Neural Neural Textures Make Sim2Real Consistent

Ryan Burgert Jinghuan Shang Xiang Li Michael S. Ryoo
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
{rburgert,jishang,xiangli8,mryoo}@cs.stonybrook.edu

Abstract: Unpaired image translation algorithms can be used for sim2real tasks, but many fail to generate temporally consistent results. We present a new approach that combines differentiable rendering with image translation to achieve temporal consistency over indefinite timescales, using surface consistency losses and *neural neural textures*. We call this algorithm TRITON (Texture Recovering Image Translation Network): an unsupervised, end-to-end, stateless sim2real algorithm that leverages the underlying 3D geometry of input scenes by generating realistic-looking learnable neural textures. By settling on a particular texture for the objects in a scene, we ensure consistency between frames statelessly. TRITON is not limited to camera movements — it can handle the movement and deformation of objects as well, making it useful for downstream tasks such as robotic manipulation. We demonstrate the superiority of our approach both qualitatively and quantitatively, using robotic experiments and comparisons to ground truth photographs. We show that TRITON generates more useful images than other algorithms do. Please see our project website: tritonpaper.github.io

Keywords: sim2real, image translation, differentiable rendering

1 Introduction

Current sim2real image translation algorithms used for robotics [1, 2] often have difficulty generating consistent results across large time-frames particularly when the objects in an environment are allowed to move. This makes training good robotic policies using such sim2real challenging. In this paper, we discuss an algorithm called TRITON (Texture Recovering Image Translation Network) that combines neural rendering, image translation, and two special surface consistency losses to create surface-consistent translations over frames. We use the term surface-consistent to refer to the desirable quality of preserving the visual appearance of object surfaces as they move, or are viewed from another angles.

TRITON is applicable when we have a simulator capable of generating a realistic distribution of geometric data, but when we do not have any information about the surfaces of those objects (which we need to render realistic images). For example, in one of our experiments we use a robotic arm model provided by the manufacturer that contains no material information, and then learn to render the materials of the arm using TRITON. As opposed to requiring an expert/artist to create 3D models with meaningful textures, TRITON can generate these from scratch using a set of photographs capturing the distribution of states in a simulation, without requiring any matches between domains. That is, given a set of unpaired, unannotated real images and geometry images, TRITON learns the underlying texture of objects and surfaces appearing in the scene without any direct supervision.

We introduce the concept of *neural neural texture*, which is an implicit texture representation having the form of a neural network function generating RGB values given surface coordinates. Unlike previous raster-based 'neural textures' to create realistic images [3, 4], we model surface textures as a function instead of discretized pixels. It is a continuous implicit pixel-less parametric texture represented by a neural network using Fourier features [5] that takes in 2D UV coordinates and outputs colors. This is similar to how neural radiance fields (NeRF) represents a 3D scene in the form of a neural function [6]. The difference is that we focus on learning textures from unpaired training images of very different scene configurations, for viewpoint as well as object motion synthesis.

6th Conference on Robot Learning (CoRL 2022), Auckland, New Zealand.

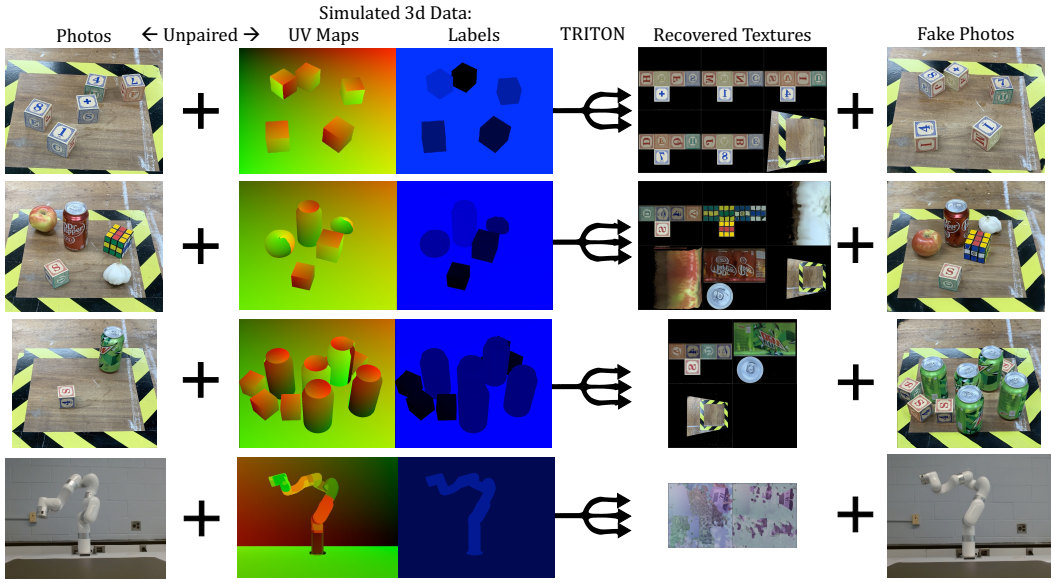

Figure 1: This is an overview of what TRITON accomplishes. TRITON learns the textures of 3D objects to help translate simulated images into photographs. It does this with unpaired data. Each row is a different dataset.

We conduct multiple experiments to confirm TRITON's advantage over prior image translation approaches commonly used for sim2real including CycleGAN [7] and CUT [8]. Importantly, we show the advantages of the proposed approach in real-robot sim2real experiments, learning the textures and training a robot policy solely based on the images generated by TRITON.

2 Related Works

Unsupervised Image to Image and Video to Video Translation Generative models have been very successful in creating images unconditionally [9, 10], creating images conditioned on text [11, 12] and creating images when trained on paired data [13]. Unpaired image translation algorithms include [14, 7, 15, 16]. Of particular interest are sim2real image to image translation algorithms such as [1, 2, 17, 18]. These algorithms aim to translate simulated images into realistic ones, for various purposes such as data augmentation or video game enhancement. In particular, Ho et al. [1], Rao et al. [2] are used to help train robots. Our paper uses an architecture based on Pfeiffer et al. [17], which is based on MUNIT [15]. Extending to video to video translation, optical flow is often used for stateful translation to improve temporal coherence [19, 20]. But they do not provide temporal coherence over long time periods.

View Synthesis NeRF [6] has spawned a large body of research on end-to-end view synthesis like [21, 22, 23]. Related to our work, which also uses geometry based backbone are [24, 25, 26]. However, all of these works only work with static scenes. There are NeRF-based approaches that work on dynamic scenes [27, 28, 29] but none of these were interactive, prohibiting their usage for sim2real. Menapace et al. [30] is interactive, but its conditioned on learned actions and not compatible with physics simulators that would be used to train robots. To learn the dynamics in the context of robot learning, self-supervised learning over time is an available approach [31, 32, 33, 34]. Differently, we only use the geometry without temporally aligned samples in those works.

Neural Textures Thies et al. [3] introduced the concept of neural textures and image translators in the context of a deferred rendering pipeline, used in other works such as [4] which is closely related to our project. Rivoir et al. [4] differs from TRITON in a few very important aspects though, the biggest being that it only synthesizes new camera views, where the only difference between each scene is the placement of the camera. In contrast, our algorithm takes in scenes where several objects are placed in different places or deformed, where one static global 3D model of the environment is not sufficient to accomplish our tasks. In addition, TRITON introduces a new type of neural texture, "neural neural texture", which is represented implicitly using a Fourier feature network [5].

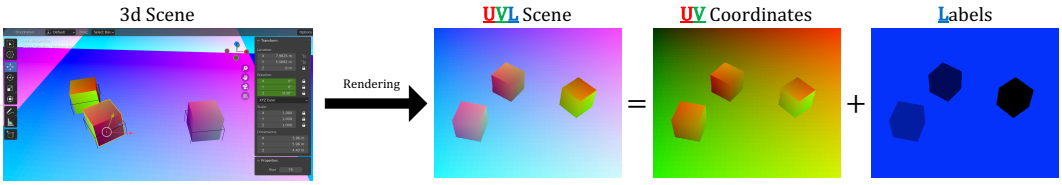

Figure 2: A scene is obtained by rendering a 3D state into an image, where the first two color channels represent u, v coordinates and the third channel l represents object labels.

3 Formulation

TRITON's goal is to turn 3D simulated images into realistic fake photographs (by training it without any matching pairs), while maintaining high surface consistency. It does this by simultaneously learning both an image translator and a set of realistic textures. These translations can be useful for downstream tasks, especially robotic policy learning from camera data, enabling sim2real.

In contrast to previous works involving neural textures [3, 4], TRITON can handle more than just camera movements: the positions of objects in the training data can be moved around or deformed as well between data samples. With high surface consistency, surfaces of translated objects will look the same even when moved around or viewed from different camera angles.

There are two components in every dataset: a set of simulated 3D scenes and a set of photographs. Datasets usually have many more scenes than photographs, since scenes can be generated automatically. A photograph is an image tensor, having (r, g, b) values between 0 and 1. We denote it as $p \in [0, 1]^{H \times W \times 3}$ where 3 refers to the three (r, g, b) channels, and H, W are the height and width.

In this work, a scene refers to a rendered 3D simulation state - which includes the position, rotation, and deformation of every object (including the camera). We represent each 3D state with a simple (r, g, b) image which we call a UVL scene, which is simple enough format that any halfway decent simulator should able to provide. A scene has the same dimensions as a photograph and is denoted as $s \in [0, 1]^{H \times W \times 3}$. However, unlike p, s's three (r, g, b) channels encode a special semantic meaning: (u, v, l) as depicted in Figure 2. The channels u and v refer to a texture coordinate, whereas l refers to a label value that identifies every object in a scene. In a given dataset, each object gets a different label. Likewise, each object is assigned a different texture. We assume that every dataset has L different label values for L different objects, and that we must learn L different neural textures.

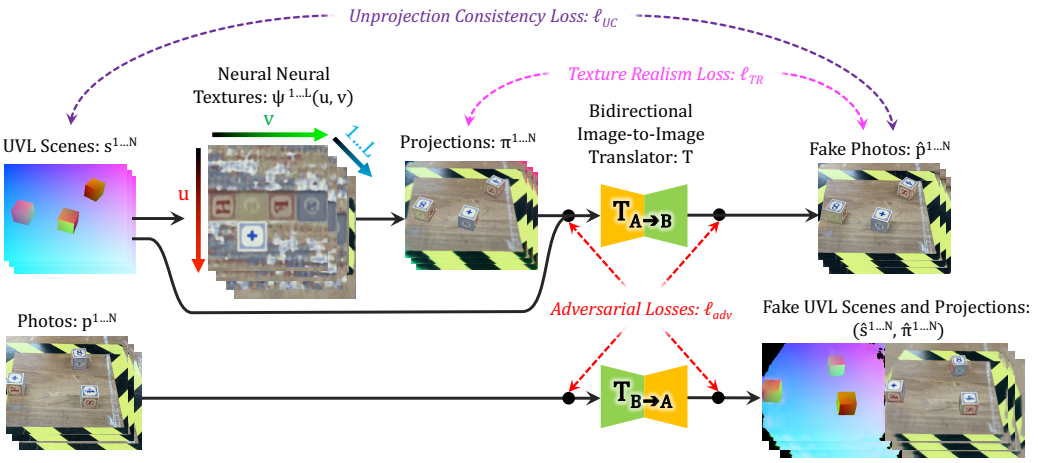

Figure 3: An overview of TRITON. Inputs are on the left, and outputs are on the right. The dashed arrows indicate losses. Although $\pi^{1\ldots N}$ and $\hat{p}^{1\ldots N}$ look similar, they are not identical - ℓ_{TR} encourages them to look as similar as possible.

