especially for Ben Giles). Without residual velocities, the results and losses are similar to a single-scale version. However, the style loss at level zero can show worse values and the results are noisier. Image-based feed-forward stylizing methods have been able to take multiple-scales into account with more drastic results, however directly producing pixel values is easier than finding a stylizing velocity field.

An explanation of why using a U-Net$_{+}$ model based on the residual-velocities results in more aggressive stylizing, might be reasoned due to the fact that the model can allow for longer range density transformations. The version that outputs a single velocity field is configured with a re-scaled Thanh so that the maximum magnitude is 14.48 ($\sqrt{2(0.08 * 128)^2}$). In the same way, each one of the residual velocities in the residual U-Net$_{+}$ approach that can also be in the range of 14.48, however since residual velocities add up, the maximum allowed magnitude of the velocity field is larger: for two scales it would be 28.96. Trying to further increase maximum allowed magnitude on a single-scale velocity field U-Net approach leads to worse loss values. Using a residual approach with a single-scale loss is also unsatisfactory because



Stylized density and velocity for: (a) no-overlap, (b) overlap and crop, (c) overlap and interpolation. The model used is U-Net with single-scale style loss and reflection padding.

Figure 4.8.: *Ben Giles 2D Stitching*

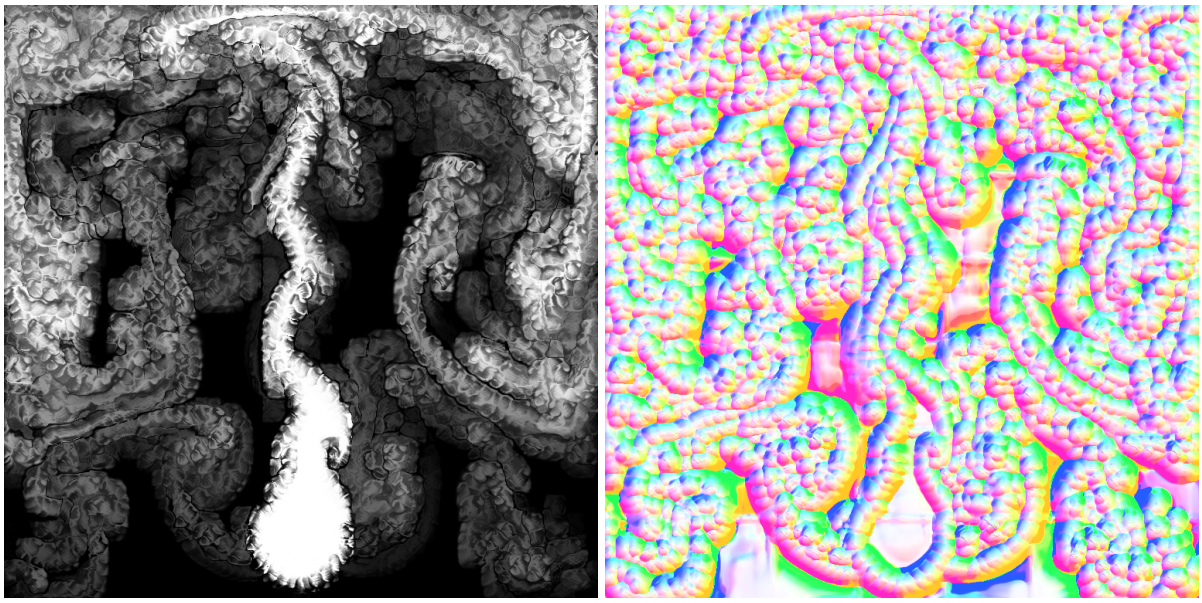


Figure 4.9.: *Results for un-masked velocity for a single-scale model for Ben Giles*

lower-resolution residual velocities are mostly ignored by the network.

Even though that by default the 2-scale results are very similar to 1-scale results, the two-scale residual has a very powerful feature: It is possible to further control the stylizing result by modifying the re-scaling factor of the Tanh activation functions at test-time. Decreasing the factor on lower-resolution residuals removes bigger patterns and leaves intact the smaller high-frequency details (see figures 4.11(a), 4.11(b)). Similarly, decreasing the factor on higher-resolution residuals allow to obtain smoother bigger patterns (see figures 4.11(c), 4.11(d)). It is also possible to boost the creation of bigger patterns 4.12(a) by increasing the scale of the lower-resolution residual, although it is also possible to increase the magnitude of velocities from 1-scale stylization 4.12(b) it adds excessive noise. The flexibility that the residual U-Net provides to adjust results at test-time is very useful for an artist.

4.1.3. Architectures Comparison

In this section the proposed U-Net customized model with a single-scale style loss and instance normalization is compared with:

- Jhonson' architecture [JAL16] based on residual layers and instance normalization previously used on image style transfer (uses learning rate 0.001).
- Ulyanov' multi-scale architecture [ULVL16] previously used on image style transfer with batch normalization (uses learning rate 0.00001).
- Architecture based on SRResNet and residual layers which is deeper and uses fewer filters compared to Johnson' (uses learning rate 0.00001). See table A.7 for architecture details.

Figure 4.13 shows the comparison of the architectures and is noticeable that U-Net with instance normalization can minimize the style loss very well without sacrificing speed.

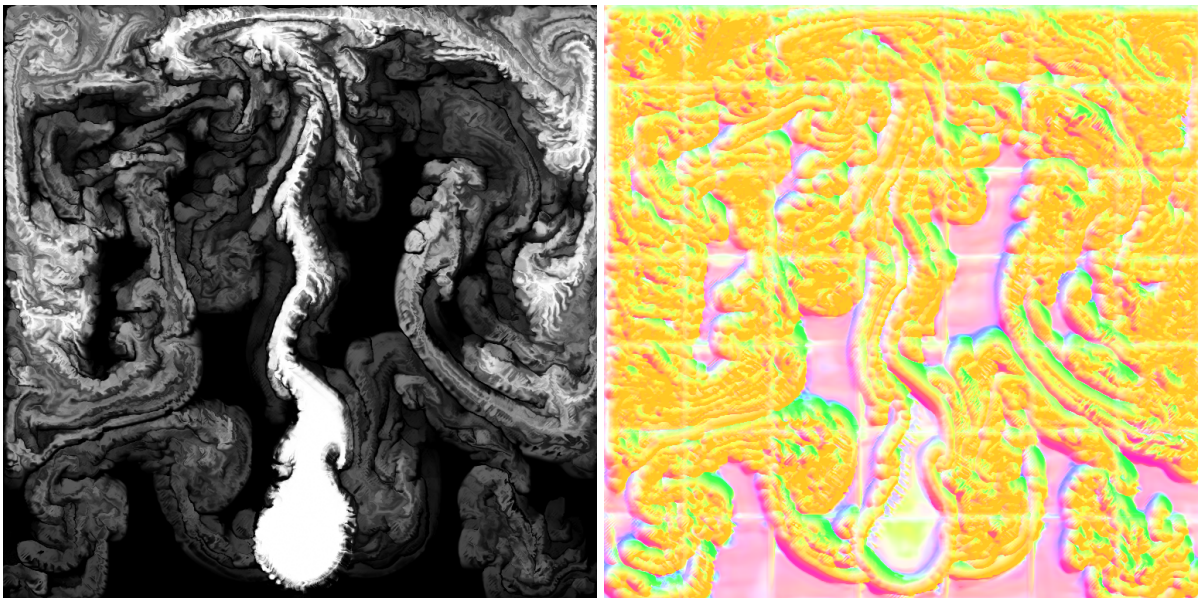
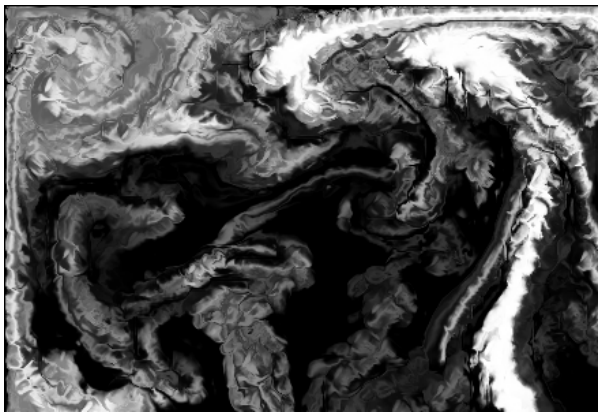
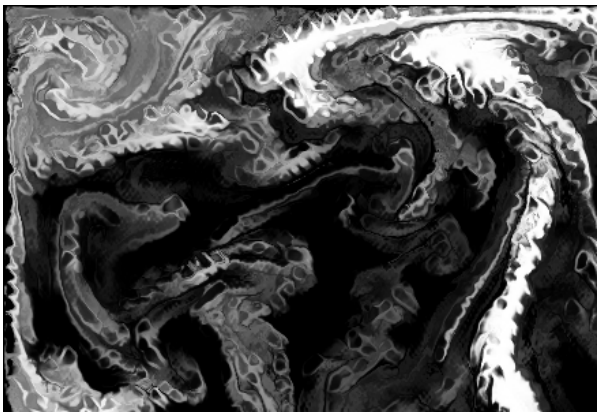


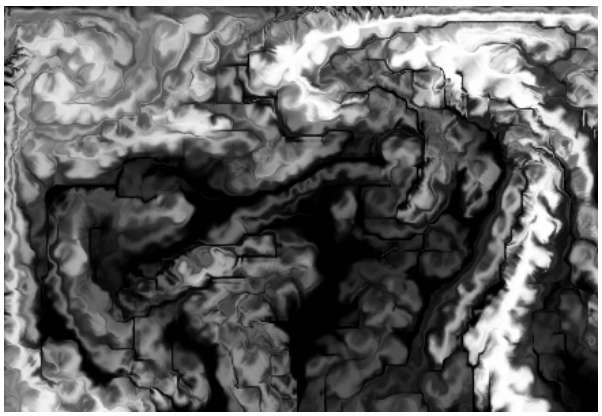
Figure 4.10.: *2-scales with U-Net$_{+2}$ with residual velocities and using Gaussian pyramid for density patches following figure 3.1 from method section without modifications.*



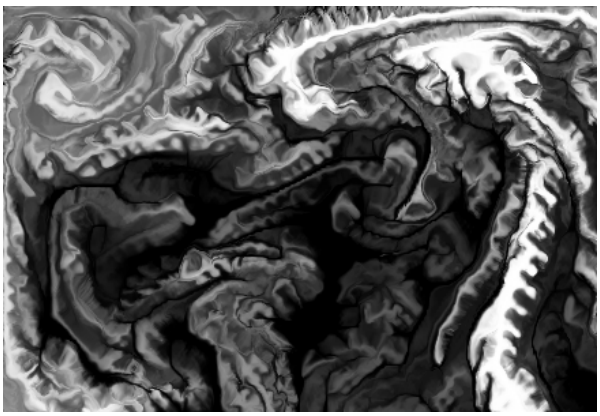
(a) Ben Giles High-Pass effect. Tanh factors are 0.02 (level 1 pyramid) and 0.08 (level 0 pyramid)



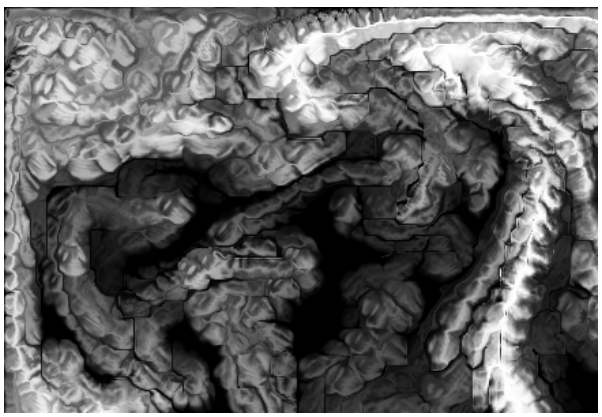
(b) Peace High-Pass effect. Tanh factors are 0.02 (level 1 pyramid) and 0.08 (level 0 pyramid)



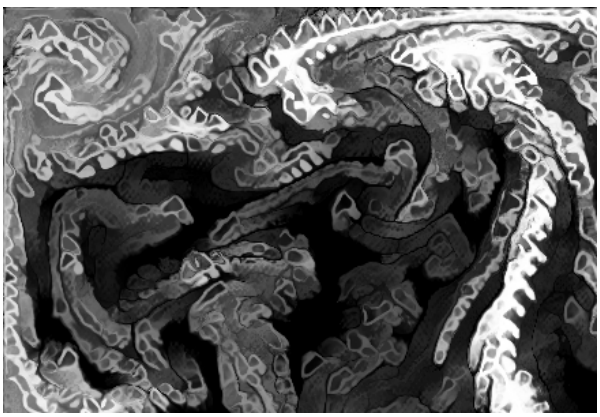
(c) Ben Giles Low-Pass effect. Tanh factors are 0.08 (level 1 pyramid) and 0.02 (level 0 pyramid)



(d) Peace Low-Pass effect. Tanh factors are 0.08 (level 1 pyramid) and 0.02 (level 0 pyramid)

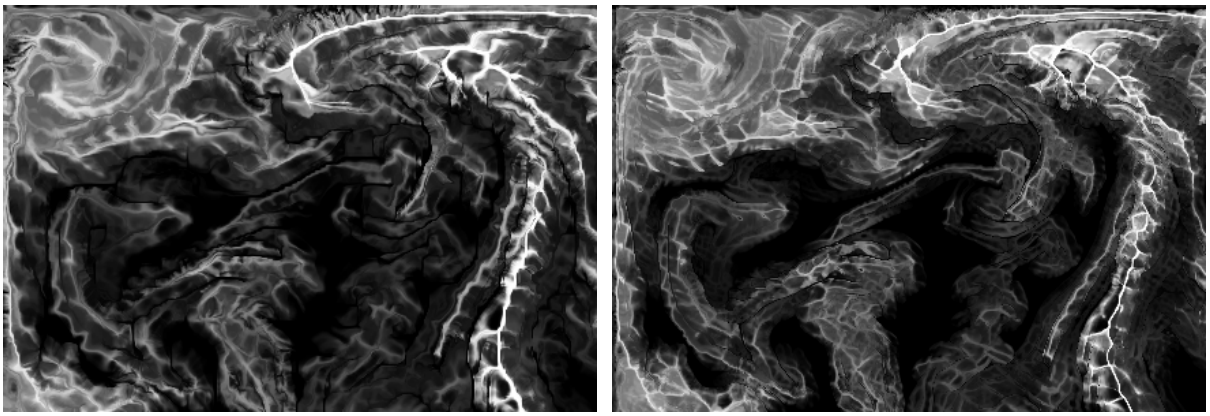


(e) Ben Giles two-scales default



(f) Peace two-scales default

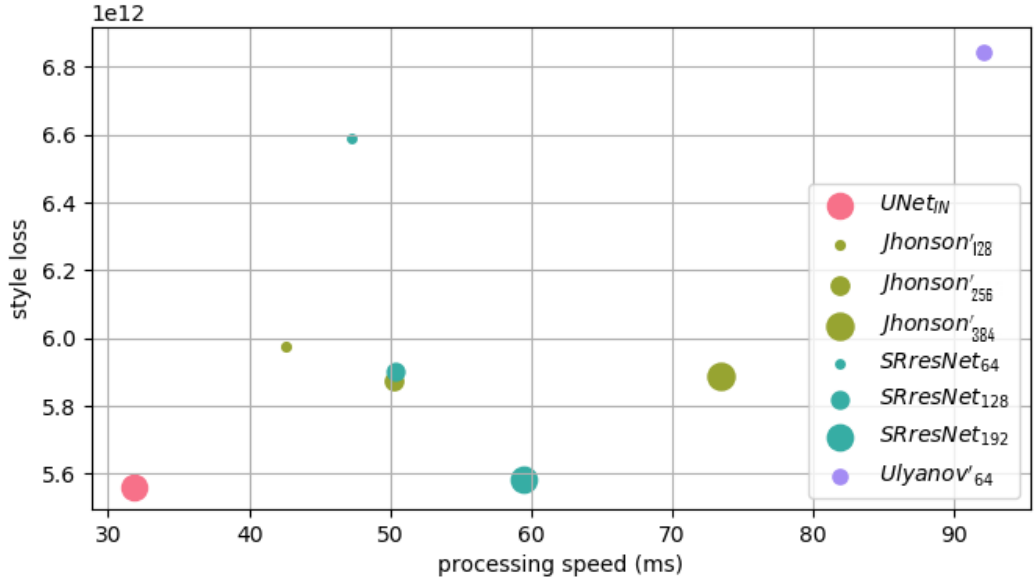
Figure 4.11.: *Adjusting style scales at test-time by changing Tanh factors.*



(a) Boost Dark Matter high-level features. Tanh factors are 0.12 (level 1 pyramid) and 0.02 (level 0 pyramid)

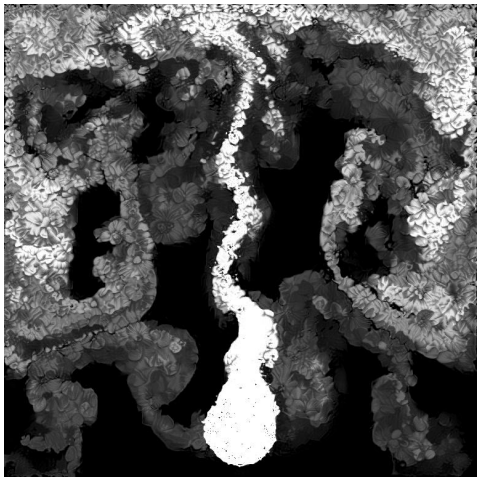
(b) Amplify single-style-scale U-Net velocities. Tanh factor is changed from 0.08 to 0.12 at test-time.

Figure 4.12.: *Amplify stylization effects*

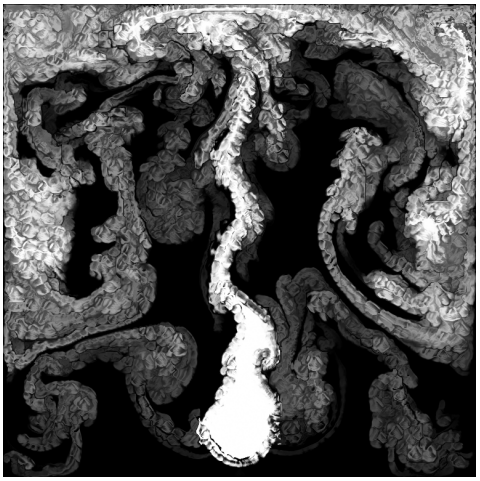


. Results correspond after 8 hours of training where all models have correctly converged.