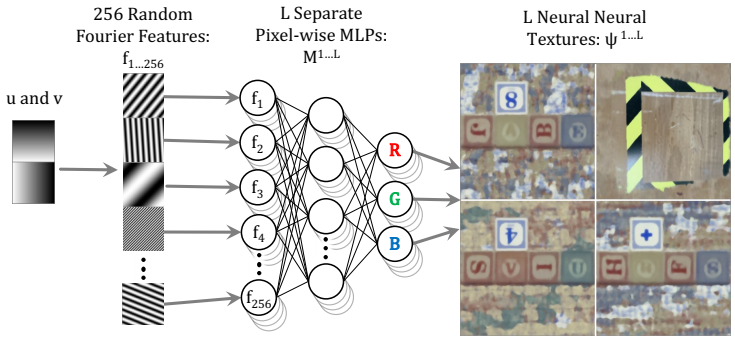

Figure 4: Details of calculating neural neural textures in Figure 3. They are fully differentiable, and represented continuously along the texture's u and v axes using Fourier feature networks.

4 Method

In this section, we describe how TRITON works to translate images from domain A (simulation) to domain B (photos) while maintaining surface consistency (Figure 3). TRITON introduces a learnable neural neural texture with two novel surface consistency losses to an existing image translator. In the end, TRITON is able to effectively generate photorealistic images.

4.1 Neural Neural Texture Projection

Instead of feeding a UVL scene s directly into the image translator $T_{A\to B}$, we first apply learnable textures ψ to the object surfaces, which is learned jointly with $T_{A\to B}$.

These neural neural textures are represented implicitly by a neural network that maps UV values to RGB values. For each texture $\psi^i \in \psi^{1\dots L}$,

$$\psi^i : (u, v) \in \mathbb{R}^2 \to (r, g, b) \in \mathbb{R}^3 \tag{1}$$

Given a UVL scene s, we obtain projection π by applying a texture $\psi^i \in \psi^{1\dots L}$ to every pixel $(u, v, l) \in s$ individually, where texture index $i = I(l)$ is decided by the label value l of that scene pixel. The projection π, which is an intermediate image is computed as:

$$\pi_{[x,y]} = \psi^i(s_{[x,y,1]}, s_{[x,y,2]}) \tag{2}$$

where $x \in \{1 \dots W\},\ y \in \{1 \dots H\}$. Texture index $i = I(l)$ where $l = s_{[x,y,3]}$ and the function $I(\cdot)$ scales and discretizes l into integers. Subscripts mean multidimensional indexing.

We now discuss the implementation of our neural neural networks $\psi^{1\dots L}$. Each neural neural texture ψ^i is a function consisting of two components: a multi-layer perceptron M^i and a static set of 256 random spatial frequencies $f_{1\dots 256} \in \mathbb{R}^{256\times 2}$ (Figure 4). Given a two-dimensional (u, v) vector, the spatial frequencies are used to generate a high dimensional embedding vector of (u, v): $(\sin(g_1), \cos(g_1), \dots, \sin(g_{256}), \cos(g_{256}))$. These embeddings are fed into M^i, where M^i is a function $\mathbb{R}^{512} \to \mathbb{R}^3$ mapping that embedding to an (r, g, b) color value:

$$\psi^i(u, v) = M^i(\sin(g_1),\ \cos(g_1),\ \sin(g_2),\ \cos(g_2),\ \dots\ \sin(g_{256}),\ \cos(g_{256})\ ) \tag{3}$$

where $g_k = f_k \cdot (u, v)$. $f_k = (\alpha_k, \beta_k)$ is a two dimensional vector, where $\alpha_k, \beta_k \in \mathcal{N}(\mu = 0,\ \sigma = 10)$ and $\cdot$ is the dot product operator. The hyperparamters 256 and 10 are selected empirically.

In practice, we found that TRITON learns faster and more stably with our neural neural textures than it does with raster neural textures. The generated textures are also smoother and less noisy. See the supplementary material for more details.

4.2 Image Translation

Our image translation module T is bidirectional: $T_{A\to B}$ translates sim to real (aka domain A to domain B), and $T_{B\to A}$ translates real to sim (aka domain B to domain A).

$$\hat{p} = T_{A\to B}(\pi, s) \quad \text{and} \quad (\hat{\pi}, \hat{s}) = T_{B\to A}(p) \tag{4}$$

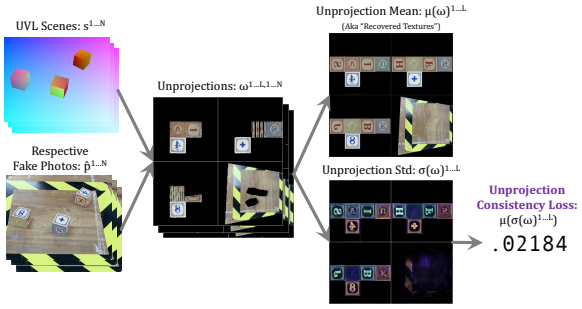

Figure 5: This is the unprojection consistency loss used in Figure 3. The Unprojection Mean is not used in any losses, but help to illustrate the content in Section 4.4.

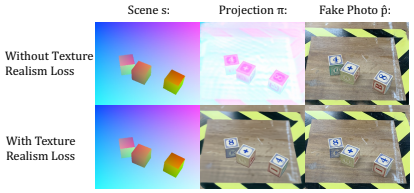

Figure 6: The neural neural texture looks more realistic with texture realism loss enabled. Note that the blocks show different letters because of different initializations and the symmetry of a cube - both solutions are valid texture assignments.

T is based on the image translation module used in Rivoir et al. [4], which is a based on a variant [17] of MUNIT [15]. T uses the same network architectures as MUNIT, and also inherits its adversarial losses ℓ_{adv}, cycle consistency losses ℓ_{cyc} and reconstruction losses ℓ_{rec}. The main differences between our image translation model and MUNIT are that we use a different image similarity loss Ω. Like Pfeiffer et al. [17] and Rivoir et al. [4] our style code is fixed (making it unimodal) and noise is injected into the latent codes during training to prevent overfitting. During inference however, this intermediate noise is removed and our image translation module is deterministic.

MUNIT translates images by encoding both domains A and B into a common latent space, then decoding them into their respective domains B and A. In our translation module (Figure 3), we define fake photos $\hat{p} = T_{A\to B}(\pi, s) = G_B(E_A(\pi, s))$ and fake projection/UVL scenes $(\hat{\pi}, \hat{s}) = T_{B\to A}(p) = G_A(E_B(p))$, where E_A, E_B, G_A, G_B are encoders and generators for domains A and B respectively.

We define an image similarity loss Ω between two images x, y that returns a score between -1 and 1, where 0 means perfect similarity:

$$\Omega(x, y) = L_2(x, y) - \text{msssim}(x, y) \tag{5}$$

This function combines mean pixel-wise L_2 distance (used in MUNIT) with multi-scale structural image similarity msssim, introduced in [35]. Note that this loss can also be applied to the latent representations obtained by E_A and E_B, because like images those tensors are also three dimensional.

We have cycle consistency loss $\ell_{cyc} = \Omega\left((\pi, s), T_{B\to A}\left(\hat{p}\right)\right) + \Omega\left(p, T_{A\to B}\left(\hat{\pi}, \hat{s}\right)\right)$, similarity loss $\ell_{rec} = \Omega\left((\pi, s), G_A\left(E_A(\pi, s)\right)\right) + \Omega\left(p, G_B\left(E_B\left(p\right)\right)\right)$ (which is effectively an autoencoder loss), and content similarity loss $\ell_{con} = \Omega\left(E_A\left(\pi, s\right), E_B\left(\hat{p}\right)\right) + \Omega\left(E_A\left(p\right), E_A\left(\hat{\pi}, \hat{s}\right)\right)$. We also have adversarial losses ℓ_{adv} that come from two discriminators D_A and D_B, targeting domains A and B respectively, using the LS-GAN loss introduced by Mao et al. [36]. In total, our image translator loss is $\ell_T = \ell_{cyc} + \ell_{rec} + \ell_{con} + \ell_{adv}$.

4.3 Unprojection Consistency Loss

To keep the object surfaces consistent, we impose a pixel-wise "unprojection consistency loss" by unprojecting surfaces in fake photos back into the common texture space. Given a UVL scene s and its respective fake photo $\hat{p}$, the unprojection ω is obtained from assigning (r, g, b) values at each pixel location (x, y) of $\hat{p}$ to its corresponding (u, v, l) coords given by s. For simplicity, we note

$$\omega^l_{[u,v]} = \mathbb{E}\,\hat{p}_{[x,y]} \quad s.t.\ (u, v, l) = s_{[x,y]}, \tag{6}$$

where the expectation $\mathbb{E}$ means we aggregate multiple (r, g, b) vectors being assigned to the same (u, v, l) coordinate by averaging them. In practice, (u, v, l) are real numbers between $[0, 1]$, where as each ω have to be rasterized, i.e., represented by L images with a size of $(D \times D \times 3)$, where L is the number of labels and $D \times D$ is the resolution of the unprojection. Therefore, we discretize and scale corresponding (u, v, l) to integers so that each (r, g, b) vector will be assigned to a pixel on ω. For the exact implementation, please see our supplementary material.

We obtain N unprojections $\{\omega^{l,i}_{[u,v]}\}_{i=1}^N$ from a batch of N UVL scenes and fake photos. The unprojection consistency loss is defined as the per pixel-channel standard deviation of ω over the batch

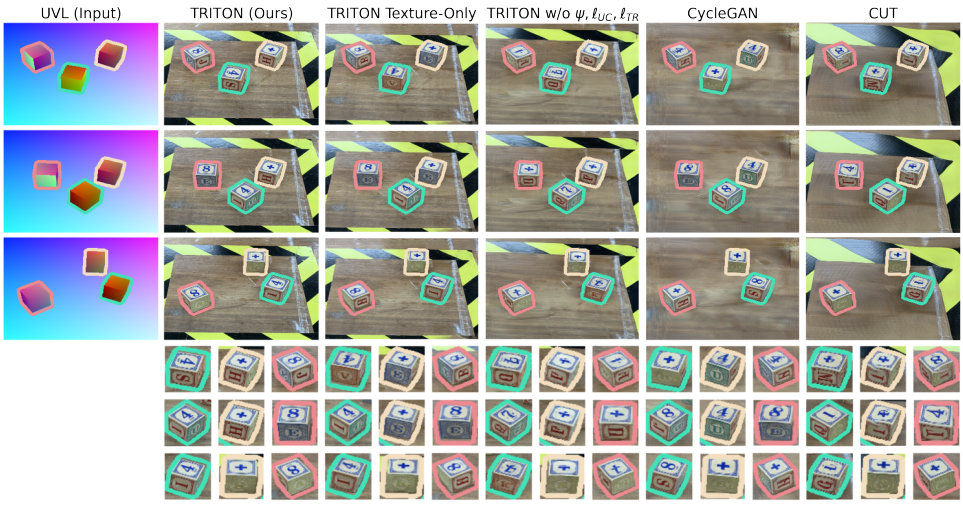

Figure 7: Image translation comparisons showing how TRITON has better temporal consistency. The first column is the UVL map as the input. The other columns are results using different image translation methods. Each row of images shows a different random placement of objects in the scene. TRITON consistently outputs high quality results over different object arrangements. "TRITON Texture-Only" stands for TRITON without two surface consistency losses ℓ_{UC} and ℓ_{TR}.

$$\ell_{UC} = \frac{1}{L \times D \times D \times 3} \sum_{l,u,v,c} \sigma(\{\omega^{l,i}_{[u,v,c]}\}_{i=1}^{N}), \tag{7}$$

where $\sigma(\cdot)$ stands for the standard deviation function and $c \in \{1 \dots 3\}$ is the channel index of ω. We minimize ℓ_{UC} to encourage unprojections to be consistent across the batch. Intuitively, if ℓ_{UC} were 0, it would mean the object surfaces in translations $\hat{p}$ appear exactly the same in every scene. In addition, we call the mean of unprojections $\mu(\{\omega^{l,i}\}_{i=1}^{N})$ as "Recovered Textures", visualzed in Figure 1 and Section 5.

4.4 Texture Realism Loss

To encourage the neural textures to look as realistic as possible, we try to make the projections look like the final output by introducing a "texture realism" loss ℓ_{TR}. ℓ_{TR} is an image similarity loss that makes π look like its translation $\hat{p}$.

$$\ell_{TR} = \Omega(\pi, \hat{p}) \tag{8}$$

Ω is the image similarity from Equation 5. Without ℓ_{TR}, ψ can look wildly different each time you train it. As seen in the top row of Figure 6, the neural texture looks very unrealistic — the colors are completely arbitrary. By adding texture realism loss ℓ_{TR}, we make the textures ψ more realistic. In practice, this makes TRITON less likely to mismatch the identity of translated objects.

5 Experiments

In this section, we perform two quantitative experiments: the first measures TRITON's accuracy directly, and the second measures it's sim2real capability in a real-world setting. *Please see the appendix for more interesting experiments!*

5.1 Datasets

We constructed two datasets AlphabetCube-3 and RobotPose-xArm to benchmark different image translators in many perspectives. In our setting, each dataset is composed of two sets of unpaired images: UVL scenes from a simulator and real photographs. Note that the images in all datasets are unpaired and UVL scenes only rely on rough 3D models of the objects in the scene without need of precisely aligning objects in the simulator to the real world. We use AlphabetCube-3 for image translation quality evaluation, and RobotPose-xArm for sim2real policy learning evaluation.

5.2 Image Translation Evaluation on AlphabetCube-3

AlphabetCube-3 dataset features a table with three different alphabet blocks on it. The dataset contains 300 real photos and 2000 UVL scenes from a simulator. Each photo features these three cubes with a random position and rotation. In this section, we evaluate the image translation accuracy of TRITON and other image translation methods using AlphabetCube-3. In this dataset, $L = 4$ because there are four different textures: three textures for the three cubes and one for the table.

Figure 7 shows qualitative results on the dataset. From Figure 7 we find that TRITON consistently outputs high quality results over different object arrangements. The textures keep aligned even when the position of blocks change dramatically over multiple scenes (See examples in green rectangles in Figure 7). In contrast, though MUNIT [15] and CycleGAN [7] manage to generate realistic images, the surfaces of the cubes are either consistent over scenes (See examples in the red rectangle) or replicated by mistake (See examples in the orange rectangle). Also note that the floor background of the outputs from CycleGAN are quite different from the ground truth. CUT [8] fails to generate meaningful images regarding both the foreground blocks and the background.

In quantitative experiments shown in Table 1, we manually align the simulator with 14 photos of different real world scenes and generate the UVL maps for translation. We measured the LPIPS [37] and $l2$-norm between the translated images and the real images using multiple configurations. 'Masked' means we mask out the background (which is the floor in AlphabetCube-3 dataset) and only measure the translation quality w.r.t three foreground blocks. For the 'unmasked' tests, we compare the whole image without any masking instead, and the background contributes much more to the losses due to its larger area. Similar to the qualitative results, TRITON consistently outperforms other methods under different metrics and configurations. Further ablations TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$ and TRITON w/o ℓ_{UC}, ℓ_{TR} show the importance of the new losses we introduce.

Table 1: Quantitative results on AlphabetCube-3. LPIPS [37] and $l2$-norm between the translated images and the real images are reported. We exclude backgrounds in the 'Masked' configuration. TRITON consistently outperforms other methods in different metrics and configurations

	Masked		Unmasked	
	LPIPS ($\times 10^{-1}$) ↓	L2 ($\times 10^{-2}$) ↓	LPIPS ($\times 10^{-1}$) ↓	L2 ($\times 10^{-2}$) ↓
CycleGAN	0.450	0.559	6.02	4.25
CUT	0.469	0.553	5.40	6.37
TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$	0.437	0.500	4.34	4.25
TRITON w/o ℓ_{UC}, ℓ_{TR}	0.442	0.586	2.85	3.64
TRITON	**0.286**	**0.479**	**1.17**	**1.23**

5.3 Sim2real Transfer for Robot Learning

In this section we demonstrate how TRITON improves sim2real transfer for robot policy learning. To this end, we first train TRITON on a dataset consist of unpaired UVL scenes from a robot simulator (Gazebo) and photos from the real robot. Once TRITON is trained, we learn the robot policy while only utilizing the simulator. The input of the policy is the fake photo $\hat{p}$ translated from the sim-

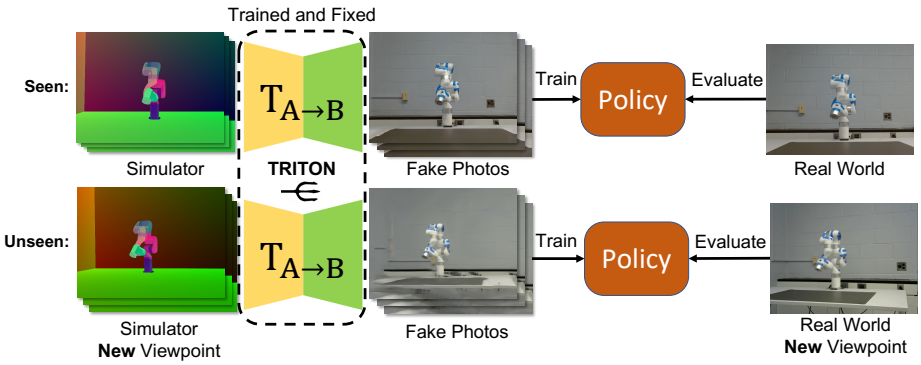

Figure 8: Sim2real framework of using TRITON. We evaluate in two settings: Seen and Unseen. In Unseen, we use a new camera viewpoint to train and evaluate the policy. This new viewpoint is not used for training TRITON.

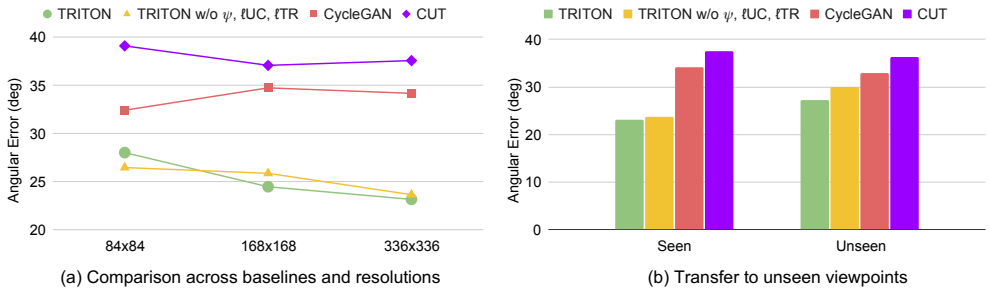

Figure 9: Evaluation results of sim2real robot learning. We compare TRITON against two baselines and one TRITON variant w/o $\psi, \ell_{UC}, \ell_{TR}$ across 3 different input image resolutions. TRITON outperforms CycleGAN [7] and CUT [8] consistently across (a) all resolutions and (b) unseen viewpoints. Results in (a) are from Seen setting only. Results in (b) are from 336x336 resolution.

ulated UVL scene using TRITON's translator $T_{A\to B}$. Finally, the trained policy is directly evaluated on the real robot, taking real photos p as input.

Ideally, a good image translation model makes $\hat{p}$ similar to p, so that the policy trained on synthesized photos will transfer to real domain seamlessly and will have better performance. All our policies are learned with zero real-world robot interaction with the environment.

Task We use a robot pose imitation task for our experiments. Given the input of a photo of real robot pose, the policy outputs robot controls to replicate that target pose. We measure the angular error between the replicated and target poses as our evaluation metric: $\sqrt{\sum_j (\hat{a}_j - a_j)^2}$, where $\hat{a}_j$ and a_j are replicated and target joint angles respectively, and j is the joint index. We use an xArm robot which has seven joints. We also attached patterns and tapes on the robot to make its texture more challenging to model. **Robot policy and Baselines** Since the sim2real formulation allows benefiting from abundant training data using the simulator, we use behavioral cloning to train a good robot policy. The policy network is a CNN similar to [38]. We compare TRITON against two baseline image translation methods: CycleGAN [7] and CUT [8], by replacing TRITON with each baseline method respectively in the above pipeline. The robot policy learning method is same for all the methods.

Evaluation Settings We introduce two main evaluation settings, **Seen** and **Unseen**. In the **Seen** setting, the policy is trained and tested using the same camera viewpoint that is used during TRITON (or baseline) training. Whereas in **Unseen** setting, we introduce a camera at a new viewpoint for training and testing the policy. That is, the image translator has to generate the fake photos out of its training distribution (seen viewpoints), which becomes a challenging task for the translator. We also vary input image resolutions to show the power of photo-realistic images in higher resolutions.

Results Figure 9 shows the evaluation results of the sim2real transfer. Both Seen and Unseen evaluations show that TRITON outperforms baselines consistently. In Figure 9(a), we find that TRITON continuously improves from low to high resolutions due to its ability to generate more photo-realistic images compared to the baselines. From Figure 9(b), we confirm that TRITON outperforms the others also in the Unseen setting, showing that TRITON learns image translations that better generalize to new viewpoints. TRITON outperforms TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$ consistently except at the lowest resolution. This again shows the effectiveness of ψ, ℓ_{UC} and ℓ_{TR} to generate photo-realistic images, and such high quality images are important in higher resolutions for sim2real transfer.

In Appendix in the supplementary material, we provide more experimental results.

6 Limitations

TRITON relies on 3D models of objects in order to generate photo-realistic images, making it less applicable when the geometry of an environment is unknown. Acquiring such 3D models could be challenging in the wild, especially in robots allowed to roam the world. TRITON is not able to handle transparent objects, because the UVL format is opaque. Generated shadows, though visually realistic, are not physically accurate. Moreover, the probability of incorrectly matching textures to objects increases as the number of object classes increases.