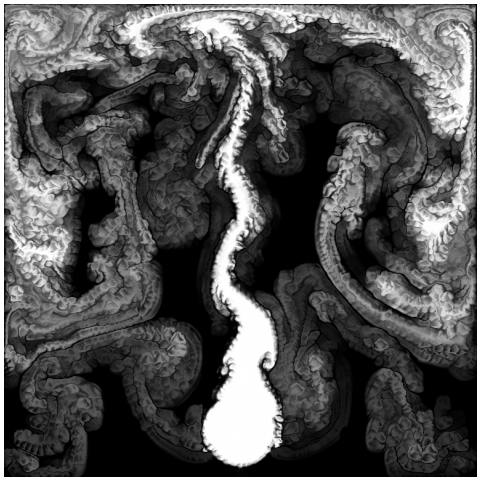
Figure 4.13.: *Comparison of accuracy, speed, and model size for different 2D architectures with a 1-scale style loss. Models near to the bottom-left corner are better in terms of speed and stylizing capabilities. Speed correspond to the average to process a patch of size 128^2. The radius of each circle represents the number of parameters of the model. The test-time style loss correspond to the average over three style scales and two styles images (Ben Giles and Dark Matter), for details refer to A.1*



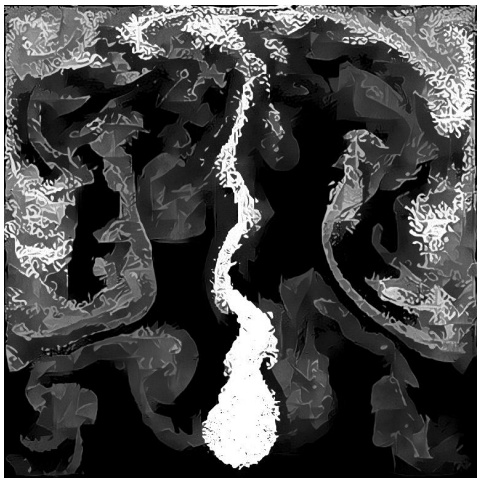
(a) Iterative TNST with 2 scales (1k iterations in total) for Ben Giles



(b) 2-scales feed-forward for Ben Giles



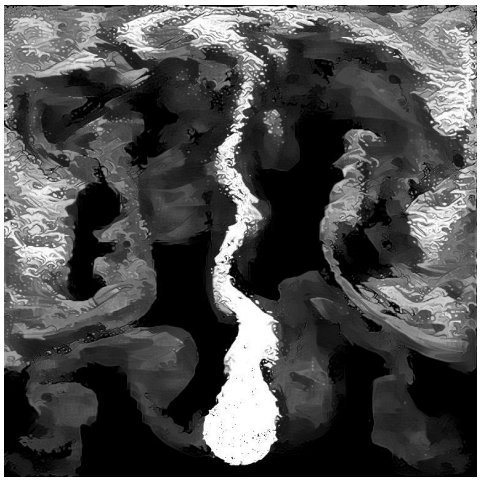
(c) 1-scale feed-forward for Ben Giles



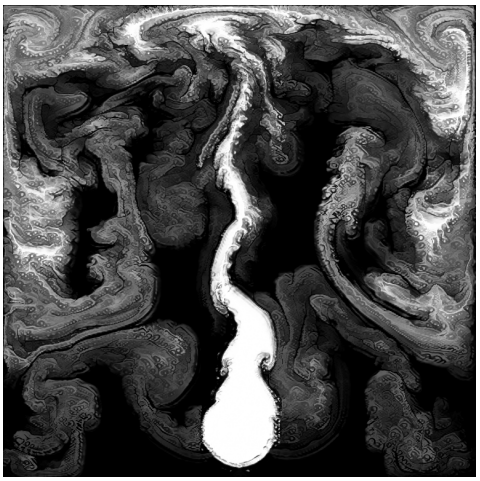
(d) Iterative TNST with 2 scales (1k iterations in total) for Peace

(e) 2-scales feed-forward for Peace

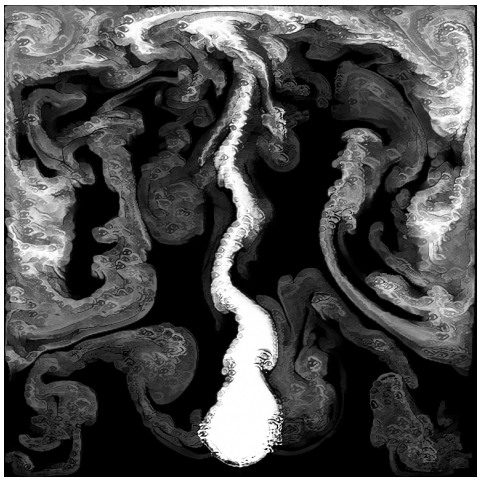
(f) 1-scale feed-forward for Peace



(g) Iterative TNST with 2 scales (1k iterations in total) for Wave



(h) 2-scales feed-forward for Wave



(i) 1-scale feed-forward for Wave

Figure 4.14.: *Feed-forward with residual velocities U-Net$_{+2}$ and batch normalization compared to iterative TNST for two-scale style. Single-scale feed-forward result with batch normalization is also shown. 16 pixels overlap and velocity interpolation is used in the CNN results.*

Training can become unstable on an architecture based on Jhonson', without gradient clipping the model fails to converge. In addition, on some styles, such as Dark Matter, if the velocity mask is disabled it also fails to converge: An explanation for this behavior is that the velocity mask reduces the space of possible solutions and guides the network on learning non-ill transformations. Also, using more smooth ELU activation units instead of ReLU can help.

Compared to U-Net which provides good-quality results with the prescribed number of filters used on image-based processing. Using the prescribed number of filters on the architectures based on SRResNet_{64} or Ulyanov_{8} gives poor results. Better results can be achieved by increasing the number of filters (SRResNet_{128}, Ulyanov_{64}) however this also slows the processing speed.

4.2. 3D

Due to memory restrictions, patch size is set to 64^3 on most experiments and training batch-size is one on all experiments. Since batch-size it is one, it is adequate to use instance normalization, which we use for all 3D experiments. Note that batch-normalization can be used with batch-size of one when momentum is activated because because the mean and variance estimates are computed incrementally, however instance normalization showcases sharper results at test-time. Depending of the resolution of the style image it is possible to train with higher patch-size, on some tests it was possible to train at 80^3 on a 1080 Ti (11GB of dedicated memory) high-end GPU. All 3D models use replication padding for convolutions. Excluding the dense-block experiment in section 4.2.7 all other experiments use models with convolution blocks based on two convolution layers for which a detailed description of the architecture for 1-style-scale and 2-style-scales is found in tables A.4 and A.5.

For the velocity visualization, one slice along depth dimension is used and only x and y vector components are considered for the color-map. Note that since on 3D velocity magnitudes vary more from style to style, a different limit has been used for each style. The mapped velocity for the same style corresponds across different models and can be directly used for comparison.

The test-loss at each step is evaluated by taking a random patch on the last frame of the "smoke gun" simulation shown in figure 4.15. The same frame is used to compare models.

On many of the examples we indicate if the patches are aligned to the top of the domain, this allows to compare how the patch behaves on the borders. For the final rendered results we use padding at the beginning and end of the domain to avoid artifacts (see fig. 4.23 to see how border artifacts are avoided). We use the same minimal orthogonal renderer used in training to show many of the figures results (those with black-background) and Mantra for final results with more complex lighting configurations.

4.2.1. Advection Order

Compared to 2D, we have found the advection order to play a more important role for 3D quality results. When first order advection is used for training, the final velocity field lacks

Figure 4.15.: *Original Smoke gun frame without stylization*

detail on some areas: For example on fig. 4.16(a), with first order advection the network learns to harshly push the densities towards the center of the density field leaving an empty area where originally there were densities. In comparison, with second order Maccormak advection 4.16(b) the velocity field looks more turbulent and can generate more correct details on the same region. In addition, it is possible to train on second order advection and use first order advection at test-time, this approach still gives preferable results than using first order at training stage. When using first-order at training stage and second-order at test-time details are smooth out excessively and lost.

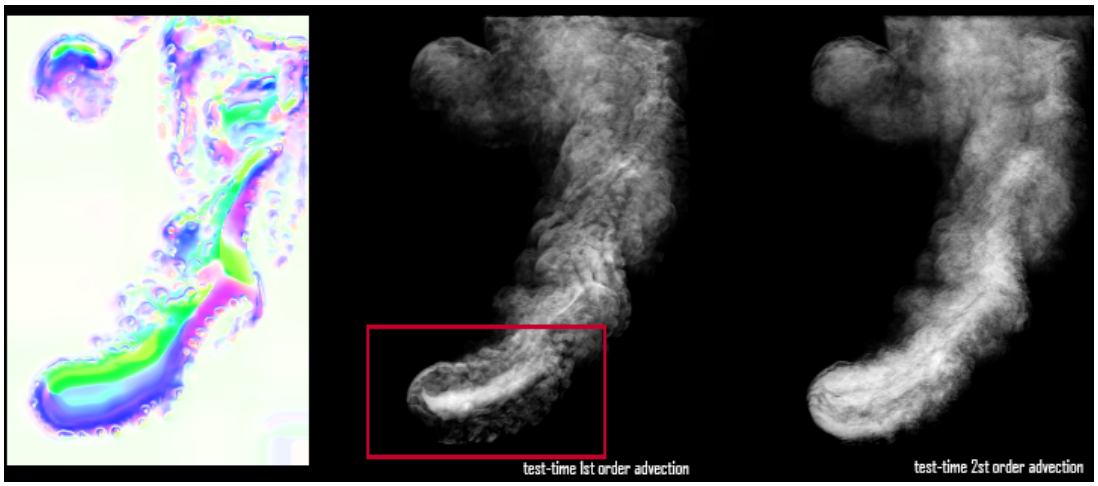
4.2.2. Camera sampling

Camera parameters were tuned on experiments made on a smaller data-set (several smoke gun sequences) without augmentation and with random patch extraction. We found sharper results when visualizing the stylized density field in Houdini when using many orthogonal cameras (we are able to configure up-to 16 simultaneous views at each training step without running out of memory) combined with position sampling at each optimization step 4.17(b). View-sampling results in sharper results. The same configuration was used for the rest of the experiments on the large-dataset.

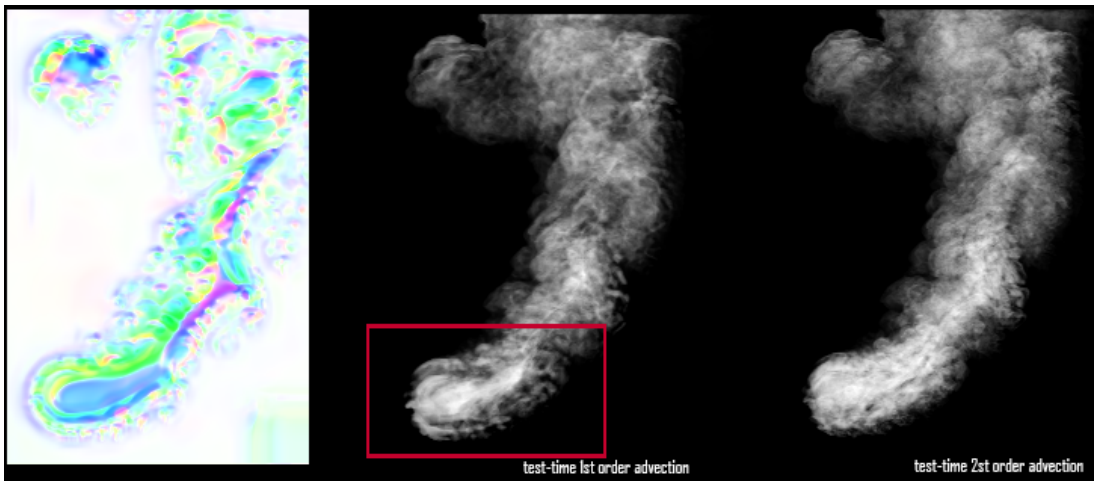
4.2.3. Style Scales and Velocity Masking

Spirals

Comparing a U-Net$_1$ model without residual velocities and with a single-scale style loss to a model with two-scale loss U-Net$_{+2}$, the most differences can be found with Spirals style, which has also been the most difficult style to tune to obtain good results.

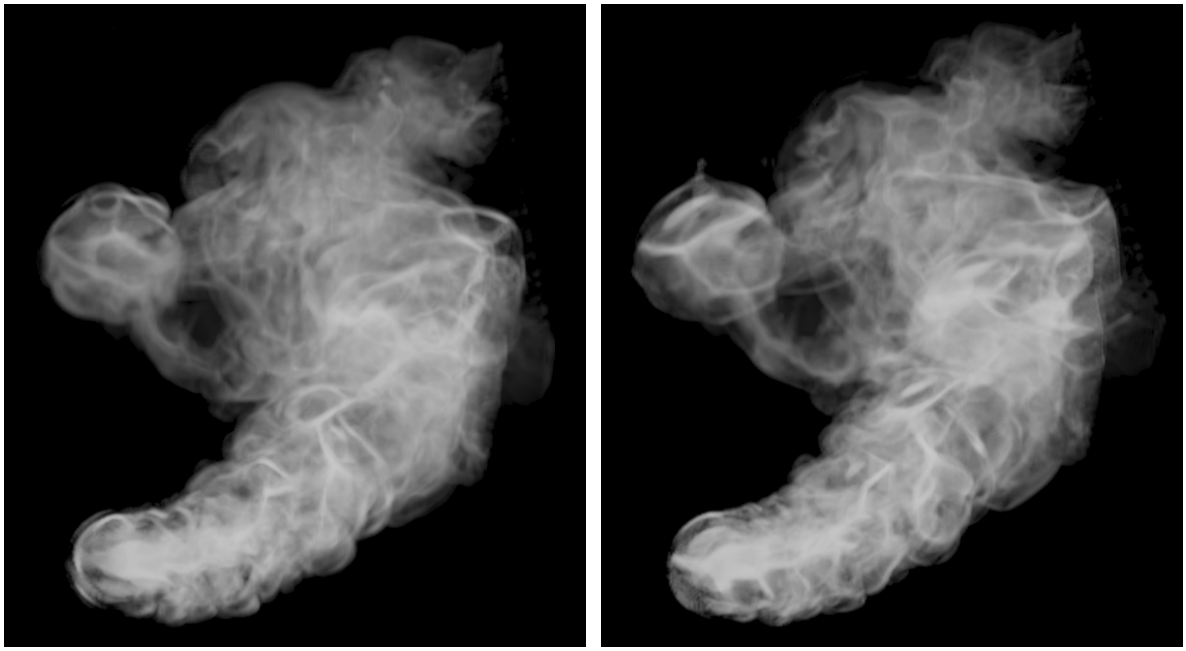


(a) Results for model trained with first order advection



(b) Results for model trained with second order advection

Figure 4.16.: *Models trained for Volcano style and different advection configurations. Interpolation at final velocities between 32-voxel patches overlap is performed. Note that the results correspond to 24 hours of training.*



(a) 16 fixed views *(b) 16 views sampled at each step*

Figure 4.17.: *Camera sampling comparison for small data-set and Dark Matter style. Both models were trained for the same number of steps (15k). Visualization of densities is done with perspective camera on Houdini.*

When training $Spirals_{1.0}$ with patch-size 80^3 the network learn to produce the biggest sized patterns, for this case a single-scale model has showcased difficulty to learn patterns and the two-scale model is prone to showcase border artifacts. When $Spirals_{1.0}$ is trained at path-size 64^3 both models are able to produce slightly smaller shapes. In this case, both using a two-scale model and disabling mask can help to show more clear and sharp patterns, however, using a single-scale loss with train-time mask showcases the best temporal coherence. With $\text{Spirals}_{0.5}$ style image the smallest patterns are produced however even when mask is used both single-scale and two-scale showcase similar temporal instabilities and further temporal post-processing is required.

For $\text{Spirals}_{0.5}$ style the test-loss show similar values for level 0 but U-Net_{+2} is able to showcase significant decrease on level 1 loss compared to U-Net_1 both when train-time velocity mask is activated or deactivated (see figure 4.18(a). Trying a U-Net model without residual velocities and 2 style-losses the model is not able to converge due to training instabilities, trying to increase maximum allowed velocity (tanh factor 0.256) on a single-scale loss model results on worse loss values (figure 4.19).

When spiral style is trained with two scale loss U-Net_{+2} the network is able generate velocities that produce bolder changes than when trained with a single-scale loss U-Net_1 (figure 4.18). If the stylization results are too harsh, the stylization strength can easily be controlled by reducing stylization velocity magnitudes with a element-wise multiplication with a strength factor. A more advanced control is possible by changing Tanh factors for each scale at test-time (figure 4.21(f)). When doing temporal velocity post-processing the results are already being smoothed and it might be preferable to choose the sharpest configuration.

Dark Matter and Volcano

For Dark Matter and Volcano style, the U-Net$_{+2}$ is also able to showcase greater decrease on level 1 pyramid test-loss (on volcano only if mask is deactivated) - see right hand side of figures 4.18(b) and 4.18(c). Loss values for level 0 are similar, or slightly worse when mask is deactivated. However, because the styles are based on smaller patterns, the fine-grained results that the U-Net$_1$ model is able to produce are visually preferable. On volcano style the use of a velocity-mask at train-time combined with a single-scale loss shows the best results and loss values (left-hand side of 4.18(c)) for scale 0. For Dark Matter the best quality is achieved by only applying the velocity-mask at a post-processing stage, the model that learns the mask at training-time showcases artifacts at the beginning of the "smoke gun" sequence (see figure 4.24).

4.2.4. Temporal coherence

Some of the models can showcase temporal incoherence more than others. By tuning training patch-size, style image resolution, and the mask it was possible to stylize one sequence of Spirals without any temporal problems. In cases of Spirals where there is temporal inconsistencies, they can be significantly reduced by smoothing velocities on a frame-window (method explained on section 3.4). Figure 4.26(b) show how with temporal smoothness post-processing there is less undesired variance in the results of consecutive frames

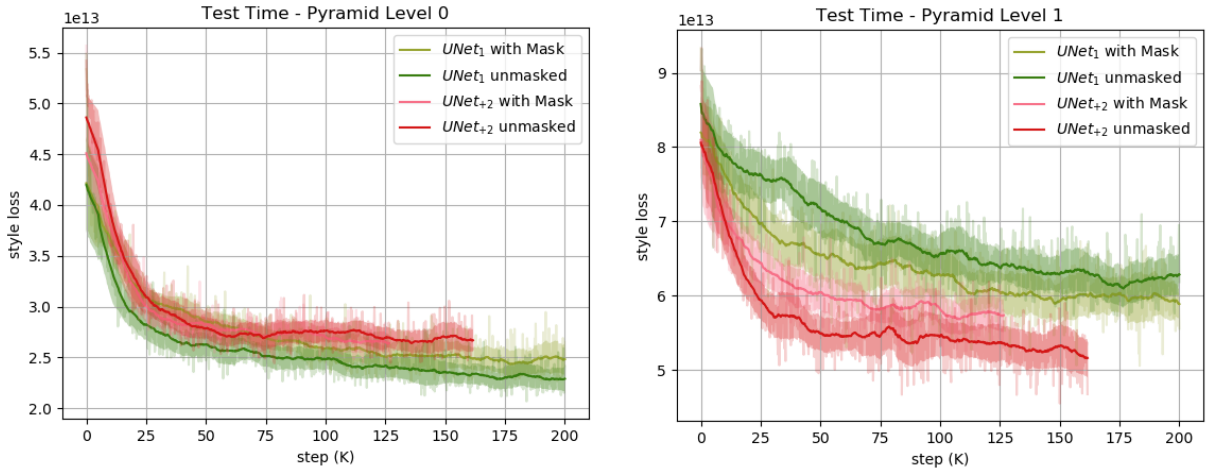
Dark matter models generally don't showcase any temporal incoherence but when trained without mask and the velocity-mask is applied at a post-processing step there are some minor inconsistencies near the surface of the smoke in the direction it is moving. In volcano temporal artifacts are more apparent, without any post-processing the flickering is noticeable, see volcano_window0.mp4 on supplementary material. In this case a post-processing window of 3 or 5 can reduce flickering, see *volcano_window3.mp4* and *volcano_window5.mp4*.

4.2.5. Performance

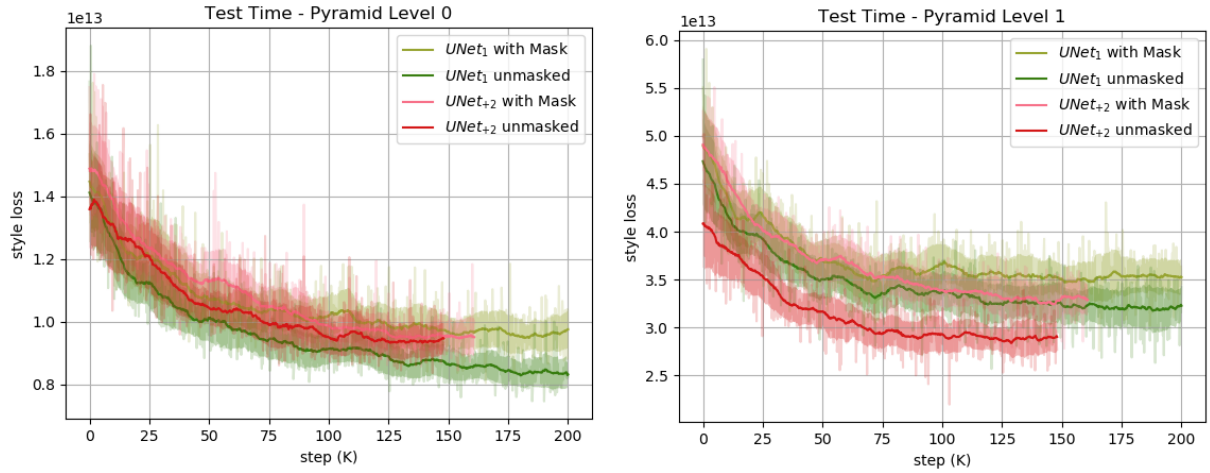
Table 4.1 show that with a high-end GPU, the method need less than 40 minutes to processes the 120 frames from the "smoke gun" sequence, taking below 20 seconds to process all patches on a frame. In addition, if temporal smoothing is required it can take at least 7.6 seconds more per frame to temporally smooth the velocities (see table 4.2). An iterative TNST approach needs between 10-20 mins per frame for a volume of the same size - our method is two orders of magnitude faster. Our method can even be run on a lower-end laptop GPU (see table 4.3) needing significantly less time to process a high-resolution frame than iterative TNST.

4.2.6. Stitching based on Feature Interpolation

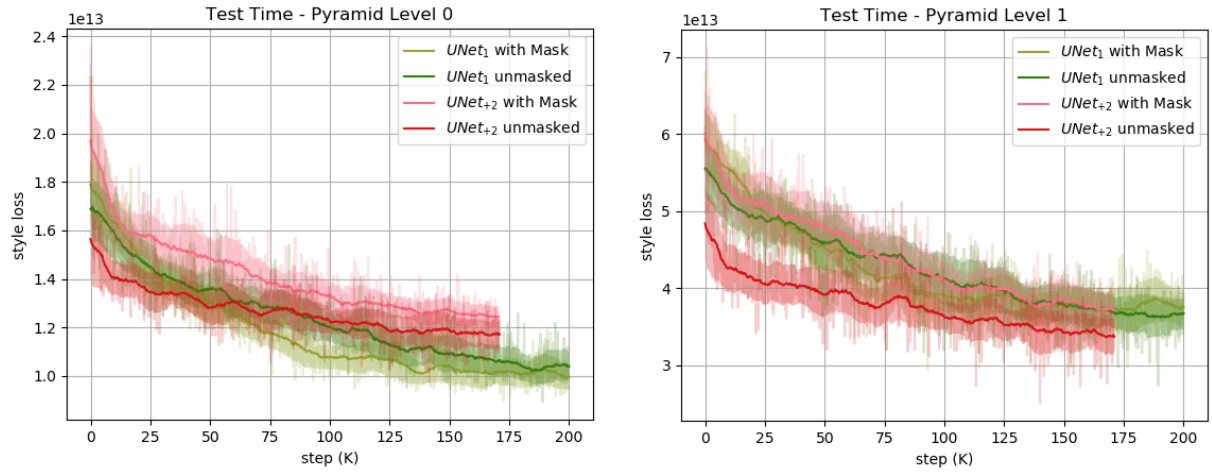
The results on figures 4.27 and 4.28 show that is possible to efficiently fix most of the spatial incoherence's using a feature interpolation approach without any overlap. The number of patches required to cover the volume is smaller and it is possible to process the volume in less time



(a) Spirals$_{0.5}$



(b) Dark Matter



(c) Volcano

Figure 4.18.: *Loss values for mask and multi-scale 3D style options after 48 hours of training*

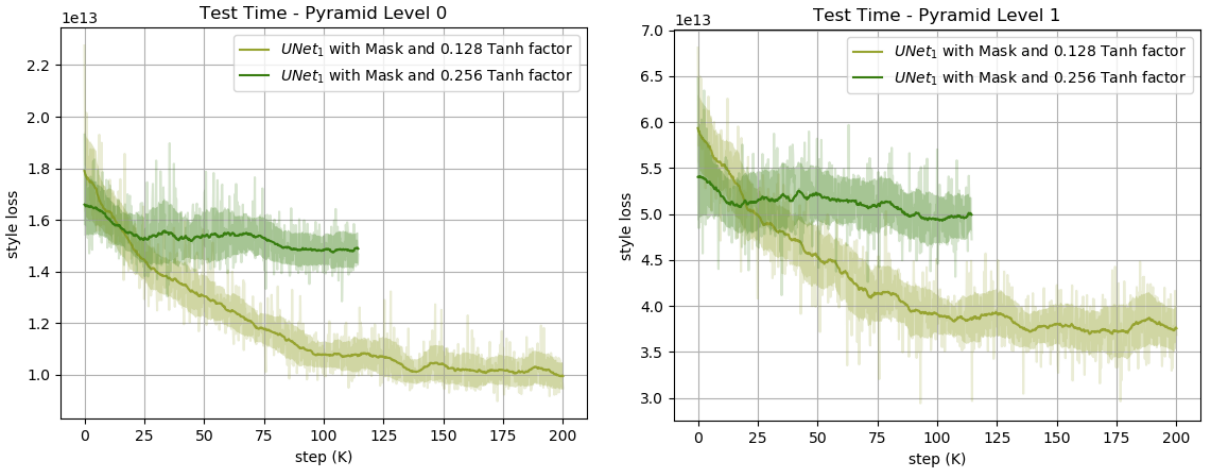
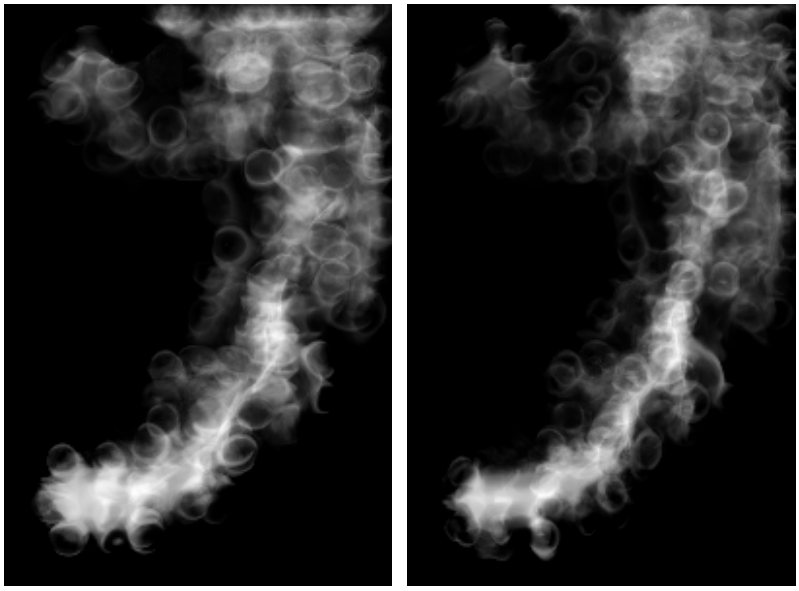
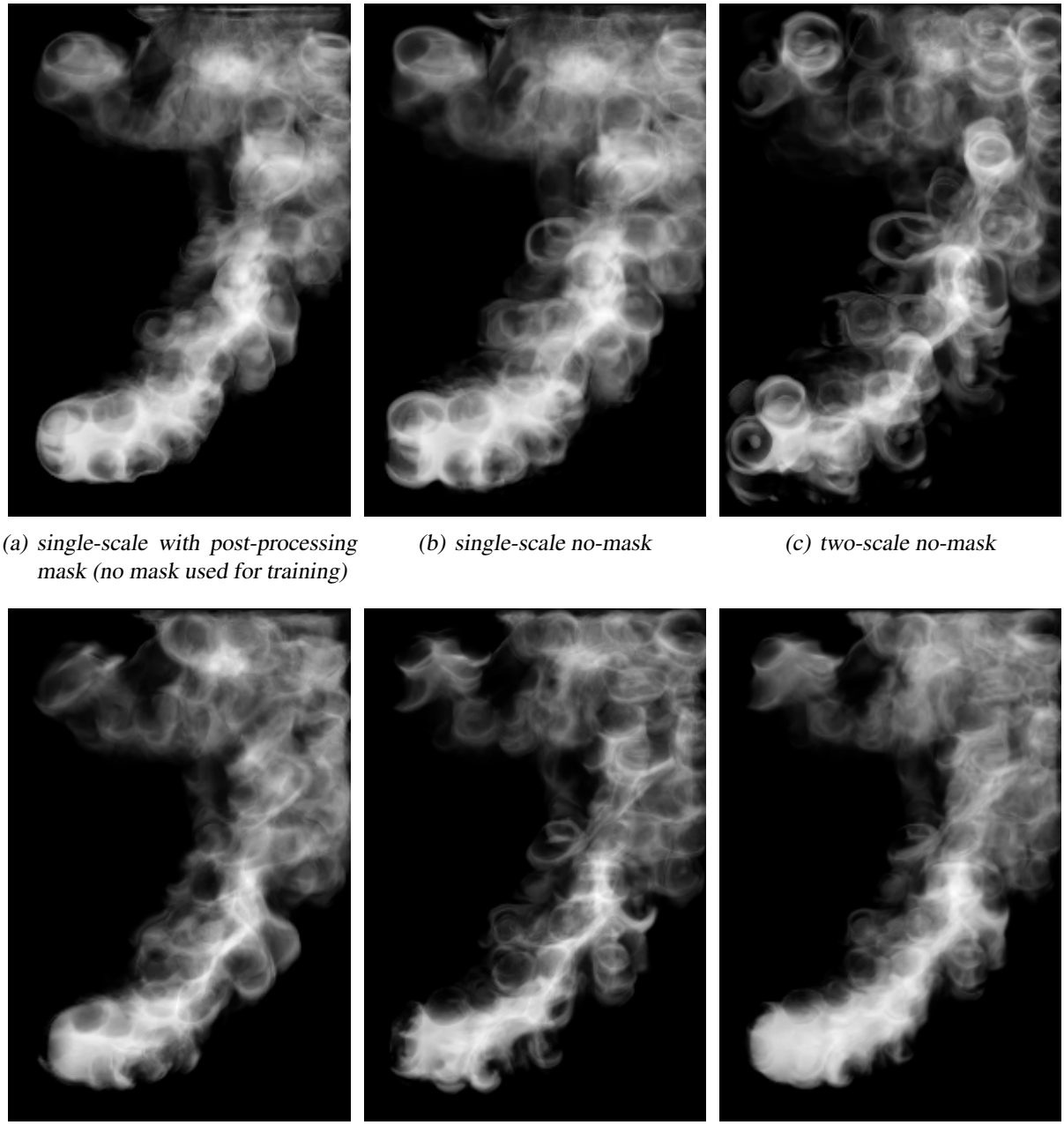


Figure 4.19.: *Train-time Tanh factors for Volcano style*



(a) single-scale with velocity-mask *(b) two-scale with velocity-mask*

Figure 4.20.: *1-scale U-Net$_1$ and 2-scale U-Net$_{+2}$ results and masking configurations after 48 hours of training for Spirals$_{0.5}$. Velocity-masking is learned at training time so that the network can directly produce masked results. Patch border is aligned to the top of the domain so border artifacts can be visible.*



(a) single-scale with post-processing mask (no mask used for training)

(b) single-scale no-mask

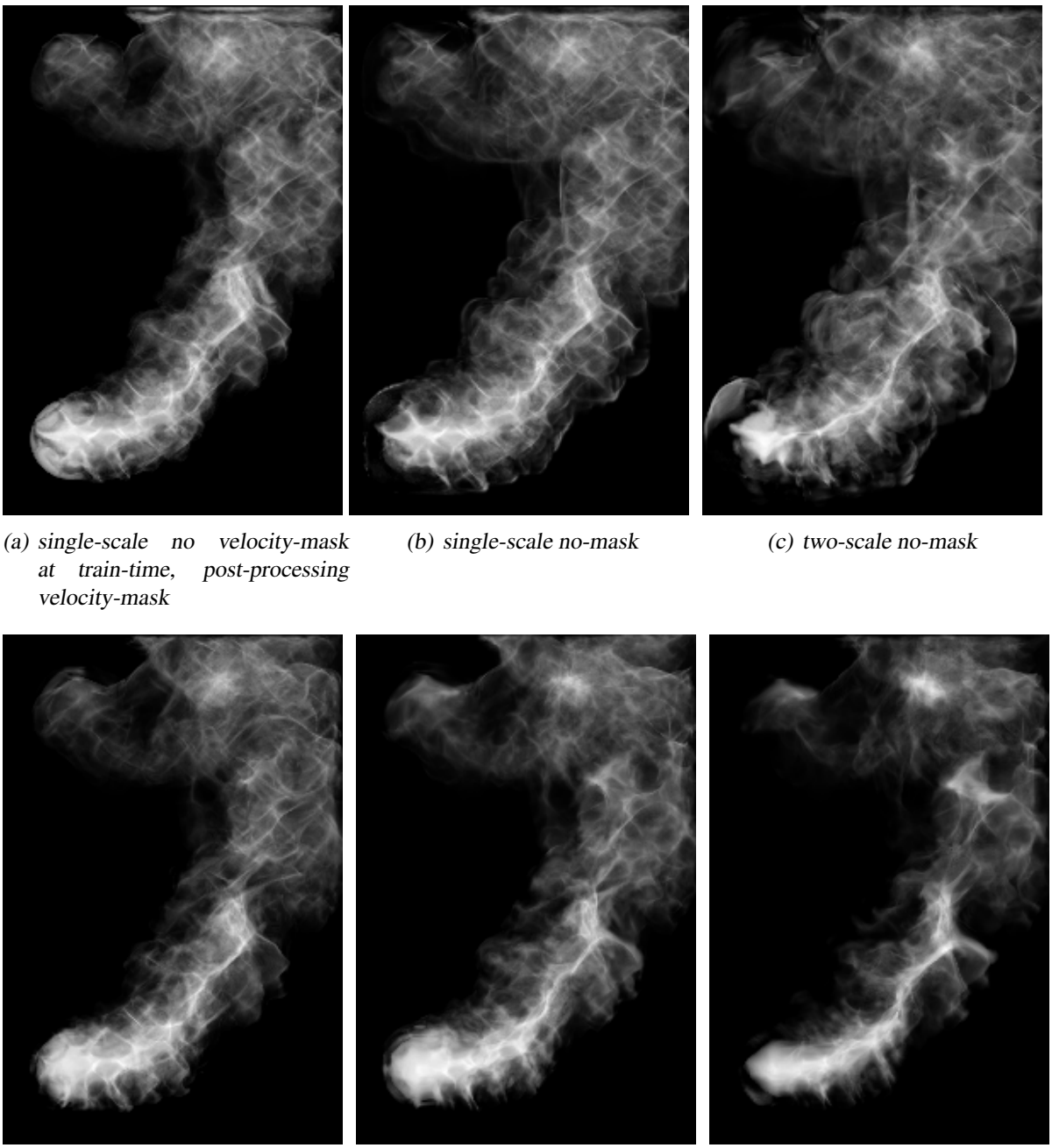
(c) two-scale no-mask

(d) single-scale with learned velocity-mask

(e) two-scale with learned velocity-mask

(f) two-scale with learned velocity-mask and test-time modified Tanh factors (0.064, 0.128)

Figure 4.21.: *1-scale U-Net$_1$ and 2-scale U-Net$_{+2}$ results and masking configurations after 48 hours of training for Dark Matter. Velocity-masking is learned at training time so that the network can directly produce masked results. Patch border is aligned to the top of the domain so border artifacts can be visible.*



(a) single-scale no velocity-mask at train-time, post-processing velocity-mask

(b) single-scale no-mask

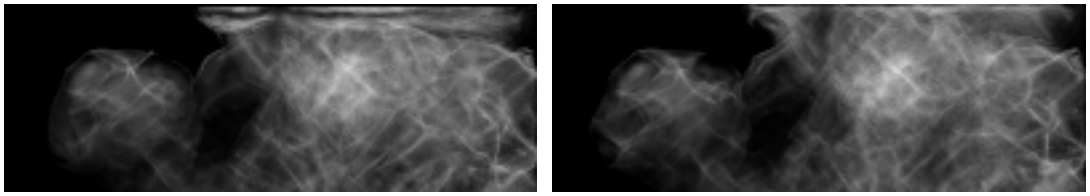
(c) two-scale no-mask

(d) single-scale with trained velocity-mask

(e) two-scale with trained velocity-mask

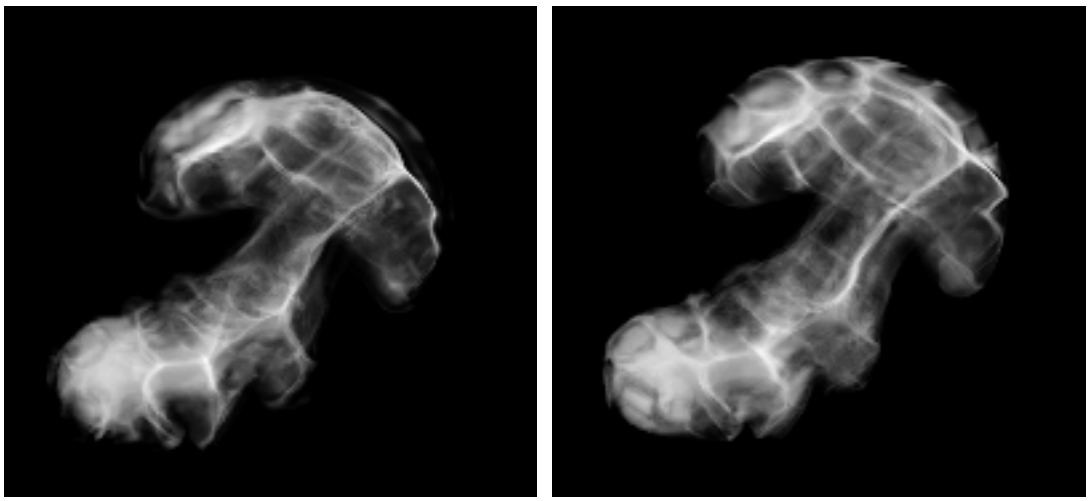
(f) two-scale with velocity-mask and test-time modified Tanh factors (0.128, 0.032)

Figure 4.22.: *1-scale U-Net$_1$ and 2-scale U-Net$_{+2}$ results and masking configurations after 48 hours of training for Dark Matter. Velocity-masking is learned at training time so that the network can directly produce masked results. Patch border is aligned to the top of the domain so border artifacts can be visible.*



(a) Patches aligned with top of the domain (b) Density with 20 voxels of padding at the top

Figure 4.23.: *Using 20 of padding at the beginning and at the end of up-axis of the density field helps to avoid artifacts compared to adding 40 only at the bottom.*

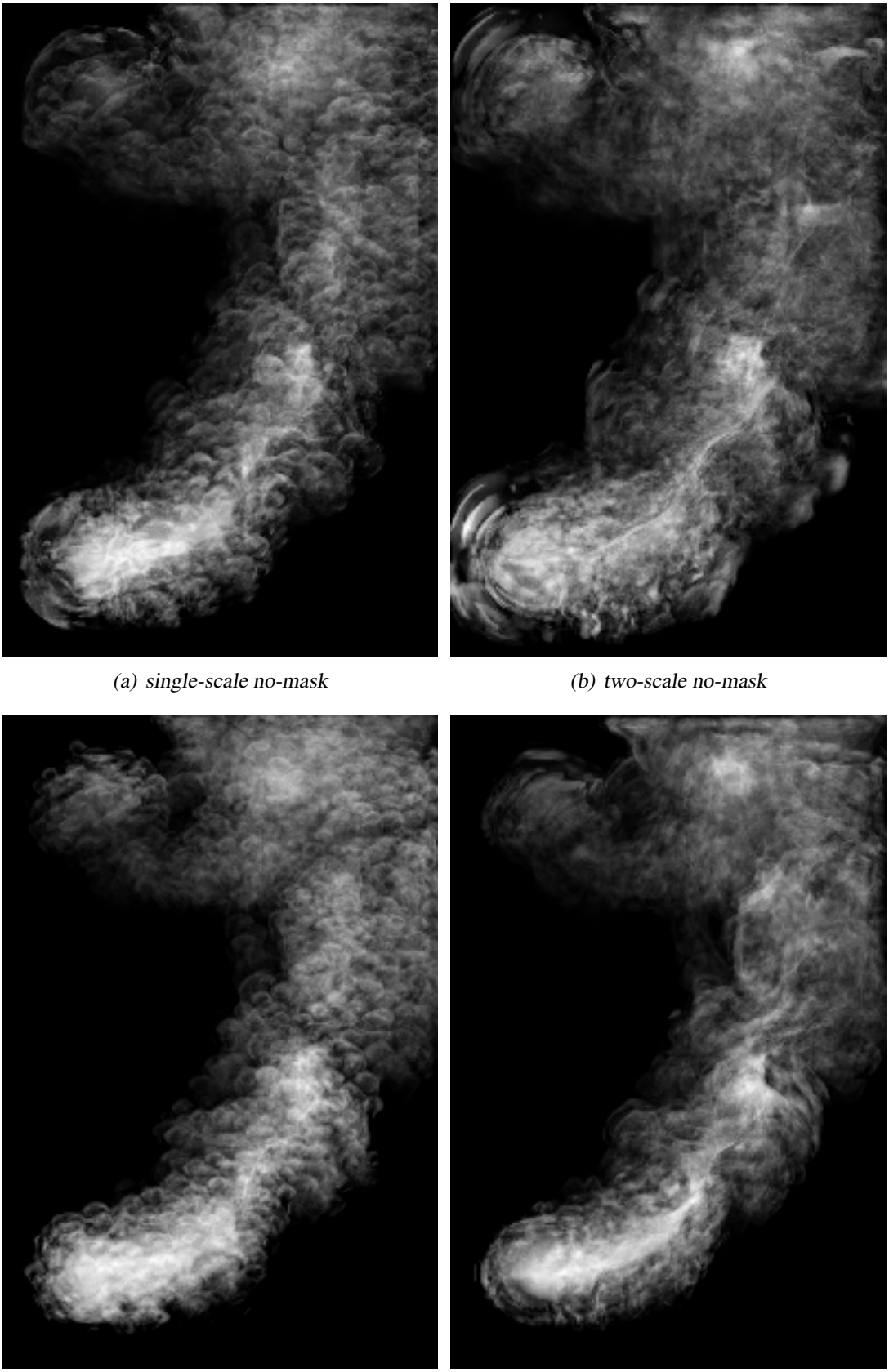


(a) With learned velocity-mask (b) Velocity-mask only at test-time

Figure 4.24.: *Comparison of frame 52 of the sequence with masking options for 1-scale Dark Matter model. . See dark_matter_x1scale_learned_mask.mp4 (a) and dark_matter_x1scale_post_mask.mp4 (b) on the supplementary material for the full sequence.*