Acknowledgments

We thank Srijan Das and Kanchana Ranasinghe for valuable discussions. This work is supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT (No.2018-0-00205, Development of Core Technology of Robot Task-Intelligence for Improvement of Labor Condition. This work is also supported by the National Science Foundation (IIS-2104404 and CNS-2104416).

References

[1] D. Ho, K. Rao, Z. Xu, E. Jang, M. Khansari, and Y. Bai. RetinaGAN: An Object-aware Approach to Sim-to-Real Transfer. *arXiv*, 2020.

[2] K. Rao, C. Harris, A. Irpan, S. Levine, J. Ibarz, and M. Khansari. RL-CycleGAN: Reinforcement Learning Aware Simulation-To-Real. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 00:11154–11163, 2020. doi:10.1109/cvpr42600.2020.01117. RL-CycleGAN.

[3] J. Thies, M. Zollhöfer, and M. Nießner. Deferred Neural Rendering: Image Synthesis using Neural Textures. *arXiv*, 2019.

[4] D. Rivoir, M. Pfeiffer, R. Docea, F. Kolbinger, C. Riediger, J. Weitz, and S. Speidel. Long-Term Temporally Consistent Unpaired Video Translation from Simulated Surgical 3D Data. *arXiv*, 2021.

[5] M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng. Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. *arXiv*, 2020.

[6] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

[7] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *arXiv*, 2017.

[8] T. Park, A. A. Efros, R. Zhang, and J. Zhu. Contrastive learning for unpaired image-to-image translation. *CoRR*, abs/2007.15651, 2020. URL https://arxiv.org/abs/2007.15651.

[9] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks, 2014. URL https://arxiv.org/abs/1406.2661.

[10] T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks, 2018. URL https://arxiv.org/abs/1812.04948.

[11] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-shot text-to-image generation, 2021. URL https://arxiv.org/abs/2102.12092.

[12] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes, T. Salimans, J. Ho, D. J. Fleet, and M. Norouzi. Photorealistic text-to-image diffusion models with deep language understanding, 2022. URL https://arxiv.org/abs/2205.11487.

[13] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks, 2016. URL https://arxiv.org/abs/1611.07004.

[14] X. Su, J. Song, C. Meng, and S. Ermon. Dual Diffusion Implicit Bridges for Image-to-Image Translation. *arXiv*, 2022.

[15] X. Huang, M.-Y. Liu, S. Belongie, and J. Kautz. Multimodal Unsupervised Image-to-Image Translation. *arXiv*, 2018.

[16] M.-Y. Liu, T. Breuel, and J. Kautz. Unsupervised Image-to-Image Translation Networks. *arXiv*, 2017. UNIT.

[17] M. Pfeiffer, I. Funke, M. R. Robu, S. Bodenstedt, L. Strenger, S. Engelhardt, T. Roß, M. J. Clarkson, K. Gurusamy, B. R. Davidson, L. Maier-Hein, C. Riediger, T. Welsch, J. Weitz, and S. Speidel. Generating large labeled data sets for laparoscopic image processing tasks using unpaired image-to-image translation. *arXiv*, 2019.

[18] S. R. Richter, H. A. AlHaija, and V. Koltun. Enhancing photorealism enhancement. *arXiv:2105.04619*, 2021.

[19] M. Chu, Y. Xie, J. Mayer, L. Leal-Taixé, and N. Thuerey. Learning temporal coherence via self-supervision for GAN-based video generation. *ACM Transactions on Graphics (TOG)*, 39 (4):75:1–75:13, 2020. ISSN 0730-0301. doi:10.1145/3386569.3392457.

[20] L. Amsaleg, B. Huet, M. Larson, G. Gravier, H. Hung, C.-W. Ngo, W. T. Ooi, Y. Chen, Y. Pan, T. Yao, X. Tian, and T. Mei. Mocycle-GAN. *Proceedings of the 27th ACM International Conference on Multimedia*, pages 647–655, 2019. doi:10.1145/3343031.3350937.

[21] P. P. Srinivasan, B. Deng, X. Zhang, M. Tancik, B. Mildenhall, and J. T. Barron. NeRV: Neural Reflectance and Visibility Fields for Relighting and View Synthesis. *arXiv*, 2020.

[22] S. J. Garbin, M. Kowalski, M. Johnson, J. Shotton, and J. Valentin. FastNeRF: High-Fidelity Neural Rendering at 200FPS. *arXiv*, 2021.

[23] C. Lin, W. Ma, A. Torralba, and S. Lucey. BARF: bundle-adjusting neural radiance fields. *CoRR*, abs/2104.06405, 2021. URL `https://arxiv.org/abs/2104.06405`.

[24] G. Riegler and V. Koltun. Free View Synthesis. *arXiv*, 2020.

[25] G. Riegler and V. Koltun. Stable View Synthesis. *arXiv*, 2020.

[26] J. Y. Zhang, G. Yang, S. Tulsiani, and D. Ramanan. NeRS: Neural Reflectance Surfaces for Sparse-view 3D Reconstruction in the Wild. *arXiv*, 2021.

[27] A. Pumarola, E. Corona, G. Pons-Moll, and F. Moreno-Noguer. D-NeRF: Neural Radiance Fields for Dynamic Scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[28] K. Park, U. Sinha, J. T. Barron, S. Bouaziz, D. B. Goldman, S. M. Seitz, and R. Martin-Brualla. Nerfies: Deformable neural radiance fields. *ICCV*, 2021.

[29] Y. Du, Y. Zhang, H.-X. Yu, J. B. Tenenbaum, and J. Wu. Neural radiance flow for 4d view synthesis and video processing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[30] W. Menapace, S. Lathuilière, A. Siarohin, C. Theobalt, S. Tulyakov, V. Golyanik, and E. Ricci. Playable Environments: Video Manipulation in Space and Time. *arXiv*, 2022.

[31] P. Sermanet, C. Lynch, Y. Chebotar, J. Hsu, E. Jang, S. Schaal, S. Levine, and G. Brain. Time-contrastive networks: Self-supervised learning from video. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1134–1141. IEEE, 2018.

[32] J. Shang and M. S. Ryoo. Self-supervised disentangled representation learning for third-person imitation learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 214–221, 2021.

[33] J. Shang, S. Das, and M. S. Ryoo. Learning viewpoint-agnostic visual representations by recovering tokens in 3d space. In *Advances in Neural Information Processing Systems*, 2022.

[34] Y. Li, S. Li, V. Sitzmann, P. Agrawal, and A. Torralba. 3d neural scene representations for visuomotor control. In *Conference on Robot Learning*, pages 112–123. PMLR, 2022.

[35] J. Snell, K. Ridgeway, R. Liao, B. D. Roads, M. C. Mozer, and R. S. Zemel. Learning to Generate Images with Perceptual Similarity Metrics. *arXiv*, 2015.

[36] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley. Least Squares Generative Adversarial Networks. *arXiv*, 2016.

[37] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. *arXiv*, 2018.

[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. arXiv:1312.5602.

[39] X. Li, J. Shang, S. Das, and M. S. Ryoo. Does self-supervised learning really improve reinforcement learning from pixels? *arXiv preprint arXiv:2206.05266*, 2022.

[40] A. Zeng, S. Song, J. Lee, A. Rodriguez, and T. Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. 2019.

Appendix

A Algorithmic Details

A.1 Neural Neural Textures: Why use them? An Ablation

In Section 4.1, we mentioned that neural neural textures learn better than raster neural textures. TRITON can be ablated to use raster textures. Let's call this version "Raster TRITON". Raster TRITON is extremely sensitive to the resolution of its neural texture, and we found that it can be numerically unstable if that resolution is too high. In comparison, regular TRITON's neural neural textures do not have a specific resolution: they are continuously defined over the UV domain using Fourier feature networks.

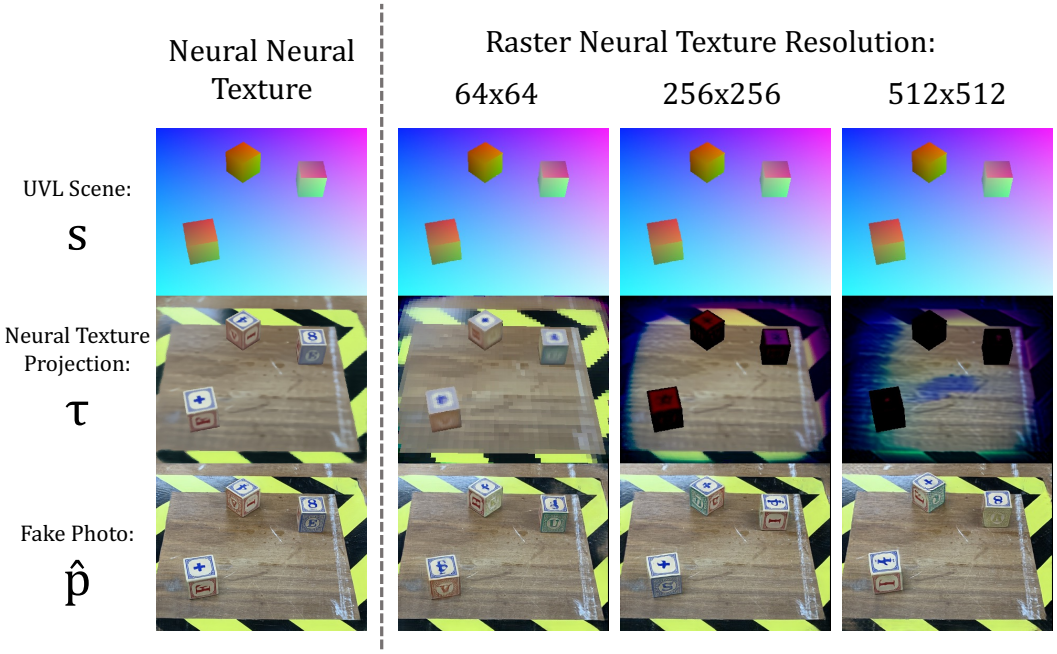

Figure 10: The resolution of the raster neural texture seriously impacts performance in "Raster TRITON". Each column has a different neural texture resolution. The neural texture only looks realistic when the resolution is low.

In Figure 10, we observe that the output quality of Raster Triton degrades as the resolution of the neural texture increases, requiring a very low resolution to give a proper output. However, a low resolution neural texture isn't capable of capturing much detail. By comparison, the neural neural texture is able to both capture a high level of detail and learn a realistic texture.

Here's our explanation for this phenomenon. With raster-based textures, only the side of an object that currently is seen in a view is allowed to change during each update, because the gradient can not be propagated into pixels which are not seen. If this means changing overall brightness of an object for example, the brightness of that object can not be changed over the entire object until every side has been seen - which will not happen in a single iteration, since the batch size is limited.

In addition to not propagating the gradient to unseen sides of objects, it also cannot propagate to texels in-between the UVL values seen in a scene image. When the raster neural texture is too large, aliasing effects occur: if you have a raster texture with very high resolution, the loss gradient is less likely to be passed to a given texel because the chance that a given UV value in a scene will be rounded to that pixel's coordinates is very small. In practice, this makes large raster neural textures unstable and limits us to using small amounts of detail. With neural neural textures, this aliasing problem is mostly mitigated, because when a certain texture receives a gradient at particular UV coordinates, the areas of the texture between those coordinates are also changed.