120 Frames sequence	Total	Per Frame	Per Call
Process Entire Volume	37.19 min	18.59 s	18.59 s
Split into patches	1.65 min	0.82 s	0.82 s
Feed-forward Model	24.94 min	12.47 s	191.86 ms
Stitch Patches	4.53 min	2.26 s	2.26 s
Masking	3 min	1.5 s	1.5 s

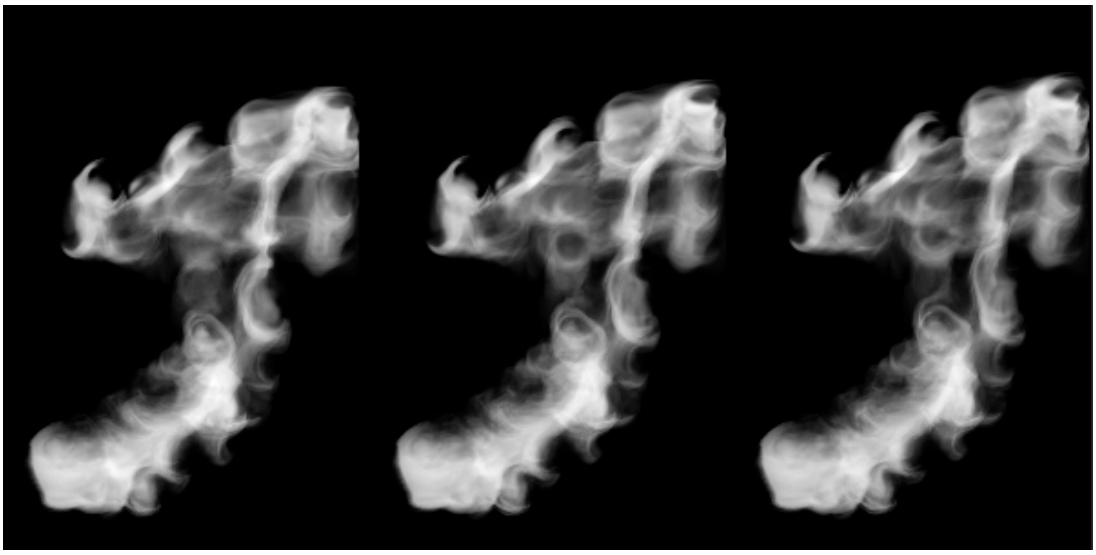
Table 4.1.: *Speed of feed-forward approach on a volume of $200x300x200$ with 32 voxel overlap. Each feed-forward call takes a batch of $5x80^3$ patches). Values are correspond to the average of three runs. Patch stitching is performed with velocity interpolation on overlaped regions. Tested on GTX 1080 Ti.*



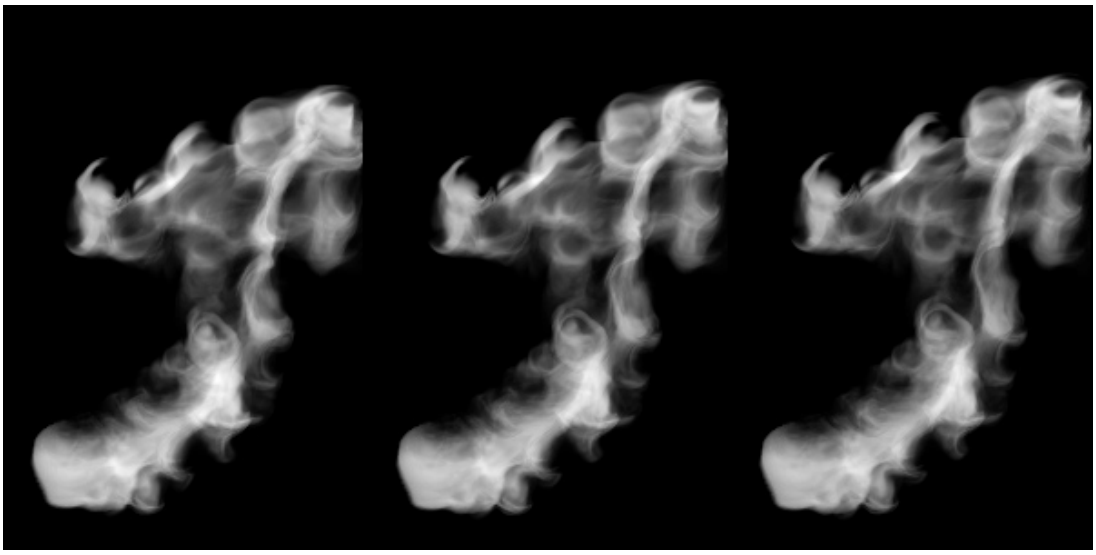
(a) single-scale no-mask (b) two-scale no-mask

(c) single-scale with velocity-mask (d) two-scale with velocity-mask

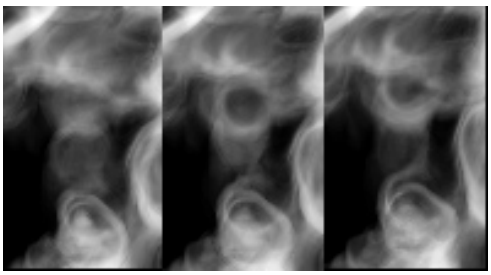
Figure 4.25.: *1-scale U-Net$_1$ and 2-scale U-Net$_{+2}$ results and masking configurations after 48 hours of training for Dark Matter. Velocity-masking is learned at training time so that the network can directly produce masked results. Patch border is aligned to the top of the domain so border artifacts can be visible.*



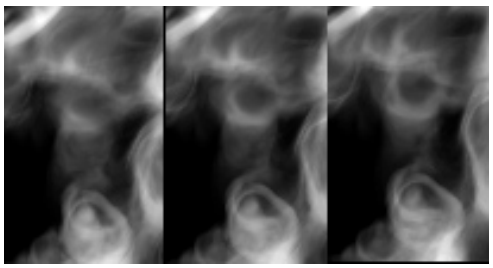
(a) Results without any post-processing



(b) After temporal post-processing



(c) No post-processing zoom



(d) Smoothed zoom

Figure 4.26.: *Effect of temporal post-processing for a U-Net$_{+2}$ model trained on Spirals style with patch size of 80^3. Three consecutive frames taken from the video spirals_tempo_vs.mp4 supplied in the supplementary material are shown. A window of size three is used for smoothing. Model trained and tested with patch size of 80^3. Stitching performed with velocity interpolation on a overlap of 32 voxels. During training a mask loss was activated, and no additional mask is used at post-processing.*

120 Frames	Total	Per Frame
Window size 5	25.5 min	12.8 s
Window size 3	15.2 min	7.6 s

Table 4.2.: *Speed of temporal post-processing measured on a volume of* $200x300x200$ *for a sequence of 120 frames at different window sizes. Tested on GTX 1080 Ti*

Volume $200x300x200$	32 overlap and linear interpolation of velocities	No overlap and feature interpolation
Total seconds	133.23	54.83
Total seconds for white cells		23.42
Total seconds for black cells		28.33
Number of patches	96	36
Average ms per feed-forward call	1269.09	1303.05

Table 4.3.: *Speed of two interpolation methods for a density field of* $200x300x200$ *with a U-Net*$_{+2}$ *model. Convolution blocks are comprised of two convolutions. Tested on GTX 960M.*

(see table 4.3). Since final velocities are not interpolated the results are also sharper. However, some patch to patch transitions can be noticeable and some of the patches whose features are interpolated (second pass) look less stylized (at the bottom and top-left part of the densities at the figure 4.27).

4.2.7. Dense Blocks

Models based on dense blocks have reduced number of filters per convolution (8 times less) and increased number of convolution layers. Although these models can be around x4 times faster than models based on convolutional blocks with two convolutional layers with more filters, they also showcase significantly worse style loss (figure 4.29). The quality of the results also is worse, specially on the volcano style (figure 4.30(b)). On dark matter the quality is more reasonable, in this case using dense blocks is a valid alternative that trades off some quality for higher speed and reduced model size (can be stored in 2MB compared to 150MB of U-Net with default number of filters).

4.2.8. Slice-based approach

It is possible to generate stylization by processing the volume on a single axis (for example the front view - see figure 4.31), however the details are lost when rotating the camera. The velocities resultant from processing the volume from two orthogonal axes (X and Z) can be combined by averaging y component of the velocities (figure 4.32), however the results lose

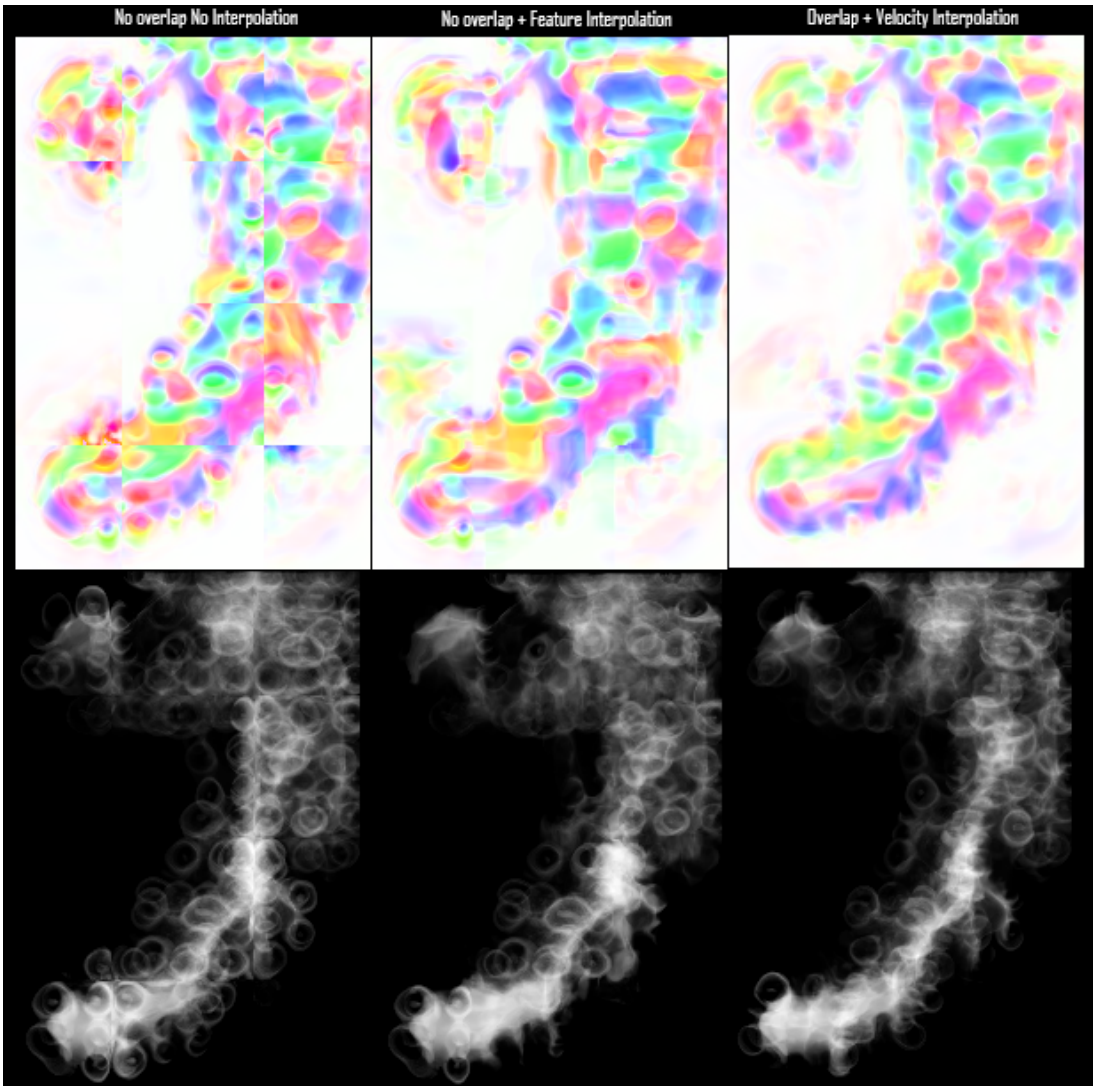
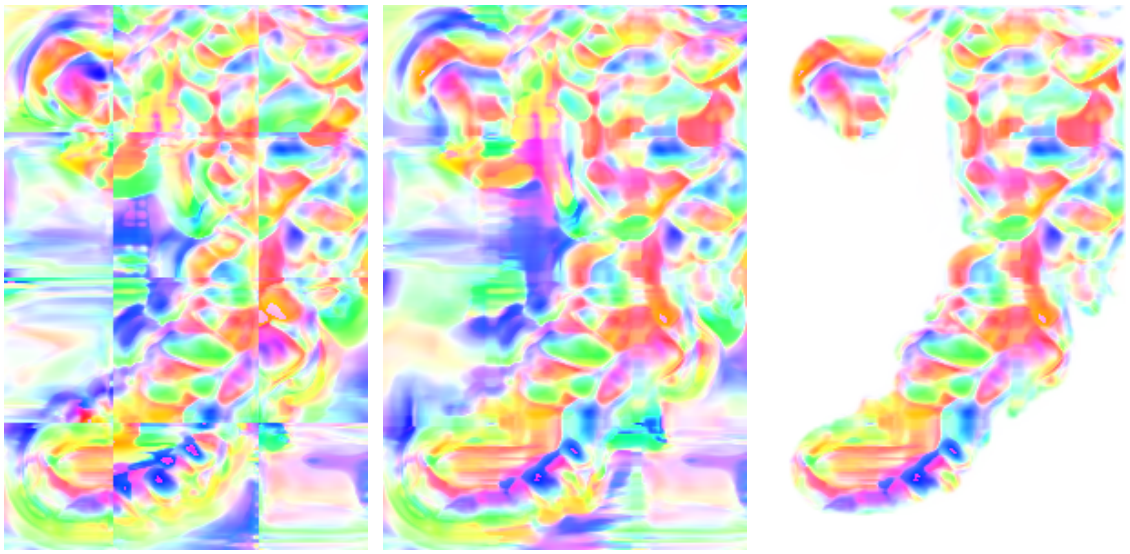


Figure 4.27.: *Comparison of (32-voxel) overlap and velocity interpolation (second column) against feature interpolation without overlap (third column) for Spirals$_{0.5}$. The first column has is the raw output without stitching and without any overlap. Model was trained for 24 hours on 64^3 patch size, test-time patch size is 80^3 .*



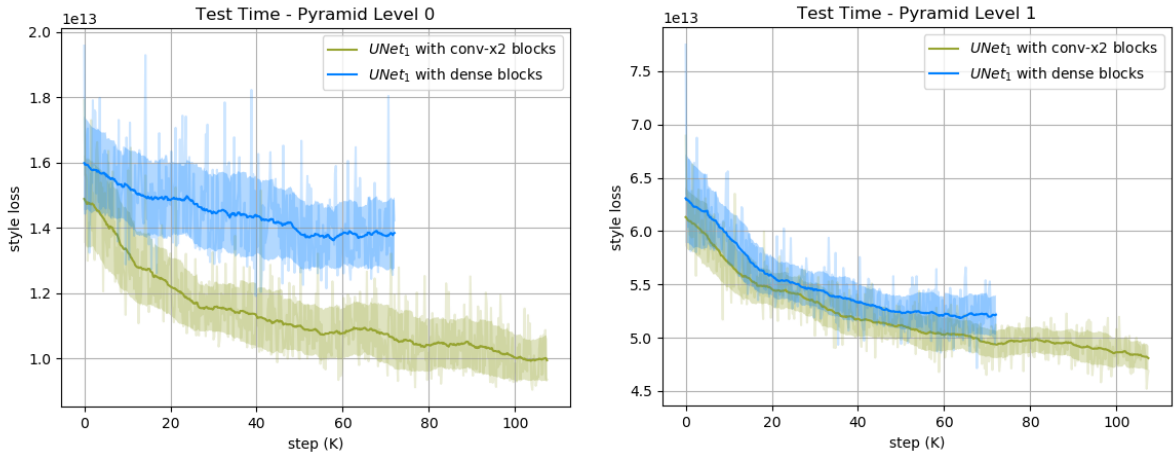
(a) Velocities without overlap and without interpolation

(b) Velocities without overlap and with feature interpolation

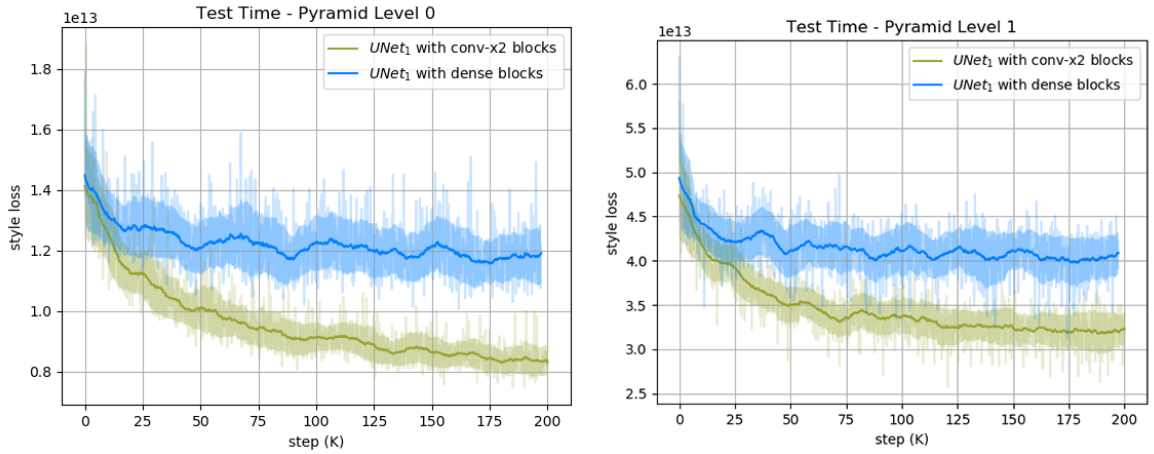
(c) Velocities without overlap, with feature interpolation and with post-processing mask

(d) Densities advected with feature interpolation masked-Velocities

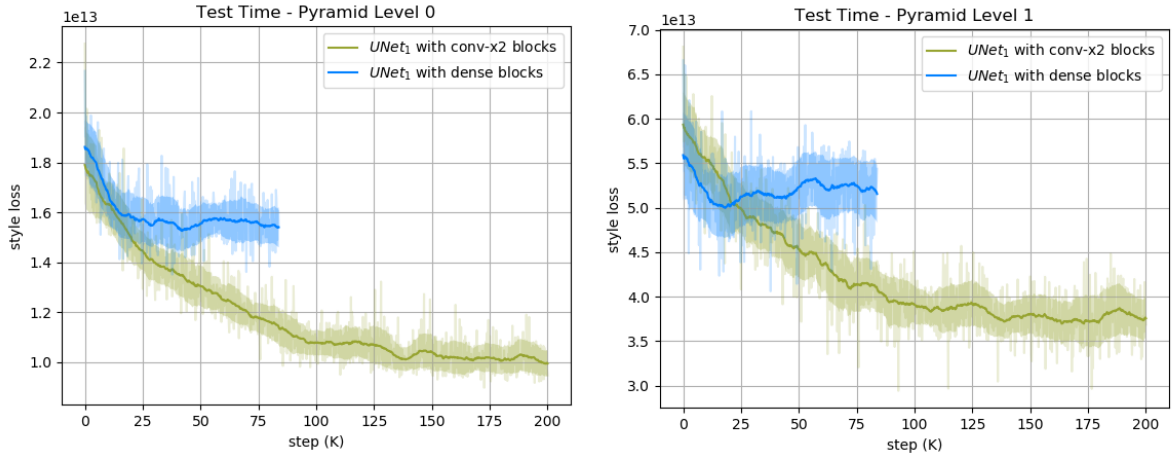
Figure 4.28.: *Feature interpolation on Dark Matter (1-scale and post-processing mask)*



(a) Spirals$_{1.0}$ (both models without train-time mask)



(b) Dark Matter



(c) Volcano

Figure 4.29.: *Loss comparison for U-Net with dense blocks and reduced number of filters*

(a) Dark Matter (b) Volcano

Figure 4.30.: *Stylized results for U-Net with dense blocks and reduced number of filters*

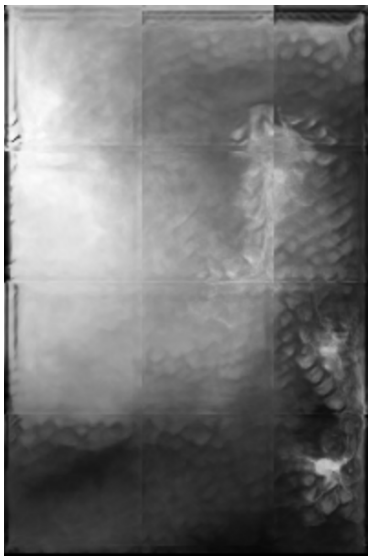


Figure 4.31.: *Slice-based approach for Ben Giles. The render-view is aligned with the slicing-axis so that features are sharp. The volume is of* $200x300x200$ *and has been processed in sub-volumes of* $200x80x80$ *without overlap*

some detail and the details are still not sharp when rotating the volume. In addition, the slice based approach generates smaller patterns.