A.2 Unprojection Consistency Loss: More Details

Here, we give more details about unprojection consistency loss ℓ_{UC}.

The exact equation for ℓ_{UC} is as follows: to formally define ℓ_{UC} we define unprojections $\omega^{1...N,1...L}$ and mean unprojections (aka recovered textures) $\bar{\omega}^{1...L}$ with each $\omega^{n,i} \in \mathbb{R}^{3\times128\times128}$:

$$\omega^{n,i}_{c,U,V} = \mathbb{E}\,\hat{p}^{n}_{c,x,y} \quad \text{and} \quad \bar{\omega}^{i}_{c,u,v} = \frac{1}{N}\sum_{n=1}^{N}\omega^{n,i}_{c,u,v} \quad \text{and}$$

$$\ell_{UC} = \frac{\sum_{i=1}^{L}\sum_{c=1}^{3}\sum_{U=1}^{128}\sum_{V=1}^{128}\sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(\bar{\omega}^{i}_{c,u,v} - \omega^{i,n}_{c,U,V}\right)^2}}{3\cdot128\cdot128\cdot L} \tag{9}$$

where $U = \lfloor 128u \rfloor$ and $V = \lfloor 128v \rfloor$ and $i = I(l)$ with $u = s^{n}_{1,x,y}$, $v = s^{n}_{2,x,y}$, $l = s^{n}_{3,x,y}$ for all $n \in \{1...N\}$, $i \in \{1...L\}$, $x \in \{1...W\}$, $y \in \{1...H\}$, $U \in \{1...128\}$, $V \in \{1...128\}$.

The hyperparameter 128 we described in Section 4.3 refers to the resolution of the unprojections used to calculate ℓ_{UC}. In Figure 11 we see that if we were to set it higher, the gradient wouldn't affect the neural texture as densely.

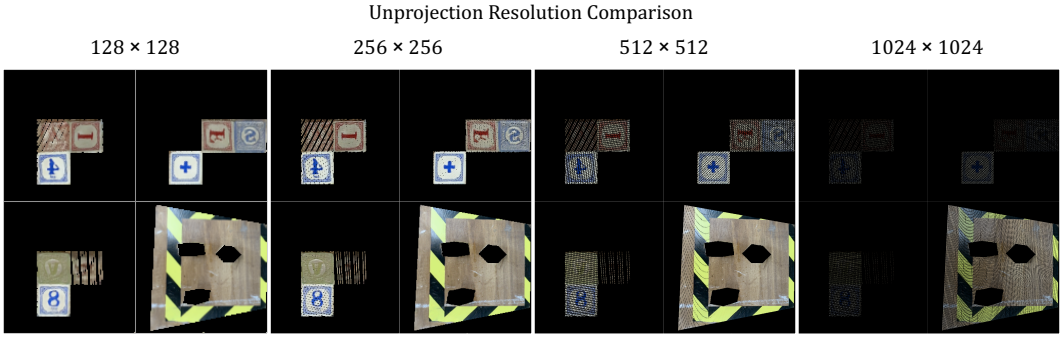

Figure 11: Here, we unproject a single fake photo $\hat{p}^{i}$. The resolution of the unprojection matters for unprojection consistency loss. The larger it is, the more precise the alignment will be but the less likely a given UV value is to be assigned a loss greater than 0, creating numerical instability and slowing down learning. When the unprojection resolution is too high, only a few areas of the neural texture will receive a gradient during backpropogation. In practice we use the resolution 128×128.

A.3 Training Procedures

In this section we detail the training procedures used to create results in Section 5 from Triton, CycleGAN, CUT, TRITON w/o ℓ_{UC}, ℓ_{TR} (aka TRITON with neither unprojection consistency loss nor texture reality loss), and TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$ (aka TRITON without textures, and therefore also without unprojection consistency loss or texture reality loss).

A.3.1 TRITON

We train TRITON for 200,000 iterations on all datasets. Each of these datasets is comprised of two sets of *unpaired* RGB images: one containing simulated UVL images, and the other containing real photographs. This takes about 36-48 hours on an NVIDIA RTX A5000. The dimension of each input scene is defined by hyperparameter height H; the width is scaled to match the aspect ratio of the original input. To avoid running out of video memory, we randomly crop each input to a 256×256 subset and run TRITON on that cropped input. We use batch size 5 during training.

We found that the best way to train TRITON is to start with smaller H values and progressively increase that value during training. For the first 50,000 iterations we use $H = 320$,

then for the next 50,000 iterations $H = 420$, then for the last 100,000 iterations $H =$ (the height of the original input scene).

Like in [4], we use three Adam optimizers: two for the translation module's encoders and decoders, and another for the neural texture. For the translation module's optimizers, we use learning rate 1×10^{-4}, and for the neural neural texture we use learning rate 1×10^{-3}. Every 100,000 iterations all learning rates are halved.

During evaluation, we render the neural neural texture onto a raster grid of pixels. Because the neural neural texture has a 2d manifold, it can be closely approximated by an RGB image. Using this method lets us avoid evlauating ψ's fourier feature network on every frame by using a raster image as a lookup table. This decreases video memory usage and speeds up calulations during inference.

A.3.2 TRITON w/o ℓ_{UC}, ℓ_{TR}

This ablation is also known as "Texture-Only" TRITON, because it has a learnable texture ψ but no consistency losses. Its trained the same way that we train TRITON normally as discussed above in A.3.1, except the surface consistency losses ℓ_{UC} and ℓ_{TR} are omitted. Effectively, it performs better than MUNIT-like TRITON (aka TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$, discussed in A.3.3)

A.3.3 TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$

This ablation is very similar to MUNIT, as TRITON is based on MUNIT and this ablation has neither learnable texture nor consistency losses. Like in subsection A.3.2, this ablation shares the same training procedures as TRITON.

A.3.4 Raster TRITON

"Raster TRITON" refers to a variant of TRITON that uses a discrete grid of pixels for its neural texture instead of a neural neural texture ψ. More details about this type of ablation are discussed in section A.1, and depicted in Figure 10. In all tests involving "Raster TRITON", we use a neural texture with a resolution of 128×128 texels.

A.3.5 CycleGAN

We use the original implementation of CycleGAN, and stick to the reccomended 200 epochs, as it tends to overfit if you go further. For our tests, we run CycleGAN on multiple different scales of the input scenes, with input sizes 286×286 (CycleGAN's default), 320×320, 420×420 and 512×512. After translation, we stretch the image into the input's aspect ratio. We also set $\lambda_{identity} = 0$, meaning we disable the identity loss. The reported results are the calculated using the best results among these resolutions. This dramatically improves CycleGAN's performance when translating UVL scenes to photographs, as this loss was built with the assumption that some parts of the input should not be changed during translation (which is not true in our case).

A.3.6 CUT

We use the original implementation of CUT, and stick to the recommended 400 epochs, as it tends to overfit if you go further. Like with CycleGAN in A.3.5, we run CUT on multiple different resolutions of the input scenes, with input sizes 286×286 (CUT's default), 320×320, 420×420 and 512×512. After translation, we stretch the image into the input's aspect ratio. The reported results are the calculated using the best results among these resolutions.

A.3.7 MUNIT

We mainly follow the official implementation of MUNIT [15], except that we use a fixed style code suggested by [4], because they demonstrated doing so yielded better temporal coherence.

B More Dataset Details

Because of the unique task of our network, we have created our own datasets. Here are some more details about various datasets used in the paper.

B.1 AlphabetCube-3

AlphabetCube-3 dataset features a table with three different alphabet blocks on it. The dataset contains 300 real photos and 2000 UVL scenes from a simulator. All photos are taken from an iPhone XS Max. Each photo features these three cubes with a random position and rotation where only the z-axis is rotated. The positions are bounded within the striped tape shown on the table. In this dataset, $L = 4$ because there are four different textures: three textures for the three cubes and one for the table.

B.2 RobotPose-xArm

We collected 891 different robot arm poses in the RobotPose-xArm dataset. Each pose was collected from 3 different camera views. The dataset was collected in pairs of simulated and real images – for evaluation purposes – resulting in 891*3 simulated images and 891*3 real images in total. In this dataset, L=2, which are the textures of the background and the robot arm.

C More Results

In this section, we highlight some more interesting results and observations from the experiments conducted in the main paper.

C.1 Unprojection Comparisons

In this section we expand on the results in Subsection 5.1 by showing the mean unprojection (aka recovered texture) for different image translation algorithms. With the 14 UVL images and corresponding fake photos (or photos) as calculated in 5.1, we average their unprojections. If a translation algorithm is consistent, these unprojections should align well and the average $\bar{\omega}$ emphould not be blurry.

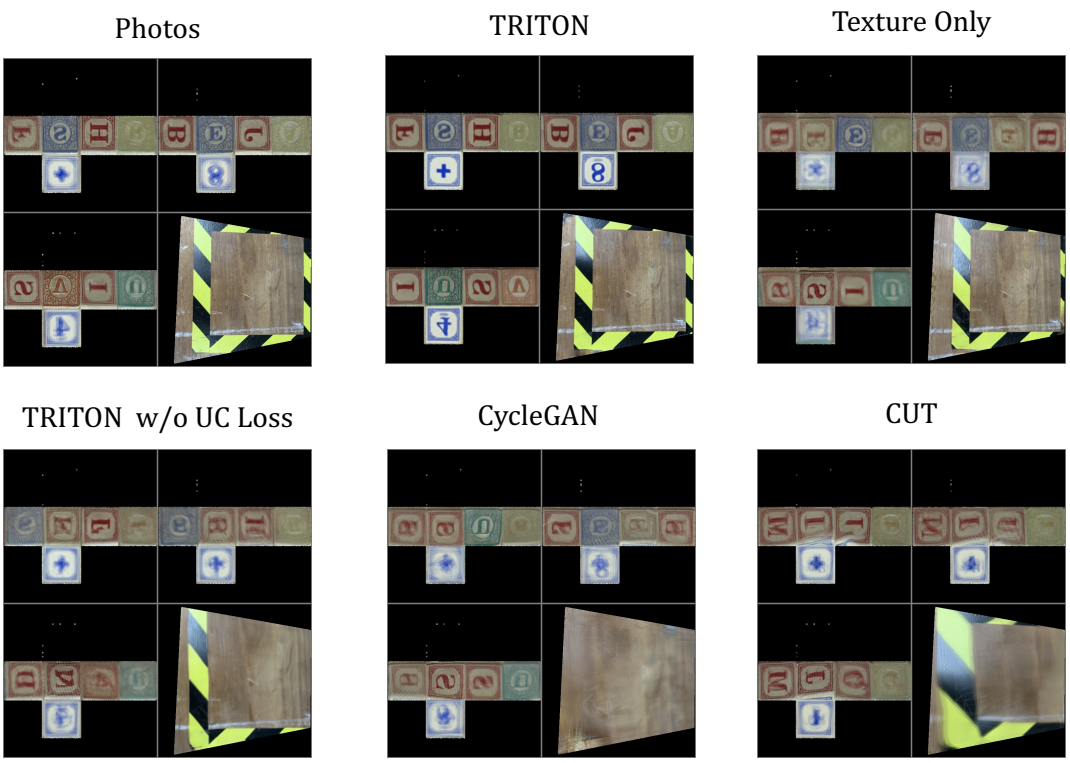

Figure 12: In this figure, we compare the recovered textures $\bar{\omega}$ (aka mean unprojections) of the 14 labeled images between different algorithms, and compare it to a ground truth. In this diagram, we align all the unprojections to make them easier to compare. Note how TRITON is more crisp and matches the unprojected photographs better than the other algorithms, which are blurrier and have the wrong colors and letters on each side of the alphabet blocks. CycleGAN and CUT for example incorrectly duplicate letters between alphabet blocks.