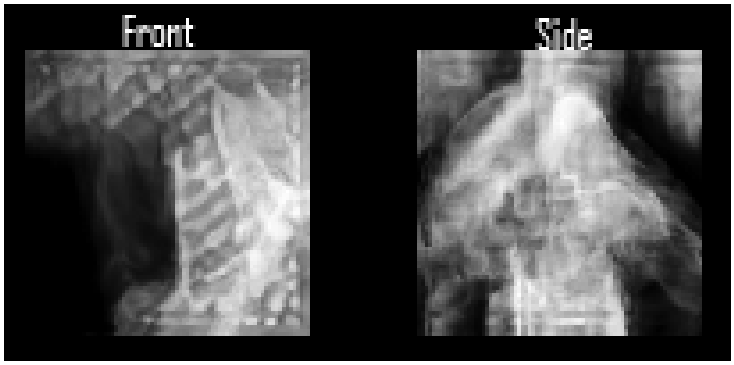
4.2.9. Final Rendered Sequences

The most promising configurations have been used to generate an stylized smoke that then has been rendered on Houdini with Mantra renderer. In particular we have rendered "smoke gun" sequence for:

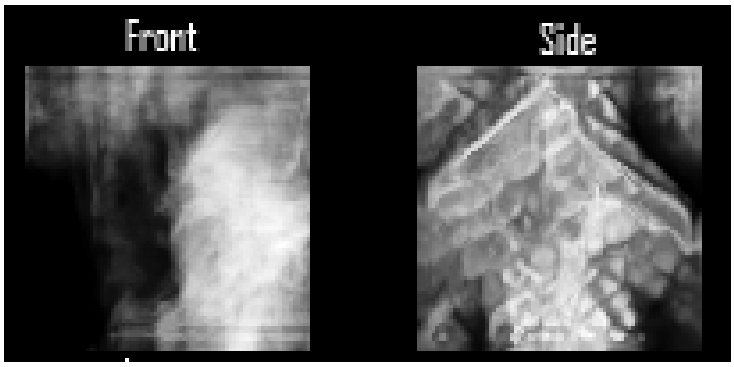
- Figure 4.33: Volcano with 1-scale style loss, velocity mask learned at training, and a window of five frames for temporal coherence.
- Figure 4.35 Dark Matter with 1-scale style loss with velocity mask at post-processing and no additional temporal post-processing. For this case an additional render is performed with increased smoke transparency (see figure 4.36).
- Figure 4.34 $\text{Spirals}_{1.0}$ with 1-scale style loss, velocity mask learned at training, and no additional temporal post-processing

The three models have been trained with patch size 64^3, then at test-time a patch size of 80^3 is used with 32 voxel overlap. The full video sequences can be found on the supplemental material: mantra_volcano.mp4, mantra_dm.mp4, mantra_dm_moretransparent.mp4, and mantra_spirals.mp4. In addition mantra_gt.mp4 contains the rendered original density field without any stylization (figure 4.37).

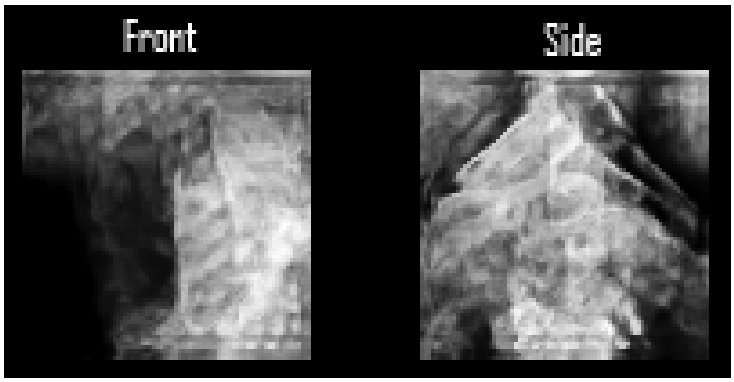
In addition we add detail and render a "dragon" sequence by up-sampling a dull low-resolution simulation of 32x120x80 voxels to 120x450x300 and processing with the patch-based approach with 80^3 patch-size and 40 voxels overlap for $\text{Spirals}_{1.0}$ and Dark Matter 1-scale-style.



(a) Using only velocities from Z-axis pass $\hat{v}^z$



(b) Using only velocities from X-axis pass $\hat{v}^x$



(c) Combine $\hat{v}^x$ and $\hat{v}^z$

Figure 4.32.: *Slice-based approach for Peace with two passes and then naively combining the velocities for a 80^3 patch*

The full sequences can be found in supplementary material files: dragon_spirals_mantra.mp4, dragon_dm_mantra.mp4, and dragon_gt_mantra.mp4 (unstylized). The last frame of the sequences is on the title-page, the frame 10 of the simulation is on figure 4.38.

4.2.10. Summary

It has been shown (on 2D) that following an architecture with a traditional bottleneck shape combined with skip-connections can be fast and yield quality results, surpassing architectures with residual layers. 3D stylizing is possible with a simple differentiable renderer and view sampling. The results show that adjusting velocity-mask, training patch-size, Tanh factors, and style image resolution, is possible to control shape and behavior of style patterns to be without spatial artifacts and often without any strong temporal incoherences as part of a patch-based approach that can process arbitrarily sized density fields. In cases that temporal incoherences are present, they can be largely removed with a fast post-processing method. Even though it is possible to a small degree to target and decrease more than one style scale loss on a more complex approach with residual velocities, it also often leads to more noisy, and prone to artifacts, results.

Figure 4.33.: *Mantra render of "Smoke Gun" with Volcano style*

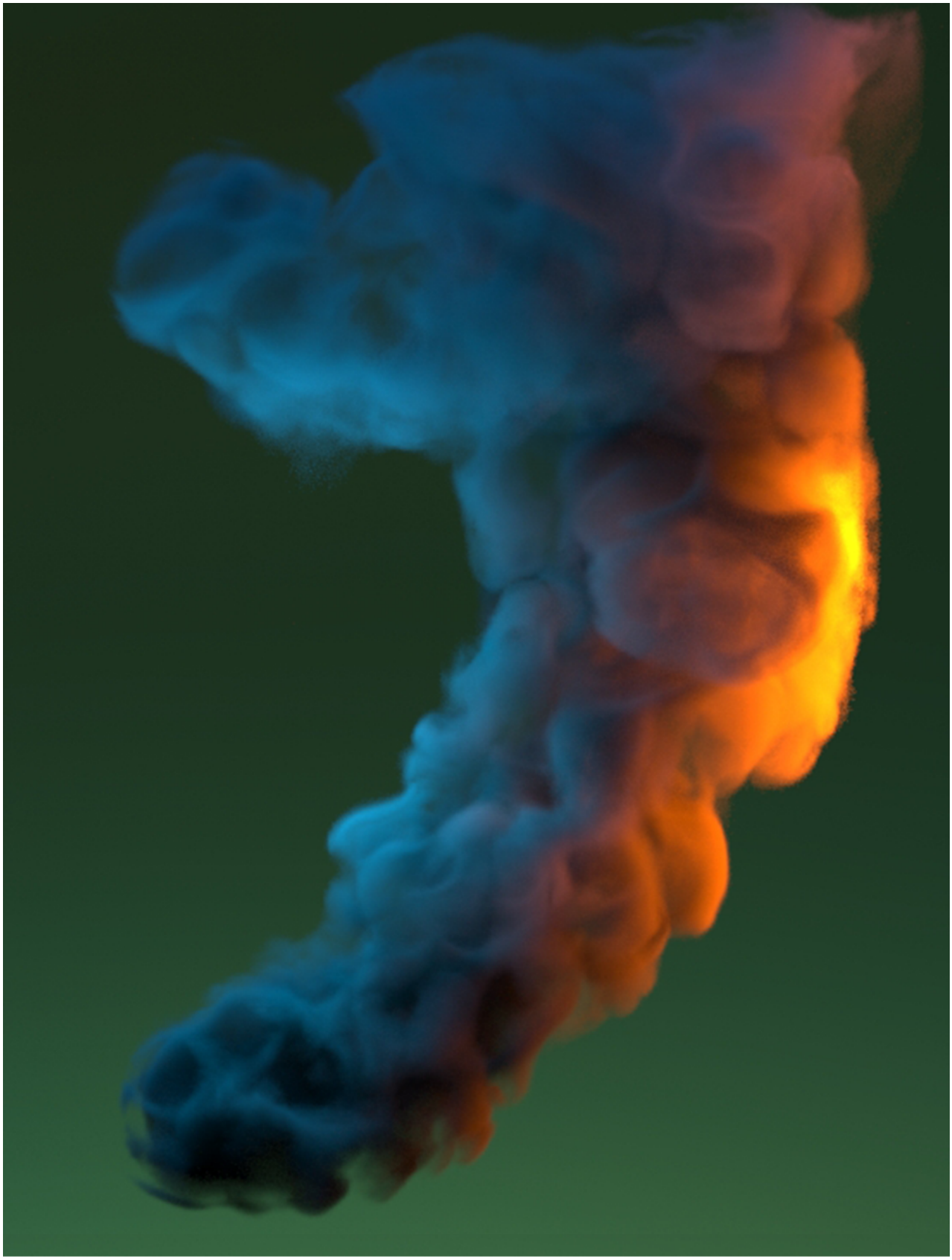


Figure 4.34.: *Mantra render of "Smoke Gun" with Spirals$_{1.0}$ style*

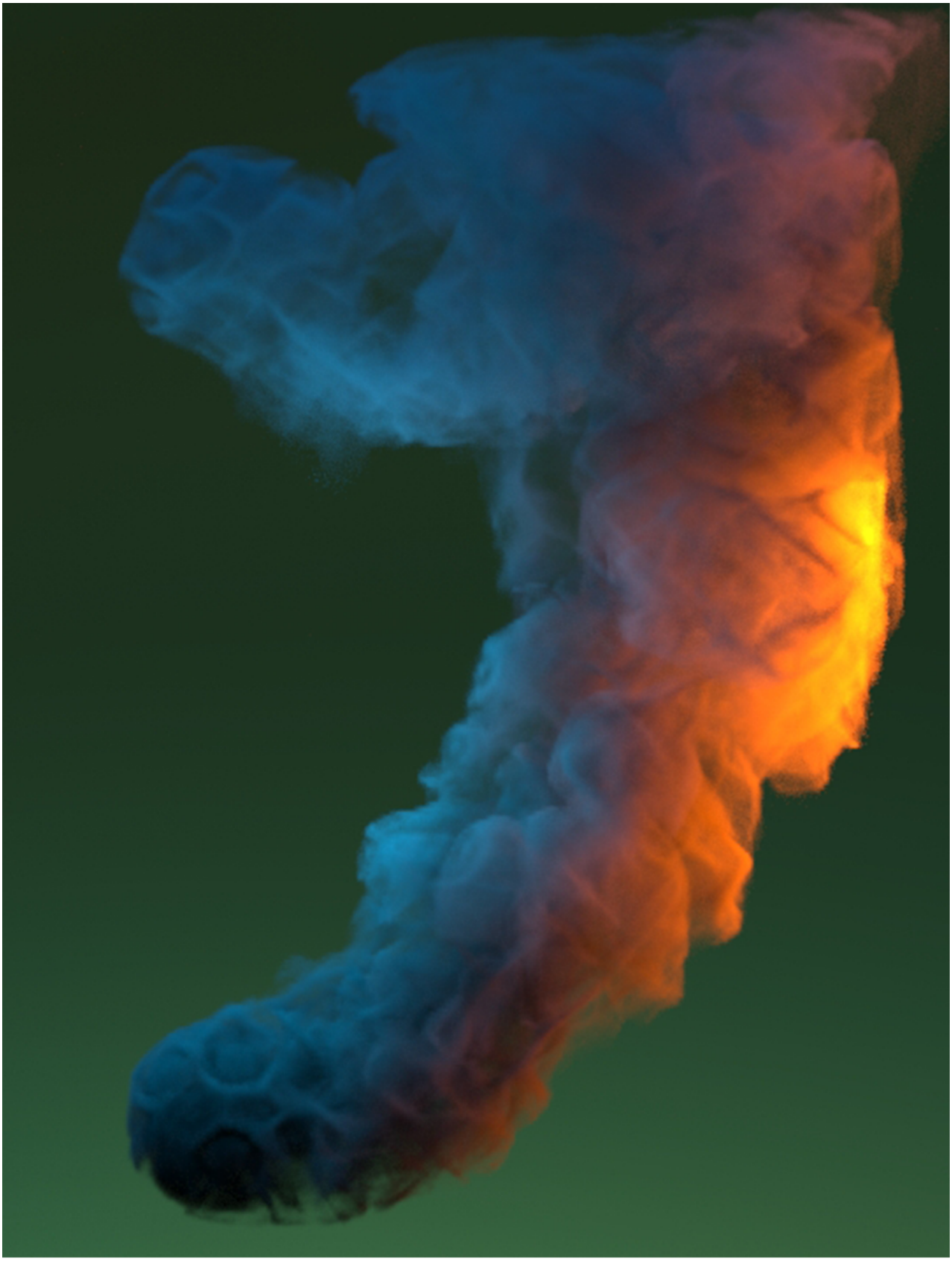


Figure 4.35.: *Mantra render of "Smoke Gun" with Dark Matter style*

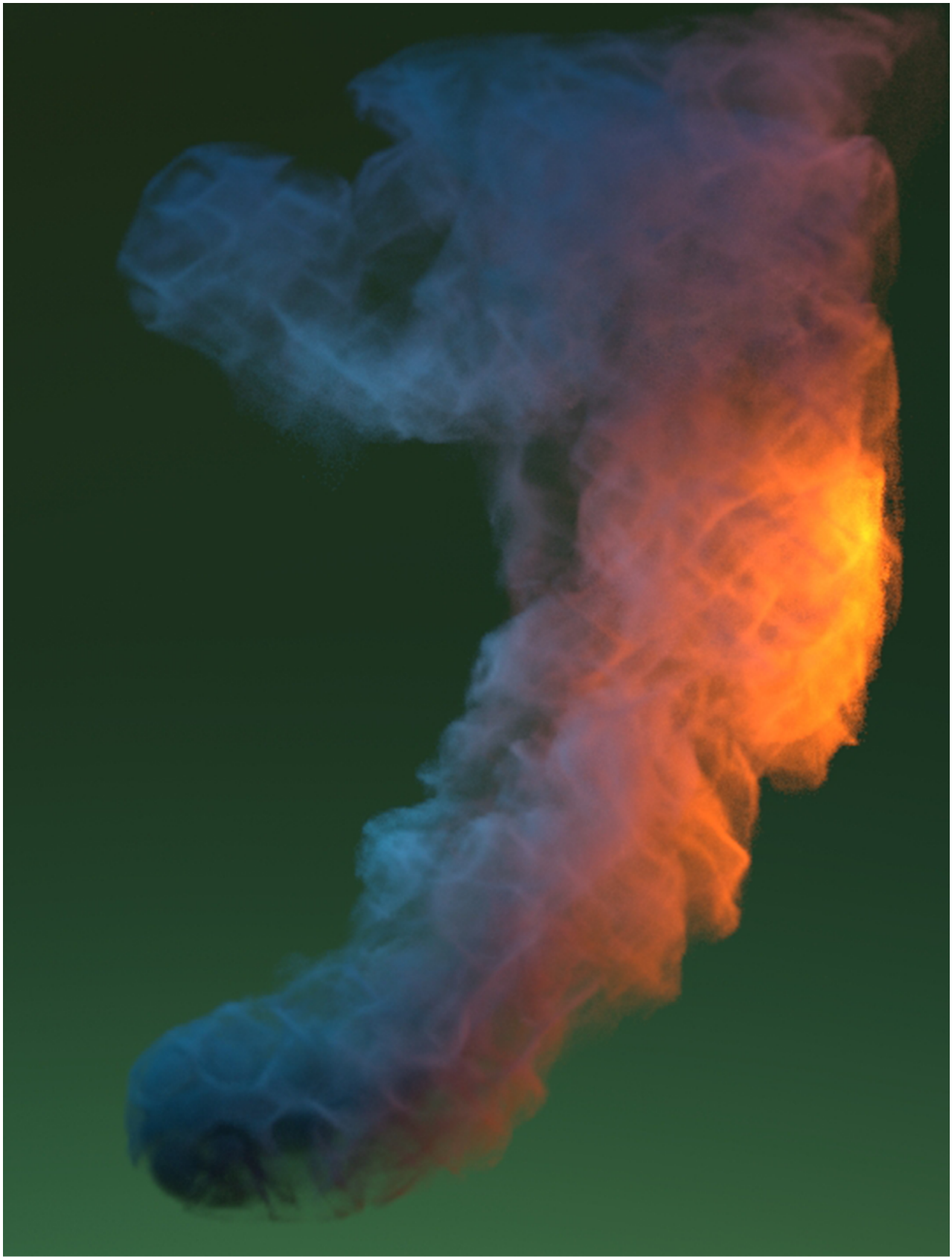


Figure 4.36.: *Mantra render of "Smoke Gun" with Dark Matter style and increased transparency*

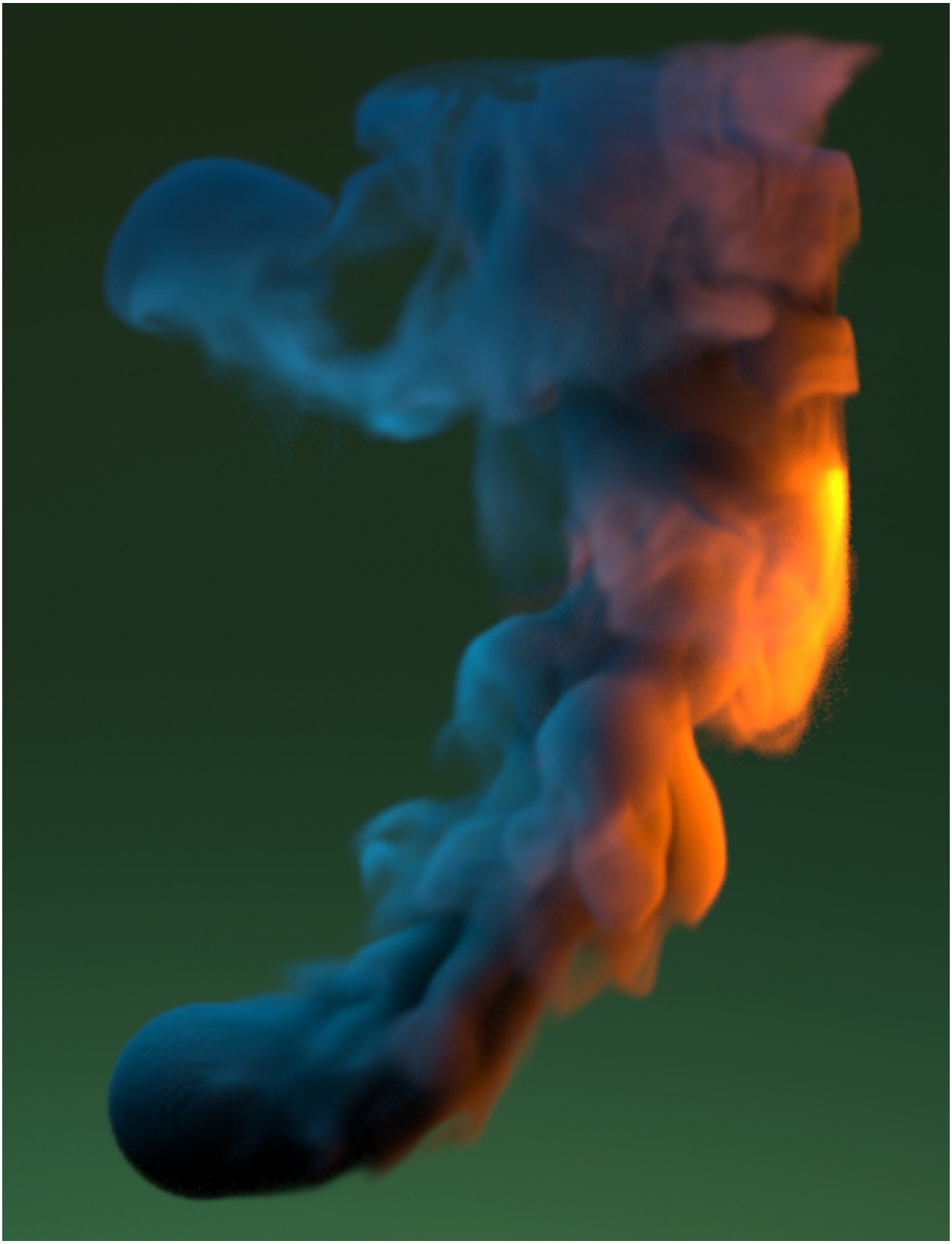


Figure 4.37.: *Mantra render of "Smoke Gun" without any stylization*

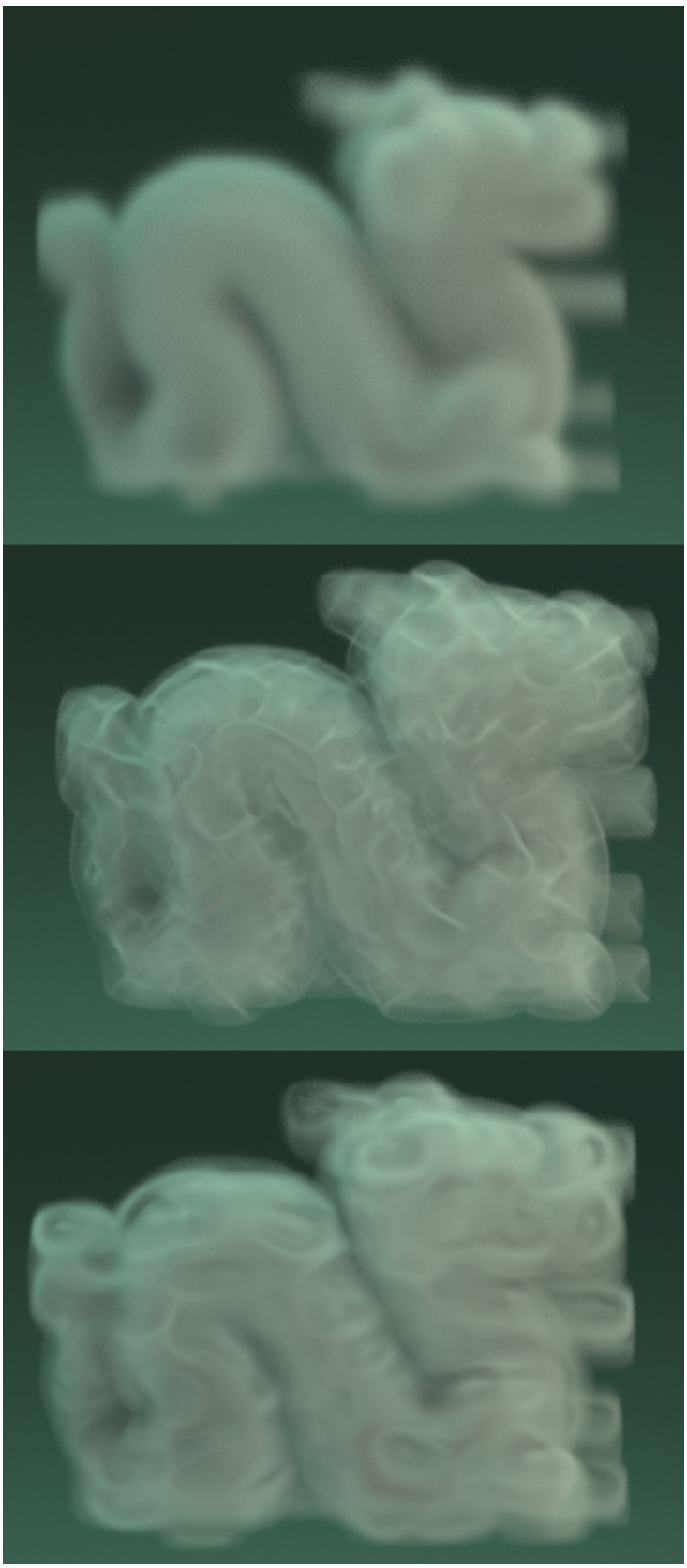


Figure 4.38.: *Mantra renders of frame 10 of "Dragon" sequence: unstylized (top), Dark Matter (middle), Spirals (Bottom)*

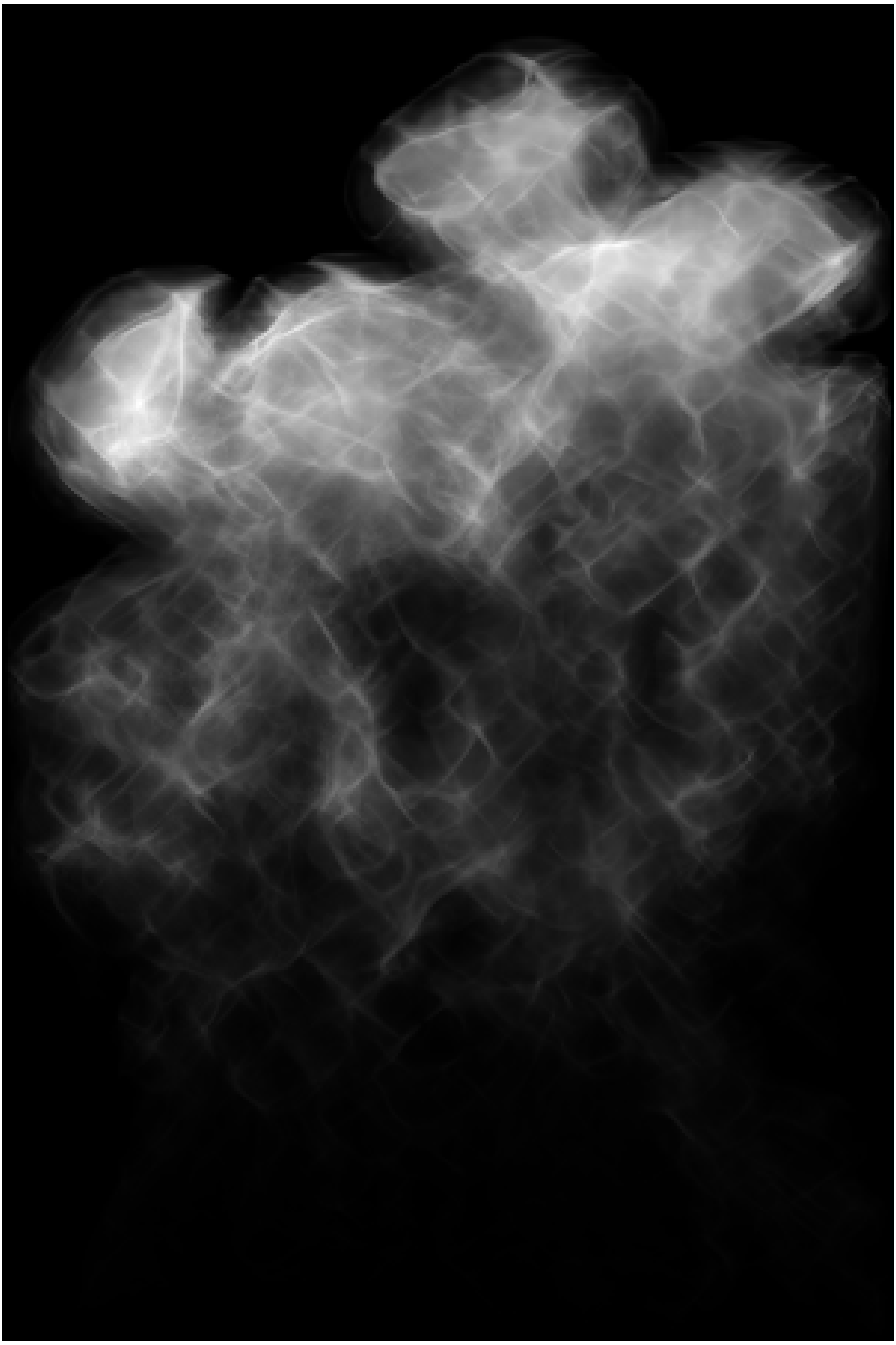


Figure 4.39.: *Last frame of "Dragon" sequence with tiny renderer for Dark Matter*

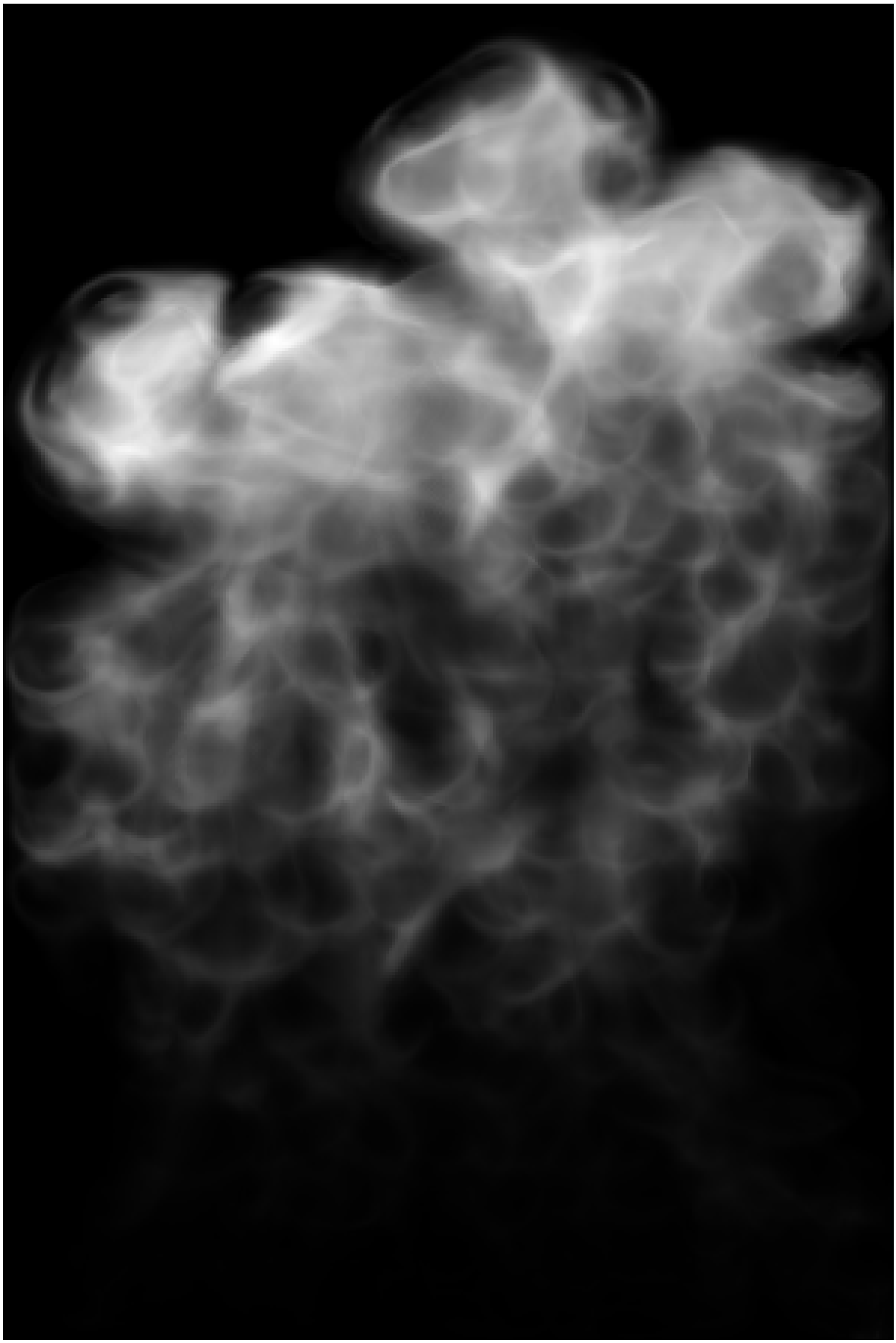


Figure 4.40.: *Last frame of "Dragon" sequence with tiny renderer for Spirals*

5

Conclusion and Outlook

This thesis work has been able to establish a baseline Pytorch implementation that is capable of stylizing of 3D fluids on a fast and spatially scalable way. Compared to iterative optimization neuronal transport method TNST [KAGS19] a stylization based on convolutional neuronal networks DCTNST exhibits a greater degree of both spatial and temporal coherence with less noise/variability between frames, facts that can be used to define a simple patch-based approach based on overlaps that is two orders of magnitudes faster. However, there is a set of limitations that needs to be mentioned. In the future work section, there is a discussion of a list of additions or changes that could solve some of the limitations or improve parts of this baseline.

The contributions of this work are:

- First feed-forward method for 3D density stylization
- First scalable approach for arbitrarily sized 3D density stylizing
- First fast temporally coherent 3D density stylization. This is possible due to inherent transport-based CNN temporal coherence and fast velocity smoothing post-process performed in a single pass.

5.1. Limitations

One important limitation of our architecture is that a separate model needs to be trained for each style image. In practice, this can be mitigated by providing artists with a diverse collection of pre-trained models. However, this still limits the freedom on which artists could experiment and tweak new style images to obtain different results.

Moreover, the tiling method used in this work needs to be carefully tuned with the CNN to avoid artifacts, otherwise, some stitching traces and temporal incoherences can be visible. There is

a relationship between the scale of the style patterns from the style image and how likely are stitch artifacts to appear. The method presented in this thesis is more suitable for more fine-grain detail styles. Stitch artifacts can be reduced by increasing test-time patch size and overlap size however this increases computational cost. Future work could focus on increasing the robustness of the patch-based approach.

5.2. Future work

- *Arbitrary Style Transfer*. In our pipeline, the same single style image is used at training time as an input to a pre-trained image classifier. To allow for arbitrary style transfer the feed-forward stylizing CNN should also take as input the style image so that it can vary its output depending on the input image style. This is more challenging for 3D CNN because the solution needs to find a strategy to combine 3D features from the density input and 2D features from the input target-style image.
- *Improved feature interpolation for stitching*. Feature interpolation shows promising results (see figures 4.27 and 4.28), however transitions between patches are still slightly noticeable. Future work could focus on improving the approach explained in section 3.3.2.
- *Neighbor Patch Priors*. On our work stitching is only performed at post-processing. However, further research could be done to find a more advanced learning scheme where a patch receives as additional inputs the velocity fields of its neighbors. The additional inputs can be used to condition the output velocity field so that it has a smooth transition with neighbors' velocity fields. In particular, a checkerboard idea could be implemented so that white cells are processed on a first pass and then black cells can be conditioned to neighbor white cells. This approach could eliminate or reduce the need for overlap, and increase the robustness of the patch-based approach.
- *Temporal Priors*. The network could take as part of the input previously stylized densities and directly minimize a temporal loss. This could reduce the need of temporal post-processing which can smooth-out part of the details.
- *Helmholtz decomposition*. The implemented CNN models directly output a stylizing velocity field but further research could focus on testing how well the neuronal network learns to instead output two separate volume fields that correspond to the incompressible and irrotational parts of the stylizing velocity field, following a Helmholtz decomposition. By doing so, the user can directly tweak the curl and divergence of the stylizing field and have additional control over the results at test-time.
- *Semantic style transfer*. This work has focused on matching the style target defined on an external image. However, instead of a style image, is possible to directly provide an array of feature maps from layers of the pre-trained network. Then a content loss based on MSE can be used to match the provided feature maps to the responses from the output density field. White noise can be used to the input of the pre-trained network to generate a set of feature maps that the user can visualize to directly choose patterns from.

- *Adversarial Training*. Using an adversarial loss has shown great results for super-resolution. Style transfer can be interpreted as a special type of super-resolution where the wanted details must match those from style image. For this reason, combining the perceptual loss with adversarial training could lead to more detailed results. [BAKS19] has shown using an approach based on GAN is able to generate impressively detailed fluid velocity fields, even though the network only mimics details which are non-physically based, on style transfer it is not necessary to be physically accurate. A future line on work could focus on adversarial training. However, it is difficult to extend a global-statistics style loss based on adversarial networks because the adversary needs to be trained to distinguish between stylized volumes from the generator and style exemplars: we are lacking reference ground truth of stylized volumes, unless those are generated synthetically, for example with the slower iterative optimization method TNST [KAGS19]. It needs to be taken into account that some of the previous GAN 2D style transfer approaches are based on collection style transfer which is less flexible because it can not stylize according to individual images. In addition, adversarial networks can be unstable to train. A feasible idea for a GAN approach is to randomly extract and compare smaller neuronal patches (from the pre-trained image classifier) both from stylized density and from the style target.

- *Super-resolution*. In addition, the output and input of our model have the same spatial resolution. In the super-resolution field, previous 2D work has shown good results combining perceptual-loss with adversarial training, and 3D work [XFCT18] using discriminator that directly compares density fields. However, there is no research that is based on comparing neuronal features for 3D density fields as part of a perceptual-loss for super-resolution. Our work shows how perceptual-loss can be propagated to a 3D volume by rendering the volume from different views. Additional research could focus on how to use the perceptual-loss, and possibly combine it with adversarial training, to perform 3D density up-scaling and compare the quality to previous 3D density super-resolution works.

- *Progressive Growing CNN Training*. A strategy based on progressively adding and training layers to the CNN could be adopted, this strategy has been especially successful in previous adversarial super-resolution works.