C.2 Videos

In this section we have links to animations that help demonstrate various aspects of TRITON. If you are viewing this document on a computer, click a thumbnail to view the video.

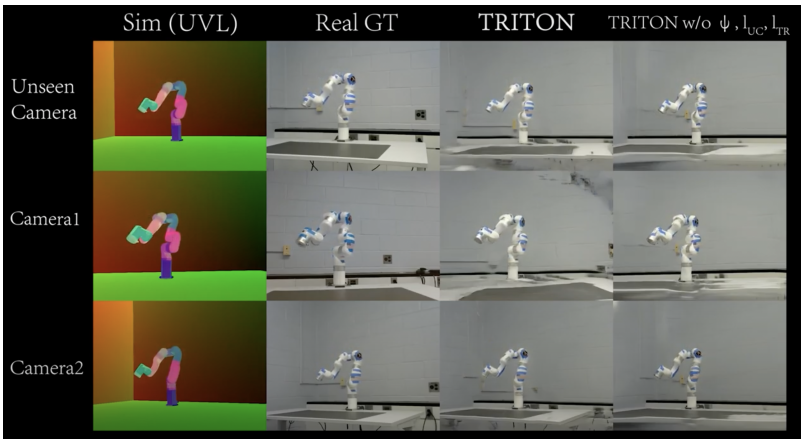

Figure 13: This animation shows a side-by-side comparison of different camera viewpoints of the results discussed in Section 5.3, as well as the inputs, ground truth video, TRITON and ablation videos. Url: youtu.be/kecK_cJgLT8

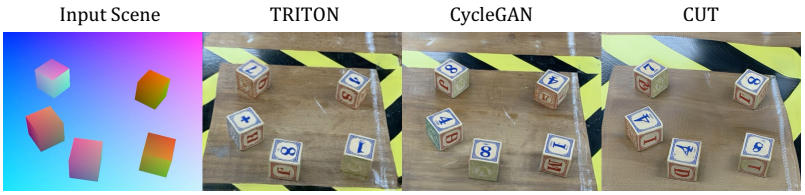

Figure 14: This animation shows an animation of various algorithms on a dataset with 5 alphabet blocks moving around. When watching, note how the numbers on the top of the cubes in both CycleGAN and CUT change over time, while with TRITON they stay the same. If you enjoy this animation, please also see Figure 17. Url: youtu.be/-GSixT4shxY

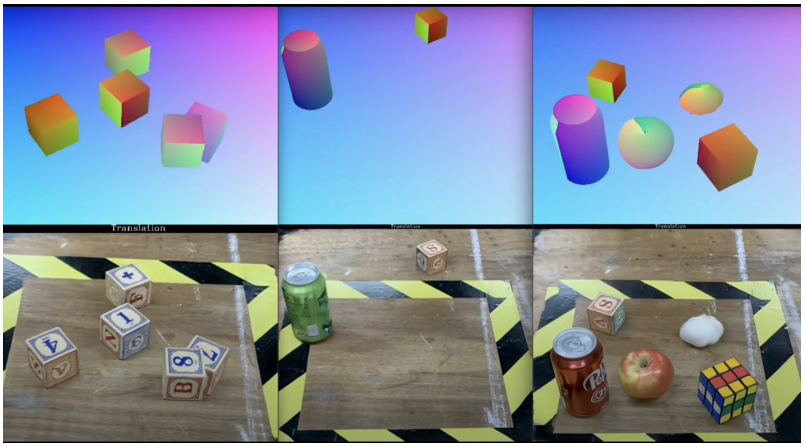

Figure 15: This video shows the results of TRITON applied on three of the datasets shown in Figure 1. The top row shows the input scenes s, and the bottom row shows the fake photographs $\hat{p}$. Url: youtu.be/0t0xiVS_8D0

D Additional Experiments

In this section of the appendix, we conduct additional experiments beyond that of the main paper. In particular, we train a robot to locate and move to objects using the RobotFive dataset, shown in section D.1. We also use synthetic dataets to measure TRITON's performance on deformable objects in section D.2 and on rigid objects in D.3.

D.1 Robot Manipulator Experiment - Reacher

Summary In this experiment, we conduct real-world robot experiments with behavioural cloning in a sim2real setting. Inspired by [39], the robot takes images from a fixed RGB camera as inputs and the goal is to localize and reach five different objects on a table. We learn the policy in a simulation using the image frames translated with TRITON (and other baselines). Then we evaluate them in the real world using actual photographs as input (See Figure 16). By evaluating the learned policies, we can compare the usefulness of TRITON with its various baselines. We find that TRITON produced the most generalizable results, and improved the accuracy of our reacher task (Table 3).

Dataset The RobotFive dataset consists of 167 photographs and 2000 UVL scenes, all of which are unpaired. It includes 5 objects that are moved around the table: a pepper, a Rubik's cube, an apple, a rubber duck and a can of soda. Since there are five objects and one table, there are a total of 6 different labels: $L = 6$. The height of the images are all 512 pixels.

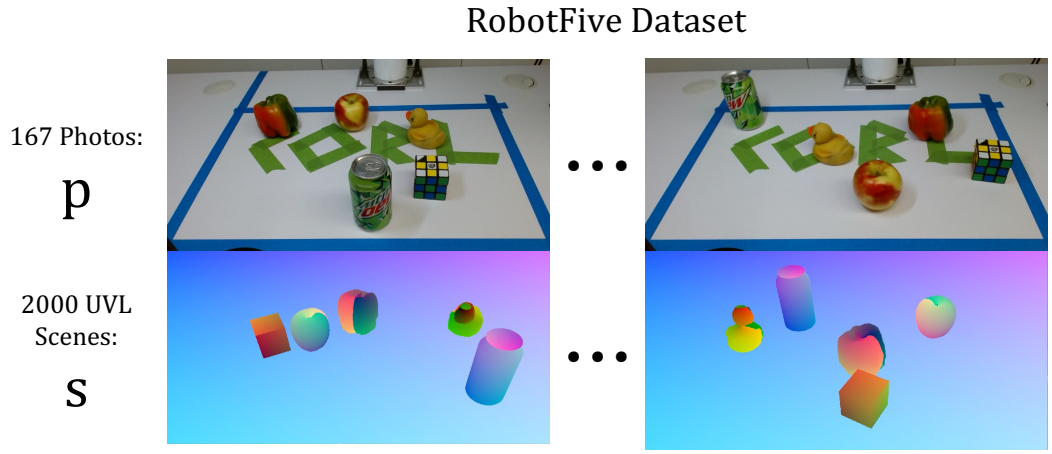

Figure 16: This figure shows some random training samples in the RobotFive dataset.

Behavioural Cloning Similar to [40], we maintain a policy network that takes one single RGB image as input and outputs a heatmap for each object simultaneously, where the values in the heatmap reflect the confidence of an object's existence. Inverse kinematics is used to drive the robot to the location with the highest value in the heat map. Our network architecture is shown in Table 2 and all the activation functions are ReLU.

For each channel of the output, we select the pixel where the target object is located as positive and randomly select other 20 pixels from the whole channel as negative samples. A cross entropy loss is applied to $1 + 20$ pixels.

The network is trained by an Adam optimizer with a learning rate of 10^{-5} and a batch size of 8 for 10 epochs and the input image is resized to 128×64 to save computation.

Quantitative Results To quantitatively evaluate the results, we manually labeled the positions of all five objects in ten real images in the world coordinates. Then we measure the distance between the outputs of the network and the ground-truth. Each translation method is trained with multiple random seeds and the mean and standard deviation of error distance are reported in Table 3. TRITON achieves best performance with a significantly lower mean error distance.

Table 2: Architecture of behavioural cloning policy.

Layer number	Type	Kernel size	Output channel	Stride	Padding
0	Conv	3	32	1	1
1	Maxpooling	3	32	2	1
2	Residual block	3	64	1	1
3	Maxpooling	3	64	2	1
4	Residual block	3	64	1	1
5	Upsampling(2x)	-	-	-	-
6	Residual block	3	32	1	1
7	Upsampling(2x)	-	-	-	-
8	Conv	1	5	1	0

Table 3: Behavioural cloning results, TRITON achieves best performance with a significant lower error distance.

Method	Mean distance ↓	Standard deviation	Number of runs
CUT	17.97	3.24	3
CycleGAN	19.26	4.68	4
MUNIT	20.77	7.18	3
Raster TRITON	16.51	1.73	4
TRITON w/o $\psi, \ell_{UC}, \ell_{TR}$	18.65	2.78	3
TRITON	**11.59**	6.51	4

Qualitative Results To visualize the outputs of TRITON and all of its baselines on this dataset, we have created an animation of those objects moving around so you can easily see the difference in temporal consistency between our algorithm and the baselines (See Figure 17). Figure 18 and its corresponding video demonstrate the robot running in the real world.

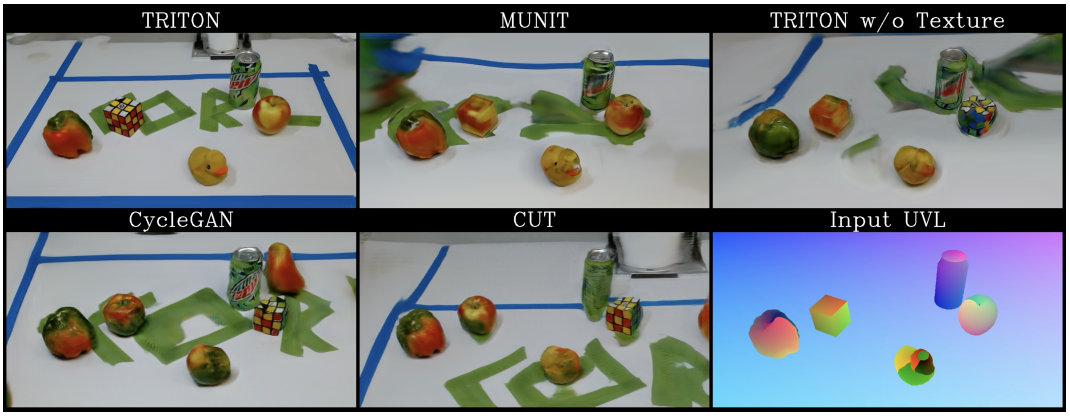

Figure 17: This animation showcases the results of TRITON and various baselines on the RobotFive dataset. To watch, click the image or visit https://youtu.be/JXNLwWoAlgQ.

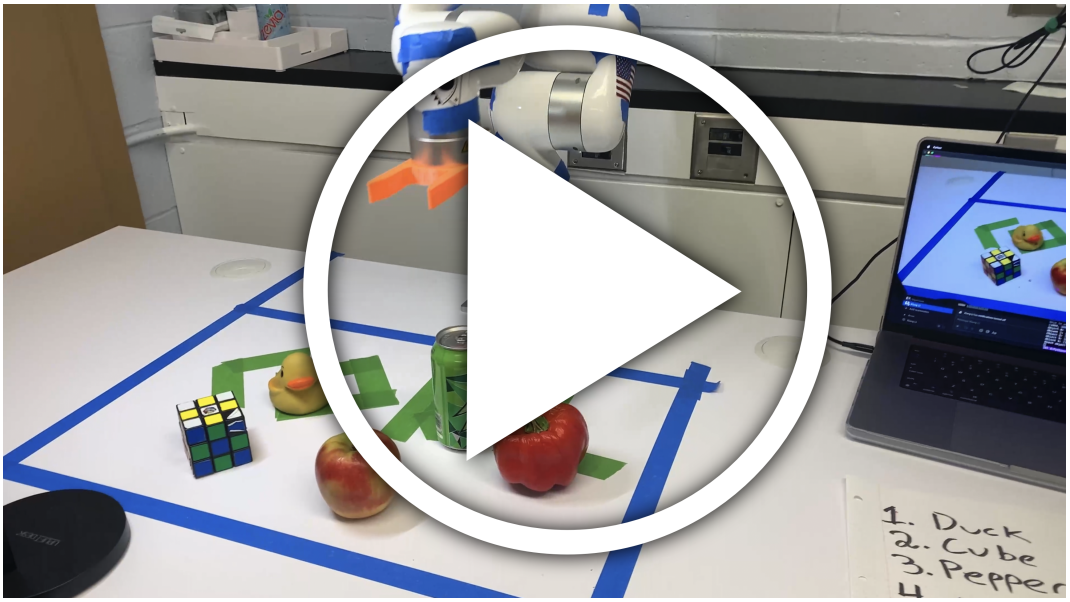

Figure 18: We've created a video of the robot arm finding various objects, using the policy learned using TRITON. To watch, click the image or visit https://youtu.be/IwAs420hY6A.

D.2 Deformable Object Experiment

Summary In this experiment, we train various algorithms to translate between synthetic UVL images and synthetic images of an American Flag blowing in the wind. This experiment demonstrates how TRITON handles deformable objects better than our baselines (MUNIT, CycleGAN and CUT) by comparing their outputs to a ground truth image.

Dataset The AmericanFlag dataset comprises of 2000 training UVL scenes, and 80 (unpaired) photographs. For the testing data, we use 2000 UVL scenes and 2000 paired ground-truth photographs. The UVL images and photographs are all generated in Blender, a 3d rendering program. We're using simulated photographs so we can easily get quantitative metrics by comparing the translated image with a ground truth. Since there is one flag and one background image, there are a total of 2 different labels: $L = 2$. The height of the images are all 512 pixels.

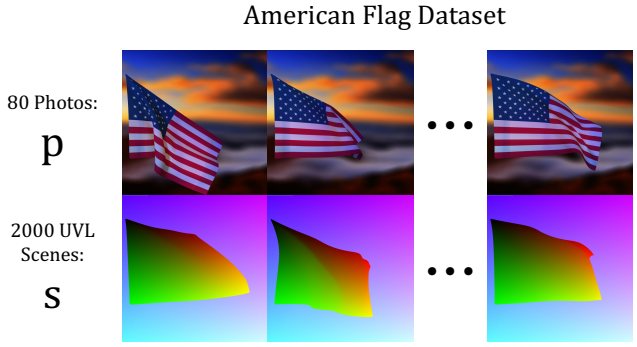

Figure 19: This figure shows some random training samples in the AmericanFlag dataset.

Results We found that TRITON vastly outperforms our baselines according to both quantitative and qualitative results. In Table 4 we show that TRITON has much better quantitative results, and

in Figure 4 we show that TRITON preserves the stripes and stars better than our other baselines. For example, MUNIT may look realistic at first glance, but at some frames it doesn't include two of the thirteen red and white stripes.

Table 4: Quantitative results on the AmericanFlag dataset. We report the average perceptual image similarity (LPIPS), multiscale structural image similarity (MSSSIM) and $l2$-norm between the translated images and the ground truth images among all samples in the test set.

	L2 $(\times 10^{-2}) \downarrow$	LPIPS $(\times 10^{-1}) \downarrow$	MSSSIM $\uparrow$
MUNIT	5.43	3.778	.5901
CycleGAN	8.58	4.833	.5600
CUT	8.72	5.737	.4736
TRITON	**0.89**	**0.722**	**.9226**

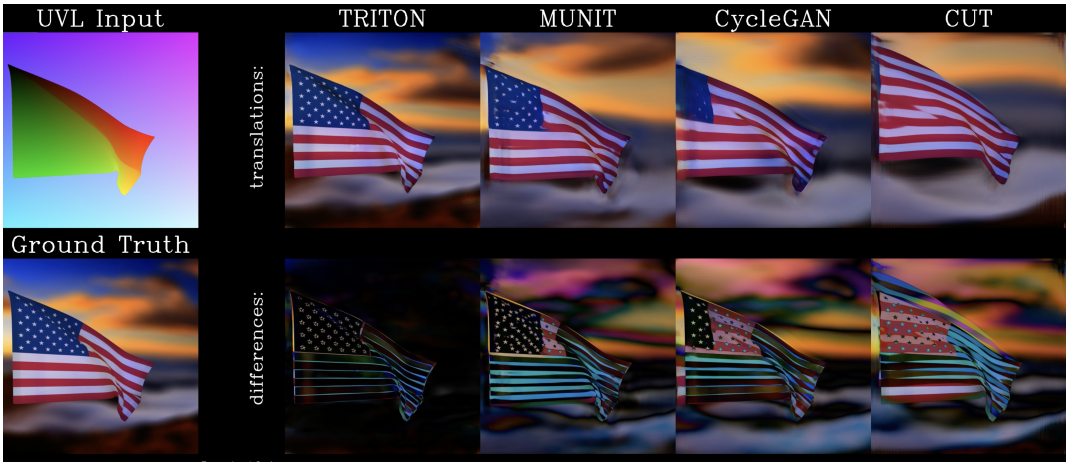

Figure 20: We have uploaded a video showing an animation of the AmericanFlag animation on all of our baselines, along with per-frame losses. Please click the above picture, or use the following link to view it: https://youtu.be/HPk4PHLaut4

D.3 Synthetic Object Experiment

Summary In this experiment, we train various algorithms to translate between synthetic UVL images and synthetic images of three rigid objects moving around a table. Because we're using a synthetic dataset, we can directly measure the accuracy of the translations across baselines. We show that TRITON outperforms the other algorithms.

Dataset The ThreeSynth training dataset comprises of 2000 training UVL scenes, and 500 (unpaired) photographs. For the testing data, we use 1795 UVL scenes and 1795 paired ground-truth photographs that were rendered in Blender. As in the AmericanFlag dataset seen in Figure 19, we use simulated photographs so we can easily evaluate accuracy by comparing the translated image with a ground truth. Because the ThreeSynth dataset uses asymmetrical objects unlike the AlphabetCube-3 dataset, we do not need to match through many permutations to measure the results. Since there are three objects and one table, there are a total of 4 different labels: $L = 4$. The height of the images are all 512 pixels.

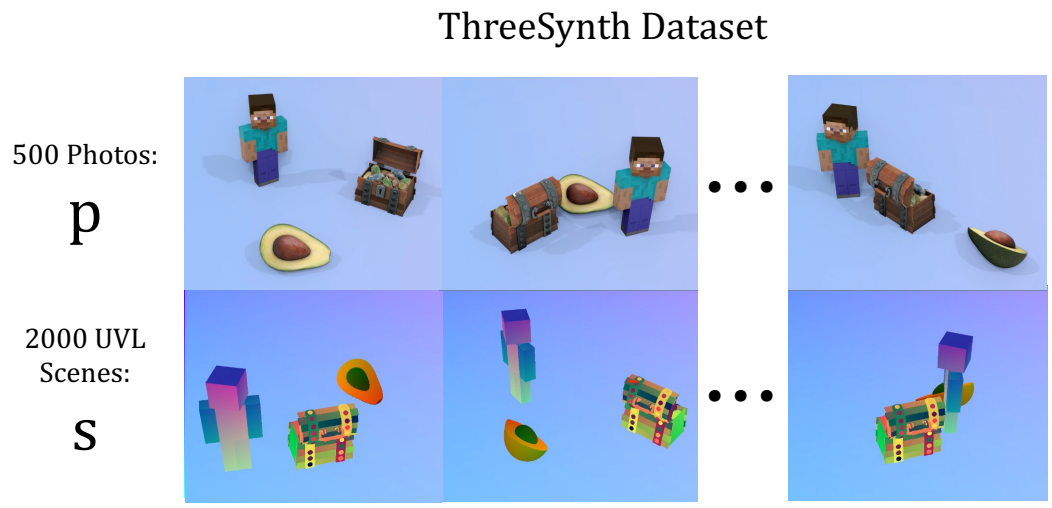

Figure 21: This figure shows some random training samples in the ThreeSynth dataset. It includes three obejcts: A half-avocado, a treasure chest, and model of Steve from the game "Minecraft".

Results We found that TRITON outperforms our baselines according to both quantitative and qualitative results. In Table 5 we show that TRITON has better quantitative results, and in Figure 22 we provide an animation of the objects moving. We observe that MUNIT distorts Steve's face.

Table 5: Quantitative results on the ThreeSynth dataset. We report the average perceptual image similarity (LPIPS), multiscale structural image similarity (MSSSIM) and $l2$-norm between the translated images and the ground truth images among all samples in the test set.

	L2 $(\times 10^{-3}) \downarrow$	LPIPS $(\times 10^{-1}) \downarrow$	MSSSIM $\uparrow$
CUT	4.07	0.8640	.91419
CycleGAN	2.62	0.7979	.93345
MUNIT	1.59	0.6754	.95468
Raster TRITON	4.01	1.4049	.95571
TRITON	**1.19**	**0.6021**	**.96221**

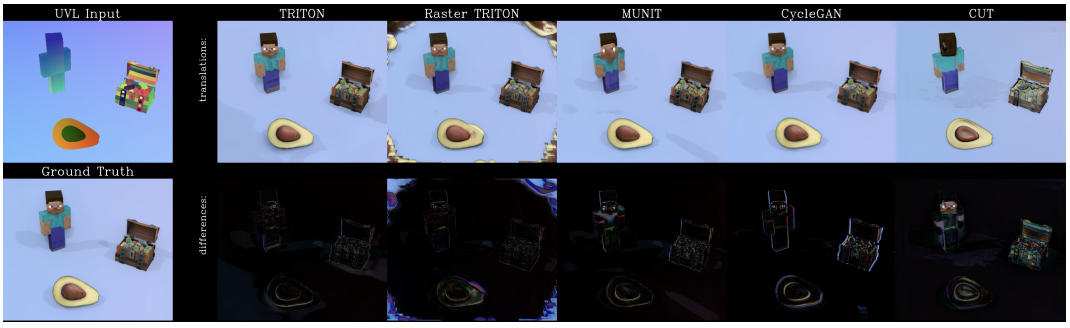

Figure 22: This figure shows an example of the translations on the ThreeSynth dataset. Click the image to view an animation of this analysis for three different datasets, or go to https://youtu.be/p6Mt0180Pvw.