- *Even faster style transfer and super-resolution*. Similarly, additional research can look into how to perform more efficient stylization by combining up-sampling and style transfer: this way the input density field can be at a low-resolution version. An extension of sub-pixel up-sampling to 3D could be used to also perform convolutions at a lower spatial-resolution on the decoder. By avoiding placing convolution layers at the highest spatial resolution (output resolution) the network can have further processing speed and possibly give more quality results because a higher number of filters can be placed at the convolution blocks.

A

Appendix

A.1. Additional 2D results

For the additional results a overlap of 32 pixels is used with cropping, there is no velocity interpolation on the overlapped regions and thus the result show the inherent spatial coherence that can be introduced just with overlapping. Note also that the color mapping function uses a different scale on each image.

Model	Parameters	Style Image	Level 0	Level 1	Level 2
UNet$_{IN}$	13.4	Ben Giles	**3.36**	**4.85**	**6.06**
UNet$_{IN}$	13.4	Dark Matter	**2.65**	4.19	8.34
Jhonson'$_{128}$	1.7	Ben Giles	3.7	5.2	10.5
Jhonson'$_{128}$	1.7	Dark Matter	3.06	4.52	8.91
Jhonson'$_{256}$	6.7	Ben Giles	3.66	5.03	9.92
Jhonson'$_{256}$	6.7	Dark Matter	2.93	4.65	9.07
Jhonson'$_{384}$	14.9	Ben Giles	3.92	5.27	10.6
Jhonson'$_{384}$	14.9	Dark Matter	2.81	4.26	8.52
SRresNet$_{64}$	1.5	Ben Giles	4.02	5.37	10.84
SRresNet$_{64}$	1.5	Dark Matter	3.9	5.45	10
SRresNet$_{128}$	6.1	Ben Giles	3.61	4.97	9.86
SRresNet$_{128}$	6.1	Dark Matter	3.42	4.59	8.97
SRresNet$_{192}$	13.8	Ben Giles	3.57	4.87	9.56
SRresNet$_{192}$	13.8	Dark Matter	3.21	4.19	**8.1**
Ulyanov$_{64}$	4.4	Ben Giles	4.17	6.08	12.1
Ulyanov$_{64}$	4.4	Dark Matter	3.31	5.35	10.45

Table A.1.: *Test-time multi-scale style loss values (*1e12) for a gaussian pyramid of three levels for different 2D architectures. Parameters values are in millions. UNet model uses instance normalization. All models were trained with a single-scale style loss .*

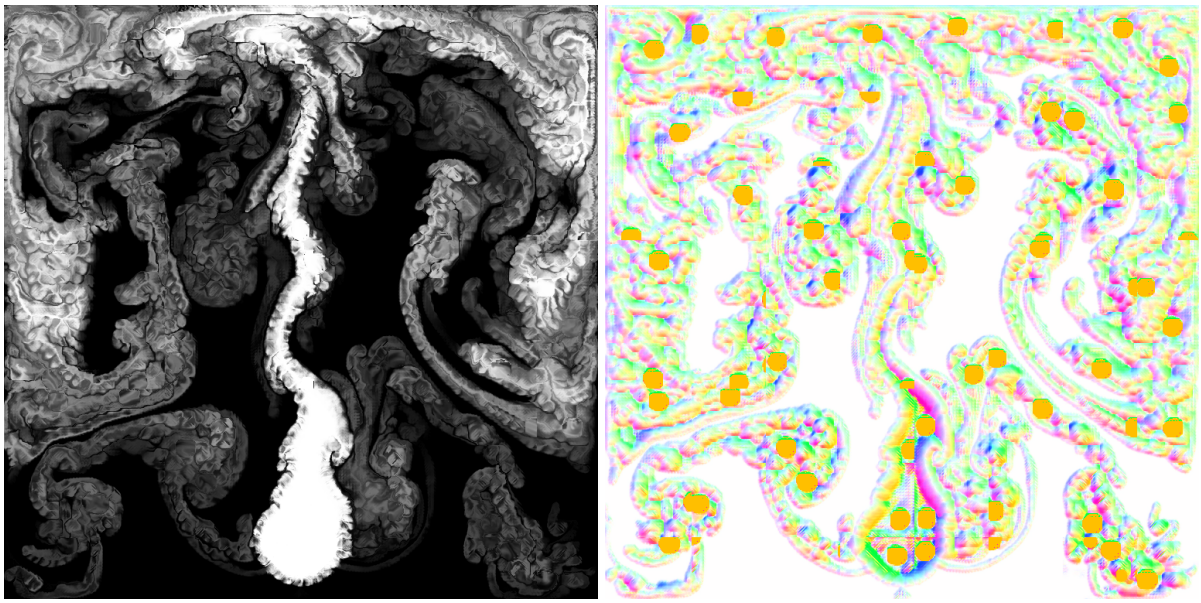


Figure A.1.: *SRResNet$_{192}$ architecture results for Ben Giles style*

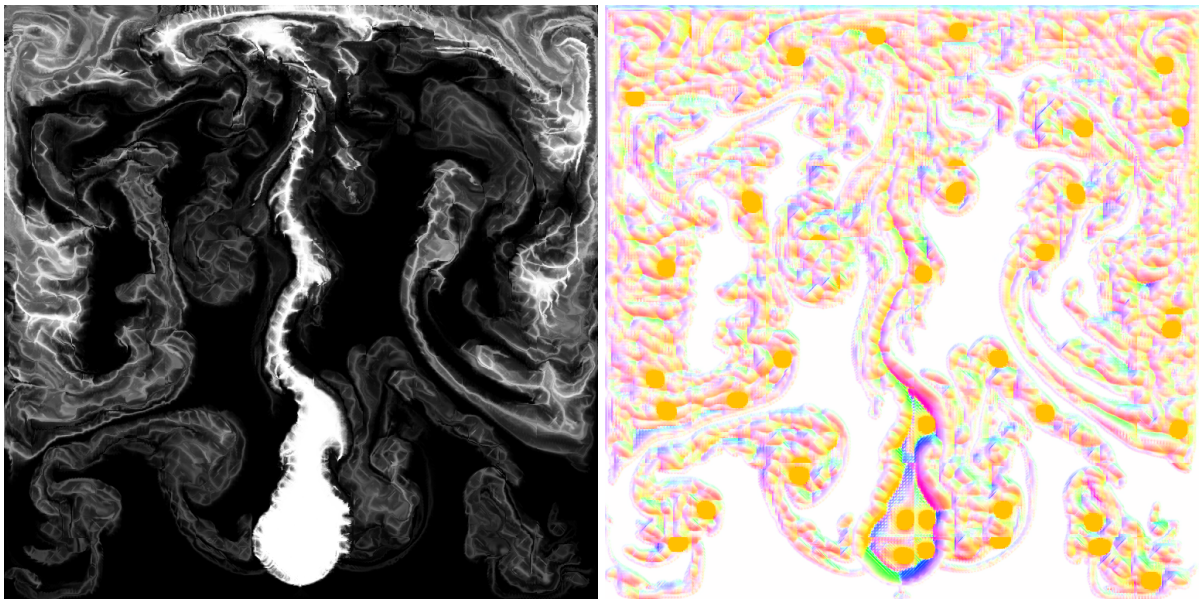


Figure A.2.: *SRResNet$_{192}$ architecture results for Dark Matter*

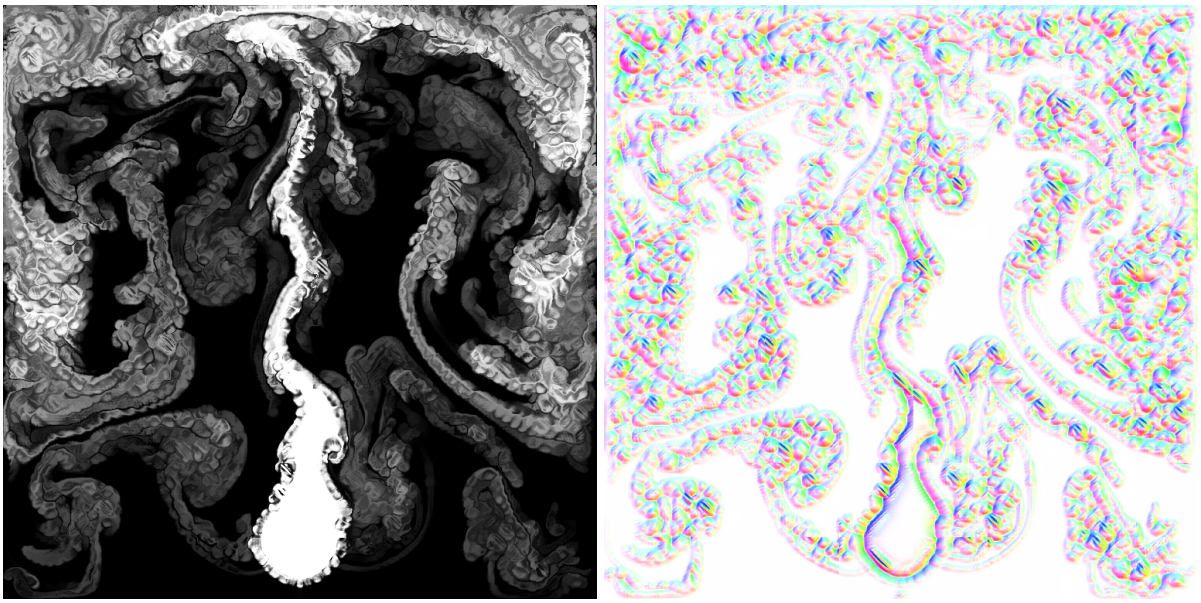


Figure A.3.: *Jhonson'$_{384}$ architecture results for Ben Giles style*

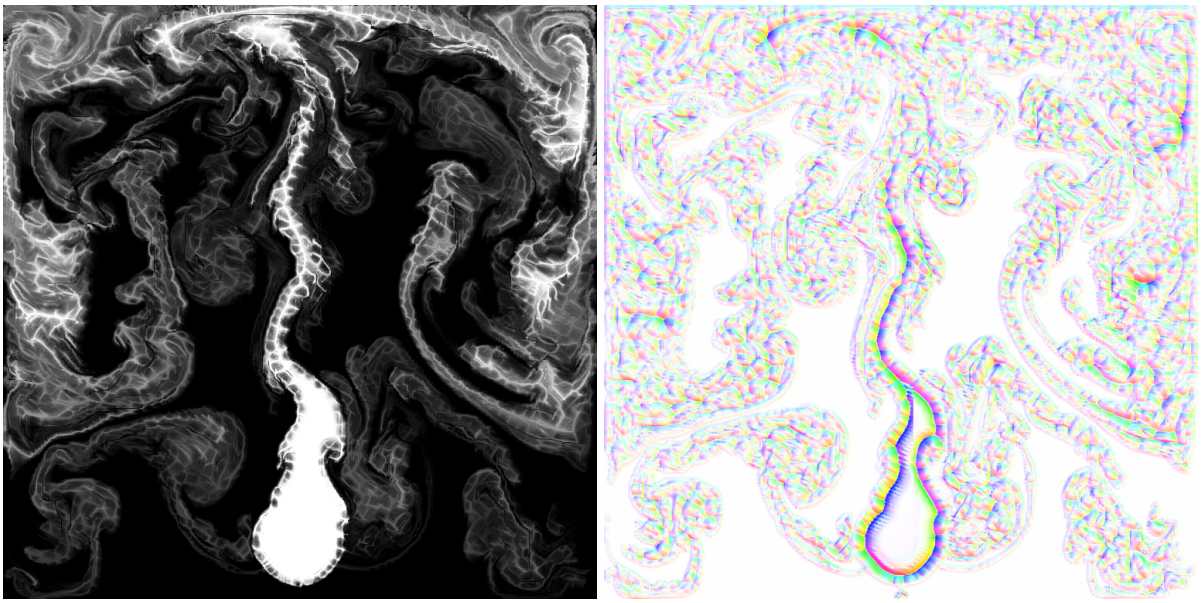


Figure A.4.: *Jhonson'$_{384}$ architecture results for Dark Matter*

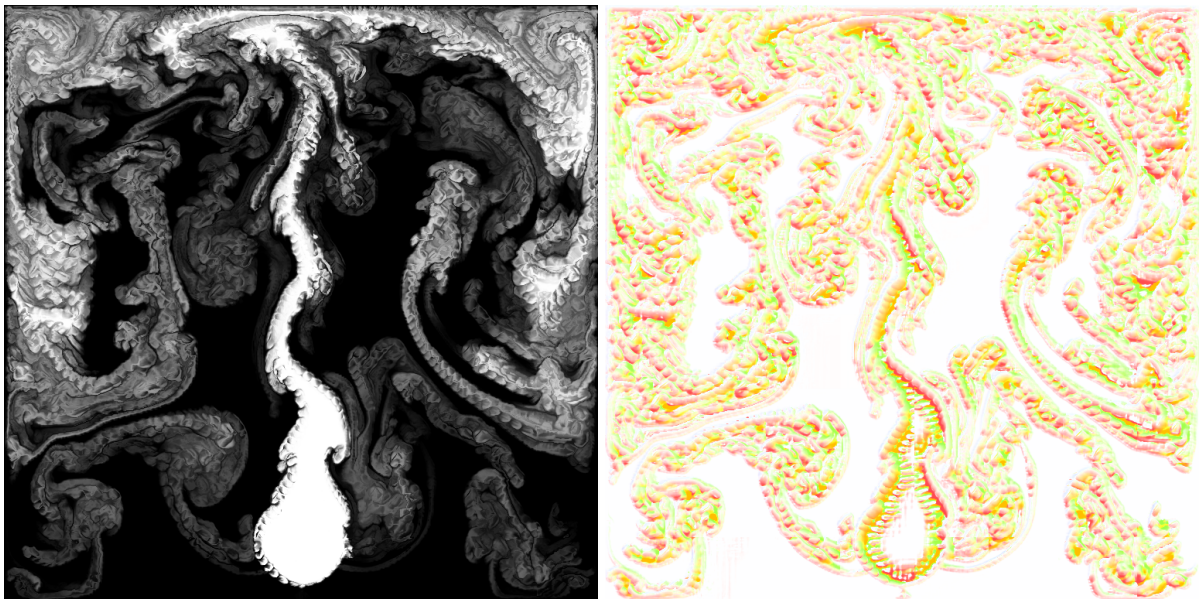


Figure A.5.: *Ulyanov$_{64}$ architecture results for Ben Giles style*

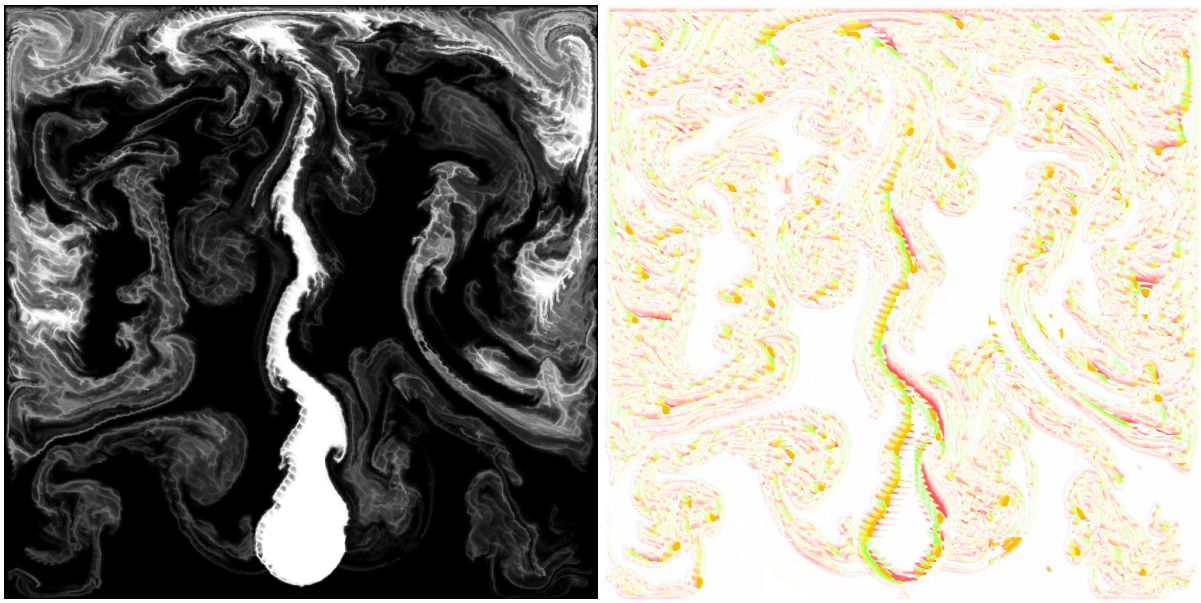


Figure A.6.: *Ulyanov$_{64}$ architecture results for Dark Matter*

A.2. Architectures Details

Table A.2.: *Customized 2D U-Net architecture with Instance Normalization*

Input	Block/Name	Layer
(d)	(0) DoubleConv	ReflectionPad2d((1, 1, 1, 1))
		Conv2d(in_channels=1, out_channels=64, kernel_size=(3, 3), stride=1)
		InstanceNorm2d(num_features=64, eps=1e-05, momentum=0.1, affine=False)
		ReLU(inplace=True)
		ReflectionPad2d((1, 1, 1, 1))
		Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3), stride=1)
		InstanceNorm2d(num_features=64, eps=1e-05, momentum=0.1, affine=False)
		ReLU(inplace=True)
(0)	(1) Down1	MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(1)	(2) DoubleConv	ReflectionPad2d((1, 1, 1, 1))
		Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), stride=1)
		InstanceNorm2d(num_features=128, eps=1e-05, momentum=0.1, affine=False)
		ReLU(inplace=True)
		ReflectionPad2d((1, 1, 1, 1))
		Conv2d(in_channels=128, out_channels=128, kernel_size=(3, 3), stride=1)
		InstanceNorm2d(num_features=128, eps=1e-05, momentum=0.1, affine=False)
		ReLU(inplace=True)
(2)	(3) Down2	MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(3)	(4) DoubleConv	ReflectionPad2d((1, 1, 1, 1))
		Conv2d(in_channels=128, out_channels=256, kernel_size=(3, 3), stride=1)
		InstanceNorm2d(num_features=256, eps=1e-05, momentum=0.1, affine=False)
		ReLU(inplace=True)
		ReflectionPad2d((1, 1, 1, 1))
		Conv2d(in_channels=256, out_channels=256, kernel_size=(3, 3), stride=1)
		InstanceNorm2d(num_features=256, eps=1e-05, momentum=0.1, affine=False)
		ReLU(inplace=True)

(4)	(5) Down3	MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(5)	(6) DoubleConv	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=256, out_channels=512, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(6)	(7) Down4	MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(7)	(8) DoubleConv	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(8)	(9) Up1	Upsample(scale_factor=2.0, mode=bilinear)
(5), (9)	(10) DoubleConv	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=1024, out_channels=256, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=256, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=256, out_channels=256, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=256, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(10)	(11) Up2	Upsample(scale_factor=2.0, mode=bilinear)

(3), (11)	(12) DoubleConv	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=512, out_channels=128, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=128, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=128, out_channels=128, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=128, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(12)	(13) Up3	Upsample(scale_factor=2.0, mode=bilinear)
(1), (13)	(14) DoubleConv	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=256, out_channels=64, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(14)	(15) Up4	Upsample(scale_factor=2.0, mode=bilinear)
(0), (15)	(16) DoubleConv	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=128, out_channels=64, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3), stride=1) InstanceNorm2d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(16)	(17) Output0	Conv2d(in_channels=64, out_channels=2, kernel_size=(1, 1), stride=1) Tanh(scale=factor_0)

Table A.3.: *Additional/Modified layers required for residual velocity computation on a customized 2D U-Net$_{+2}$ architecture with Instance Normalization*

Input	Block/Name	Layer
(14)	(18) Output1	Conv2d(in_channels=64, out_channels=2, kernel_size=(1, 1), stride=1) Tanh(scale=factor_1) Upsample(scale_factor=2.0, mode=bilinear)
(16),(18)	(17) Output0	Conv2d(in_channels=64, out_channels=2, kernel_size=(1, 1), stride=1) Tanh(scale=factor_0)