E Curiosities

In the curiousities section, we show what happens when you apply the wrong textures to the wrong objects, and show a way to isolate the shadows generated by TRITON. We've decided to include them in the appendix simply because they are interesting.

E.1 Swapping Neural Textures

What happens when you apply the wrong textures to the wrong objects?

Below, we show what happens when you swap these textures *after* training. Since the neural textures are evolved per-object, swapping labels during inference will usually not result in realistic translations. This is because the UV mapping for the neural textures will no longer match the geometries of the respective objects. Below we show an example of this using the RobotFive dataset.

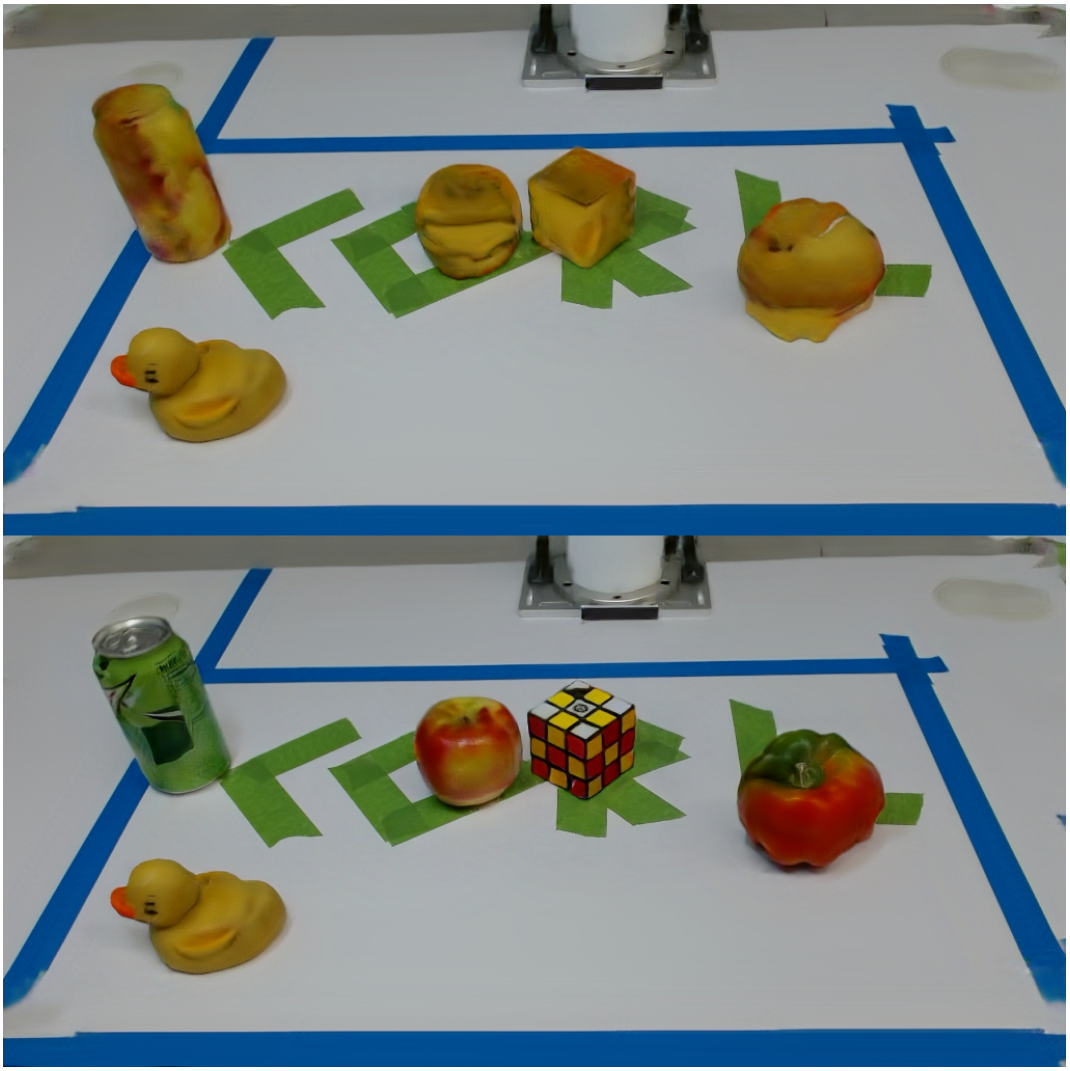

Figure 23: On the bottom, we have a normal translation: it includes five objects (a soda can, an apple, a Rubik's cube, a pepper and a duck). On the top we've set the label of all objects (besides the table) to the duck label. It looks pretty terrible, because the UV mapping was never meant for that geometry. Please note that no rubber duckies were harmed in the making of this experiment.

But what happens if we swap them *during* training? We can get more interesting results if we swap the labels and then allow it to train for a while afterwards. Typically the objects will get stuck in

a local minimum, and try to make the best of their new identities. This works because TRITON is fairly robust to geometric imperfections - it even manages to make the duck look like a Rubik's cube, and the apple to look like a very fat duck.

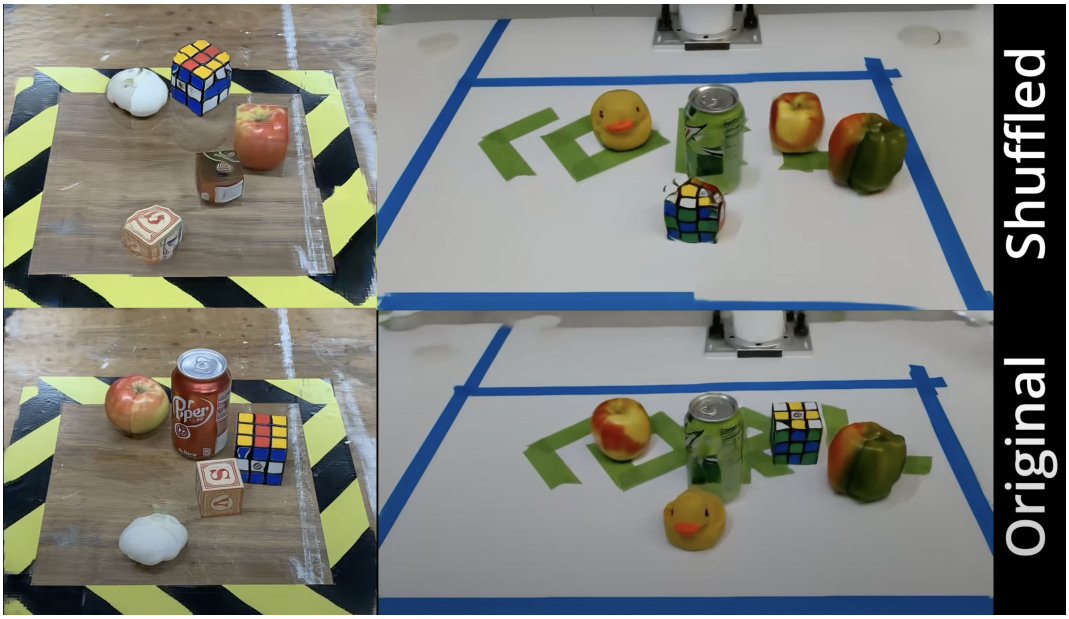

Figure 24: On the bottom two images, we have the original TRITON output for two different datasets. On the top, we've shuffled some of the labels and allowed it to train for an additional 50,000 iterations - letting it converge on a local minimum where the objects now have the wrong labels but still attempt to look good. On the left dataset, we shuffled the garlic, apple, soda, apple and alphabet cube labels. On the right dataset, we shuffled the duck, apple and rubiks cube labels. Click the image to view an animation, or go to `https://youtu.be/ivfBslj4Gsg`. Even the duck has managed to turn itself into an oddly convinving rubik's cube.

E.2 Shadow Analysis

TRITON can mimic shadows and shading effects, but they are not physically accurate. This is because, like the other baselines, TRITON is an image translation algorithm and does not have access to the latent geometry used to create a UVL scene image. Without the underlying geometry for a scene, it can't calculate the correct position for shadows. The fake shadows that TRITON generates are important for the output's realism though.

To isolate the shadows and shading artifacts, we randomly choose 100 samples from the UVL dataset and translate them using TRITON. The objects in these samples have different orientations and locations so that the shadow is seen at different parts of the object in different samples. Then we unproject the pixels of all the translated samples back onto a texture as seen in figure 5, and the average of all the unprojected textures are denoted as the mean unprojected texture $\mu(\omega)$. By averaging over 100 samples, we can get rid of the random noisy shadow in a single image and recover the original texture agreed by majority of the samples without shading artifacts. Then the mean unprojected texture $\mu(\omega)$ is projected back and this image without shadow is called "Reprojection" (See the bottom right figure in Figure 24). Finally the shadow of an individual sample can be extracted by subtracting "Reprojection" from the sample (Top right in Figure 24).

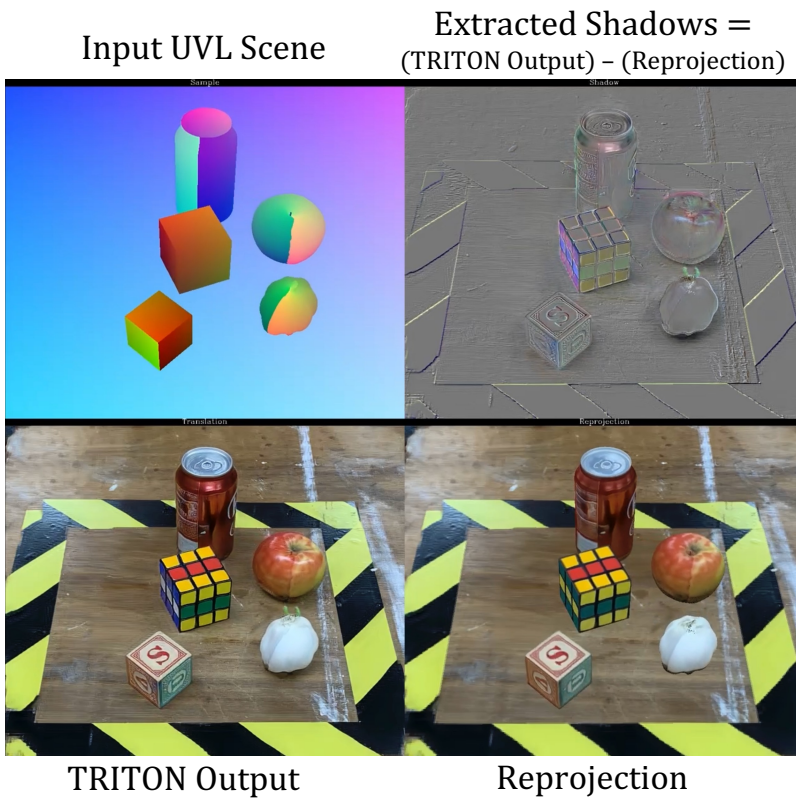

Figure 25: In this figure, we show the extracted shadows for a given scene. By looking at the 'Shadows' corner of the image, you can see that there are dark regions under the can and the apple - indicating that those shadow values are below the average surface value for that dataset's TRITON output. Note that it also shows other lighting effects beyond shadows, such as the shiny parts of the soda can. Click the image to view an animation of this analysis for three different datasets, or go to https://youtu.be/ZscnUbKOxdg.