Table A.4.: *Customized 3D U-Net architecture with Instance Normalization*

Input	Block/Name	Layer
(d)	(0) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=1, out_channels=64, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=64, out_channels=64, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(0)	(1) Down1	MaxPool3d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(1)	(2) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=64, out_channels=128, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=128, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=128, out_channels=128, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=128, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(2)	(3) Down2	MaxPool3d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(3)	(4) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=128, out_channels=356, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=256, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=256, out_channels=356, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=256, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(4)	(5) Down3	MaxPool3d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(5)	(6) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=256, out_channels=512, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=512, out_channels=512, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(6)	(7) Down4	MaxPool3d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(7)	(8) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=512, out_channels=512, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=512, out_channels=512, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=512, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(8)	(9) Up1	Upsample(scale_factor=2.0, mode=trilinear)

(5), (9)	(10) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=1024, out_channels=356, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=256, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=256, out_channels=356, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=256, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(10)	(11) Up2	Upsample(scale_factor=2.0, mode=trilinear)
(3), (11)	(12) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=512, out_channels=128, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=128, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=128, out_channels=128, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=128, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(12)	(13) Up3	Upsample(scale_factor=2.0, mode=trilinear)
(1), (13)	(14) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=256, out_channels=64, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=64, out_channels=64, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(14)	(15) Up4	Upsample(scale_factor=2.0, mode=trilinear)

(0), (15)	(16) DoubleConv	ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=128, out_channels=64, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True) ReplicationPad3d((1, 1, 1, 1, 1, 1)) Conv3d(in_channels=64, out_channels=64, kernel_size=(3, 3, 3), stride=1) InstanceNorm3d(num_features=64, eps=1e-05, momentum=0.1, affine=False) ReLU(inplace=True)
(16)	(17) Output0	Conv3d(in_channels=64, out_channels=3, kernel_size=(1, 1, 1), stride=1) Tanh(scale=factor_0)

Table A.5.: *Additional/Modified layers required for residual velocity computation on a customized 3D U-Net$_{+2}$ architecture with Instance Normalization*

Input	Block/Name	Layer
(14)	(18) Output1	Conv3d(in_channels=64, out_channels=3, kernel_size=(1, 1, 1), stride=1) Tanh(scale=factor_1) Upsample(scale_factor=2.0, mode=trilinear)
(16),(18)	(17) Output0	Conv3d(in_channels=64, out_channels=3, kernel_size=(1, 1, 1), stride=1) Tanh(scale=factor_0)

Table A.6.: *Layers for the 2D residual block such as $y = f(x) + x$ for SRResNet$_{192}$*

Input	Index	Layer
(x)	(0)	ZeroPad2d((1, 1, 1, 1))
(0)	(1)	Conv2d(in_channels=192, out_channels=192, kernel_size=(3, 3), stride=1, bias=False)
(1)	(2)	BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True)
(2)	(3)	ReLU(inplace=True)
(3)	(4)	ZeroPad2d((1, 1, 1, 1))
(4)	(5)	Conv2d(in_channels=192, out_channels=192, kernel_size=(3, 3), stride=1, bias=False)
(5)	(6)	BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True)

(x),(6)	(7)	Add

Table A.7.: *Layers for customized SRResNet$_{192}$*

Input	Block/Name	Layer
(d)	(0) ConvBlock	ZeroPad2d((1, 1, 1, 1)) Conv2d(in_channels=1, out_channels=96, kernel_size=(3, 3), stride=(1, 1)) ReLU(inplace=True)
(0)	(1) Down1	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=96, out_channels=192, kernel_size=(3, 3), stride=2) ReLU(inplace=True)
(1)	(2) Down2	ReflectionPad2d((1, 1, 1, 1)) Conv2d(in_channels=192, out_channels=192, kernel_size=(3, 3), stride=2) ReLU(inplace=True)
(2)	(3) ConvBlock	ZeroPad2d((1, 1, 1, 1)) Conv2d(in_channels=192, out_channels=192, kernel_size=(3, 3), stride=(1, 1)) ReLU(inplace=True)
(3)	(4) 15x ResBlock	See table A.6
(4)	(5) ConvBlock	ZeroPad2d((1, 1, 1, 1)) Conv2d(in_channels=192, out_channels=192, kernel_size=(3, 3), stride=(1, 1)) BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True)
(3),(5)	(6) Skip-Connection	Add
(6)	(7) SubPixelConv	ZeroPad2d((1, 1, 1, 1)) Conv2d(in_channels=192, out_channels=768, kernel_size=(3, 3), stride=1, bias=False) PixelShuffle(upscale_factor=2) ReLU(inplace=True)

(7)	(8) SubPixelConv	ZeroPad2d((1, 1, 1, 1)) Conv2d(in_channels=192, out_channels=768, kernel_size=(3, 3), stride=1, bias=False) PixelShuffle(upscale_factor=2) ReLU(inplace=True)
(8)	(9) Output	ZeroPad2d((1, 1, 1, 1)) Conv2d(in_channels=192, out_channels=2, kernel_size=(3, 3), stride=1) Tanh()*factor_0

Bibliography

[ALT+17] Andrew Aitken, Christian Ledig, Lucas Theis, Jose Caballero, Zehan Wang, and Wenzhe Shi. Checkerboard artifact free sub-pixel convolution: A note on sub-pixel convolution, resize convolution and convolution resize, 2017.

[Bai19] Kai Bai. Dynamic upsampling of smoke through dictionary-based learning. 2019.

[BAKS19] Simon Biland, Vinicius C. Azevedo, Byungsoo Kim, and Barbara Solenthaler. Frequency-aware reconstruction of fluid simulations with generative networks, 2019.

[BLRW16] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. Generative and Discriminative Voxel Modeling with Convolutional Neural Networks. aug 2016.

[CLY+17] Dongdong Chen, Jing Liao, Lu Yuan, Nenghai Yu, and Gang Hua. Coherent online video style transfer. *CoRR*, abs/1703.09211, 2017.

[CS16] Tian Qi Chen and Mark Schmidt. Fast patch-based style transfer of arbitrary style. *CoRR*, abs/1612.04337, 2016.

[CT17] Mengyu Chu and Nils Thuerey. Data-driven synthesis of smoke flows with CNN-based feature descriptors. *ACM Transactions on Graphics*, 36(4):1–14, jul 2017.

[CXY+19] Xinyuan Chen, Chang Xu, Xiaokang Yang, Li Song, and Dacheng Tao. Gated-gan: Adversarial gated networks for multi-collection style transfer. *CoRR*, abs/1904.02296, 2019.

[DSK16] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. *CoRR*, abs/1610.07629, 2016.

[FAW19] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. TileGAN. *ACM Transactions on Graphics*, 38(4):1–11, jul 2019.

[GEB16] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423, June 2016.

[GJAF17] Agrim Gupta, Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Characterizing and improving stability in neural style transfer. *CoRR*, abs/1705.02092, 2017.

[GKSC20] Steven Guan, Amir A. Khan, Siddhartha Sikdar, and Parag V. Chitnis. Fully dense unet for 2-d sparse photoacoustic tomography artifact removal. *IEEE Journal of Biomedical and Health Informatics*, 24(2):568–576, Feb 2020.

[GPAM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[GTHC19] Andrew Gilbert, Matthew Trumble, Adrian Hilton, and John Collomosse. Semantic Estimation of 3D Body Shape and Pose using Minimal Cameras. aug 2019.

[HAL+19] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable Programming for Physical Simulation. oct 2019.

[HB17] Xun Huang and Serge J. Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. *CoRR*, abs/1703.06868, 2017.

[hdg] Style transfer with gans on hd images. `https://towardsdatascience.com/style-transfer-with-gans-on-hd-images-88e8efcf3716`. Accessed: 2020-03-15.

[HLS+18] Yuanming Hu, Jiancheng Liu, Andrew Spielberg, Joshua B. Tenenbaum, William T. Freeman, Jiajun Wu, Daniela Rus, and Wojciech Matusik. ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics. oct 2018.

[HLvdMW16] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2016.

[HNW+19] Xiaodan Hu, Mohamed A. Naiel, Alexander Wong, Mark Lamm, and Paul Fieguth. Runet: A robust unet architecture for image super-resolution. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[HTL18] Tak-Wai Hui, Xiaoou Tang, and Chen Change Loy. Liteflownet: A lightweight convolutional neural network for optical flow estimation, 2018.

[HWL+17] H. Huang, H. Wang, W. Luo, L. Ma, W. Jiang, X. Zhu, Z. Li, and W. Liu. Real-time neural style transfer for videos. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7044–7052, July 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[Inc] Getting Inception Architectures to Work with style transfer. https://medium.com/mlreview/getting-inception-architectures-to-work-with-style-transfer-767d53475bf8. Accessed: 2020-03-10.

[JAL16] Justin Johnson, Alexandre Alahi, and Fei-Fei Li. Perceptual losses for real-time style transfer and super-resolution. *CoRR*, abs/1603.08155, 2016.

[JBAT17] Aaron S. Jackson, Adrian Bulat, Vasileios Argyriou, and Georgios Tzimiropoulos. Large Pose 3D Face Reconstruction from a Single Image via Direct Volumetric CNN Regression. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1031–1039. IEEE, oct 2017.

[JFA+15] Ondřej Jamriška, Jakub Fišer, Paul Asente, Jingwan Lu, Eli Shechtman, and Daniel Sýkora. Lazyfluids: Appearance transfer for fluid animations. *ACM Trans. Graph.*, 34(4), July 2015.

[JLY+18] Yongcheng Jing, Yang Liu, Yezhou Yang, Zunlei Feng, Yizhou Yu, Dacheng Tao, and Mingli Song. Stroke Controllable Fast Style Transfer with Adaptive Receptive Fields. pages 244–260. 2018.

[JSZK15] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks, 2015.

[JvST+19] 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 (Proceedings of SIGGRAPH 2019)*, 38(4):107, 2019.

[JYF+19] Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, Yizhou Yu, and Mingli Song. Neural Style Transfer: A Review. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–1, 2019.

[KAGS19] Byungsoo Kim, Vinicius C. Azevedo, Markus Gross, and Barbara Solenthaler. Transport-based neural style transfer for smoke simulations. *ACM Transactions on Graphics*, 38(6):1–11, nov 2019.

[KALL18] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive Growing of {GAN}s for Improved Quality, Stability, and Variation. In *International Conference on Learning Representations*, 2018.

[KAT+18] Byungsoo Kim, Vinicius C. Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep Fluids: A Generative Network for Parameterized Fluid Simulations. jun 2018.

[KLA18] Tero Karras, Samuli Laine, and Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. dec 2018.

[KUH18] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern*

Recognition, pages 3907–3916, 2018.

[LB14] Matthew M. Loper and Michael J. Black. OpenDR: An Approximate Differentiable Renderer. pages 154–169. 2014.

[LC19] Zhengyang Lu and Ying Chen. Single image super resolution based on a modified u-net with mixed gradient loss, 2019.

[LFY+17] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. Universal style transfer via feature transforms. *CoRR*, abs/1705.08086, 2017.

[LJS+15] L'ubor Ladický, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics*, 34(6):1–9, oct 2015.

[LLKY19] Xueting Li, Sifei Liu, Jan Kautz, and Ming-Hsuan Yang. Learning Linear Transformations for Fast Image and Video Style Transfer. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[LLWZ18] Lin Li, Jian Liang, Min Weng, and Haihong Zhu. A multiple-feature reuse network to extract buildings from remote sensing imagery. *Remote Sensing*, 10:1350, 08 2018.

[LMZ18] H. Liu, P. N. Michelini, and D. Zhu. Artsy-gan: A style transfer system with improved quality, diversity and performance. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 79–84, Aug 2018.

[LPSB17] Fujun Luan, Sylvain Paris, Eli Shechtman, and Kavita Bala. Deep Photo Style Transfer. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6997–7005. IEEE, jul 2017.

[LSH+18] Jiayun Li, Karthik Sarma, King Chung Ho, Arkadiusz Gertych, Beatrice Knudsen, and Corey Arnold. A multi-scale u-net for semantic segmentation of histological images from radical prostatectomies. *AMIA ... Annual Symposium proceedings. AMIA Symposium*, 2017:1140–1148, 04 2018.

[LSS+19] Stephen Lombardi, Tomas Simon, Jason M. Saragih, Gabriel Schwartz, Andreas M. Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *CoRR*, abs/1906.07751, 2019.

[LSW+18] Guilin Liu, Kevin J. Shih, Ting-Chun Wang, Fitsum A. Reda, Karan Sapra, Zhiding Yu, Andrew Tao, and Bryan Catanzaro. Partial convolution based padding, 2018.

[LTH+16] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew P. Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. *CoRR*, abs/1609.04802, 2016.

[LTH+17] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-Realistic Single Image Super-Resolution Us-

ing a Generative Adversarial Network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 105–114. IEEE, jul 2017.

[LTJ18] Hsueh-Ti Derek Liu, Michael Tao, and Alec Jacobson. Paparazzi: Surface Editing by way of Multi-View Image Processing. *ACM Transactions on Graphics*, 2018.

[LW16a] Chuan Li and Michael Wand. Combining markov random fields and convolutional neural networks for image synthesis. *CoRR*, abs/1601.04589, 2016.

[LW16b] Chuan Li and Michael Wand. Precomputed real-time texture synthesis with markovian generative adversarial networks. *CoRR*, abs/1604.04382, 2016.

[LWLH17] Yanghao Li, Naiyan Wang, Jiaying Liu, and Xiaodi Hou. Demystifying Neural Style Transfer. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 2230–2236. AAAI Press, 2017.

[LXC+18] H. Li, X. Xu, B. Cai, K. Guo, and X. Xing. Style transfer at 100+ fps via subpixel super-resolution. In *2018 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, pages 1–6, July 2018.

[LXNC17] Shaohua Li, Xinxing Xu, Liqiang Nie, and Tat-Seng Chua. Laplacian-Steered Neural Style Transfer. In *Proceedings of the 2017 ACM on Multimedia Conference - MM '17*, pages 1716–1724, New York, New York, USA, 2017. ACM Press.

[LY19] P. Li and M. Yang. Semantic gan: Application for cross-domain image style transfer. In *2019 IEEE International Conference on Multimedia and Expo (ICME)*, pages 910–915, July 2019.

[MS15] Daniel Maturana and Sebastian Scherer. VoxNet: A 3D Convolutional Neural Network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928. IEEE, sep 2015.

[MSZM17] Roey Mechrez, Eli Shechtman, and Lihi Zelnik-Manor. Photorealistic Style Transfer with Screened Poisson Equation. In *BMVC*, 2017.

[MTC+18] David Minnen, George Toderici, Michele Covell, Troy Chinen, Nick Johnston, Joel Shor, Sung Jin Hwang, Damien Vincent, and Saurabh Singh. Spatially adaptive image compression using a tiled deep network, 2018.

[QSN+16] Charles R. Qi, Hao Su, Matthias Niessner, Angela Dai, Mengyuan Yan, and Leonidas J. Guibas. Volumetric and Multi-View CNNs for Object Classification on 3D Data. apr 2016.

[RDB16] Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. Artistic style transfer for videos. *CoRR*, abs/1604.08610, 2016.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[RWB17] Eric Risser, Pierre Wilmot, and Connelly Barnes. Stable and Controllable Neu-

ral Texture Synthesis and Style Transfer Using Histogram Losses. jan 2017.

[SED16] Ahmed Selim, Mohamed Elgharib, and Linda Doyle. Painting style transfer for head portraits using convolutional neural networks. *ACM Transactions on Graphics*, 35(4):1–18, jul 2016.

[SF18] Connor Schenck and Dieter Fox. SPNets: Differentiable Fluid Dynamics for Deep Neural Networks. jun 2018.

[SHM+18] Shunsuke Saito, Liwen Hu, Chongyang Ma, Hikaru Ibayashi, Linjie Luo, and Hao Li. 3D hair synthesis using volumetric variational autoencoders. *ACM Transactions on Graphics*, 37(6):1–12, dec 2018.

[Sne17] Xavier Snelgrove. High-resolution multi-scale neural texture synthesis. pages 1–4, 11 2017.

[STH+19] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. DeepVoxels: Learning Persistent 3D Feature Embeddings. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2019.

[SYZ17] Falong Shen, Shuicheng Yan, and Gang Zeng. Meta networks for neural style transfer. *CoRR*, abs/1709.04111, 2017.

[SZ14] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. sep 2014.

[TFL+19] Ondřej Texler, Jakub Fišer, Michal Lukáč, Jingwan Lu, Eli Shechtman, and Daniel Sýkora. Enhancing neural style transfer using patch-based synthesis. In *Proceedings of the 8th ACM/EG Expressive Symposium*, pages 43–50, 2019.

[TSSP16] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating Eulerian Fluid Simulation With Convolutional Networks. jul 2016.

[UB18] Nobuyuki Umetani and Bernd Bickel. Learning three-dimensional flow for interactive aerodynamic design. *ACM Transactions on Graphics*, 37(4):1–10, jul 2018.

[ULVL16] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor S. Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. *CoRR*, abs/1603.03417, 2016.

[UVL17] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis. *CoRR*, abs/1701.02096, 2017.

[WBT18] Steffen Wiewel, Moritz Becher, and Nils Thuerey. Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow. feb 2018.

[WOZW16] Xin Wang, Geoffrey Oxholm, Da Zhang, and Yuan-Fang Wang. Multimodal Transfer: A Hierarchical Deep Convolutional Neural Network for Fast Artistic Style Transfer. nov 2016.

[WXCT19] Maximilian Werhahn, You Xie, Mengyu Chu, and Nils Thuerey. A multi-pass

gan for fluid flow super-resolution. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 2(2):1–21, Jul 2019.

[XFCT18] You Xie, Erik Franz, Mengyu Chu, and Nils Thuerey. tempogan: A temporally coherent, volumetric GAN for super-resolution fluid flow. *CoRR*, abs/1801.09710, 2018.

[XWY19] Xiangyun Xiao, Hui Wang, and Xubo Yang. A CNN-based Flow Correction Method for Fast Preview. *Computer Graphics Forum*, 38(2):431–440, may 2019.

[YM16] M. Ersin Yumer and Niloy J. Mitra. Learning Semantic Deformation Flows with 3D Convolutional Networks. pages 294–311. 2016.

[YS18] Evan M. Yu and Mert R. Sabuncu. A Convolutional Autoencoder Approach to Learn Volumetric Shape Representations for Brain Structures. oct 2018.

[YYX16] Cheng Yang, Xubo Yang, and Xiangyun Xiao. Data-driven projection method in fluid simulation. *Computer Animation and Virtual Worlds*, 27(3-4):415–424, may 2016.

[ZGMO19] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-Attention Generative Adversarial Networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7354–7363, Long Beach, California, USA, 2019. PMLR.

[ZJL17] Lvmin Zhang, Yi Ji, and Xin Lin. Style transfer for anime sketches with enhanced residual u-net and auxiliary classifier gan, 2017.

[ZJX+18] Jiawei Zhang, Yuzhen Jin, Jilan Xu, Xiaowei Xu, and Yanchun Zhang. Mdu-net: Multi-scale densely connected u-net for biomedical image segmentation, 2018.

[ZRL+19] H. Zhao, P. L. Rosin, Y. Lai, M. Lin, and Q. Liu. Image neural style transfer with global and local optimization fusion. *IEEE Access*, 7:85573–85580, 2019.

[ZXL+19] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N. Metaxas. StackGAN++: Realistic Image Synthesis with Stacked Generative Adversarial Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(8):1947–1962, aug 2019.

[ZYC+19] Z. Zhou, Y. Yang, Z. Cai, Y. Yang, and L. Lin. Combined layer gan for image style transfer*. In *2019 IEEE International Conference on Computational Electromagnetics (ICCEM)*, pages 1–3, March 2019.

[ZYCD18] Alex Zhu, Liangzhe Yuan, Kenneth Chaney, and Kostas Daniilidis. Ev-flownet: Self-supervised optical flow estimation for event-based cameras. *Robotics: Science and Systems XIV*, Jun 2018.

[ZZB+18] Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. Non-stationary texture synthesis by adversarial expansion. *ACM Transactions on Graphics*, 37(4):1–13, jul 2018.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

EFFICIENT DEEP CNN FOR TRANSPORT-BASED NEURAL STYLE TRANSFER

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
MARTÍN BERLANGA	JESÚS

With my signature I confirm that

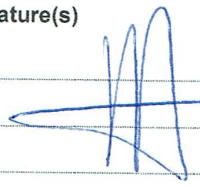
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

ZÜRICH, MARCH 14, 2020

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.