The Supplementary Material for Deep 3D-to-2D Watermarking: Embedding Messages in 3D Meshes and Extracting Them from 2D Renderings

The detailed architectures of our encoders and decoder are in Sec. 1. The parameters and implementation details are in Sec. 2. Thorough evaluations of each distortion are in Sec. 3. The bit accuracy and message bits are discussed in Sec. 4. Lastly, more results generated by our watermark encoders and the differences are in Sec. 5.

1. Architectures

As described in the paper, we used variations of PointNet [4] as backbone architectures for the vertex encoder network. For the texture encoder, we use CNN-based architectures such as HiDDeN [6]'s encoder, or a fully convolutional U-Net [5].

1.1. 3D Vertex Encoder

PointNet Fig. 1 shows the PointNet architecture, *i.e.*, our encoder backbone network. Most parts are similar to [4], yet, there are a few differences: 1) we changed the shape of input points which originally only accept 3D positional element, $\{x, y, z\}$, and now could accept any numbers of vertex elements C_v; 2) the pooling layer (marked as red in Fig. 1) in the architecture can be either max pooling (PointNet) or global pooling (PointNet v2 in the paper) for accepting mesh inputs with different size.

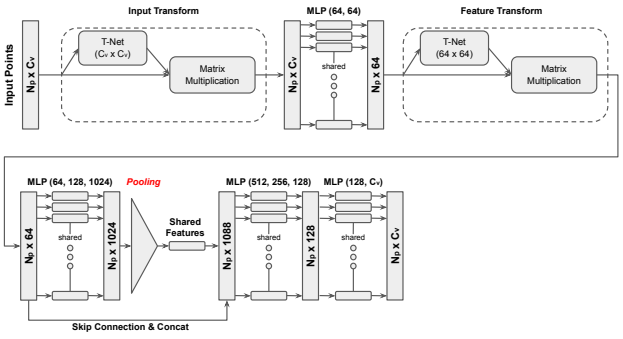


Figure 1. PointNet architecture

1.2. 2D Texture Encoder

CNN Encoder Our CNN encoder is based on HiDDeN [6]'s encoder architecture. The input texture is first passed by four CBR blocks containing 3×3 convolution, batch normalization, and ReLU activation. Each block has 64 units per layer. Then, input messages are repeated to have a same dimensions of $height \times width$ of the input textures, and concatenated in the channel dimension. The message appended feature maps are further connected to two CBR blocks. The last CBR block has the unit of 3 so that the output has the same shape of input texture (which is the message embedded texture). For the convolutional layers in the encoder architecture, we use 'VALID' padding.

U-Net U-Net [5] architecture systematically combines the autoencoder architecture and skip-connection scheme as shown in [2]. For our texture encoder, we modified U-Net architecture to make it fully convolutional through: 1) removing the fully-connected layer which generates the latent space vector in between the autoencoder's encoder and decoder; 2) using $\{64, 128, 256, 512\}$ units per U-Net block with max pooling by 2 in each block.

1.3. Image Decoder

CNN Decoder Our CNN decoder is based on HiDDeN [6]'s decoder architecture. The base CBR block is same as the CNN encoder as described in Sec. 1.2. We use seven CBR blocks. The last two CBR blocks are applying stride with 2. Then, global pooling is applied to the last CBR block to accept any image dimensions. Lastly, a fully-connected layer is used to generate the output message logits. To have the same dimension of message bits, the output unit of fully-connected layer is N_b.

2. Differentiable Rendering

As mentioned in the paper, we leverage the state-of-the-art work in differentiable rendering and follow the work by Genova et al. [1]. We explain the steps in more details here:

- The differentiable renderer first takes $N_v \times 3$ world-space vertices and a sampled camera position as input,

and computes $N_v \times 4$ projected vertices using camera projection $\mathcal{P}$, following OpenGL convention [3].

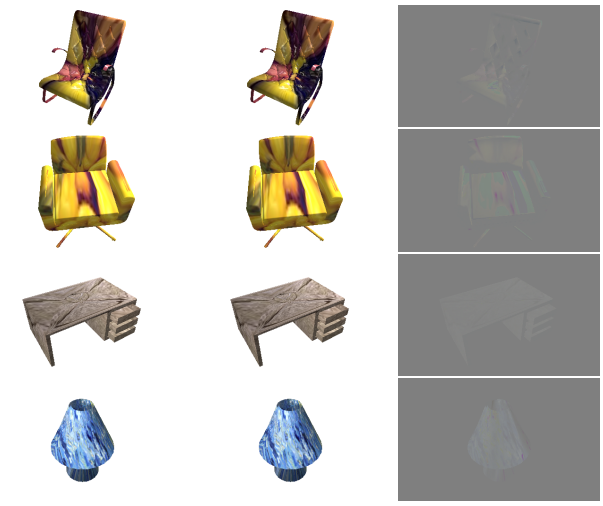
- Then it rasterizes the triangles by identifying the frontmost triangle ID $\mathcal{T}$ at each pixel and computing the barycentric coordinates of the pixel inside triangles $\mathcal{B}$.
- After rasterization, we interpolate $N_v \times 5$ vertex attributes, i.e. normals and uvs, at the pixels using the barycentric coordinates $\mathcal{B}$ and triangle IDs $\mathcal{T}$. We take the interpolated per-vertex attributes and the lighting parameters $\mathcal{L}$ to compute the shaded colors $\mathcal{C}$. Here we use a Phong model with parameters $k_a = 0.8$, $k_d = 1.4$ and $k_r = 0$, and set the constant term $K_c = 1.0$, the linear term $K_l = 0.07$, and the quadratic term $K_q = 0.017$ to calculate attenuation value.
- Finally, we form a $h \times w \times 4$ buffer of per-pixel clip-space positions $\mathcal{V}$, then apply perspective division and viewport transformation to produce a $h \times w \times 2$ screen-space splat position buffer $\mathcal{S}$. In our implementation, the rendered height h and w is 400 and 600.

3. Evaluations of Distortions

We verified the robustness of our networks for distortions. Our network is trained with four distortions: additive noise, scaling, rotation, and cropping. As described in the paper, we trained our network with two different types of textures: real and noise textures. Fig. 5 shows the graphs between bit accuracy and distortion strength. The results are trained with message length 8, real textures (left) and noise textures (right). Note that noise distortion plots in Fig. 5 are based on $\mu = \{\pm 0.1, \pm 0.15, \pm 0.2, \pm 0.25, \pm 0.3, \pm 0.4\}$, and $\sigma = \{0.03, 0.05, 0.06, 0.0833, 0.1, 0.133\}$. As we can see in the Fig. 5, the bit accuracy for network trained with real textures is lower than noise textures, however, the tendencies between bit accuracy and distortion strength are similar. Overall, our network is robust to noise, rotation, scaling, and cropping distortions.

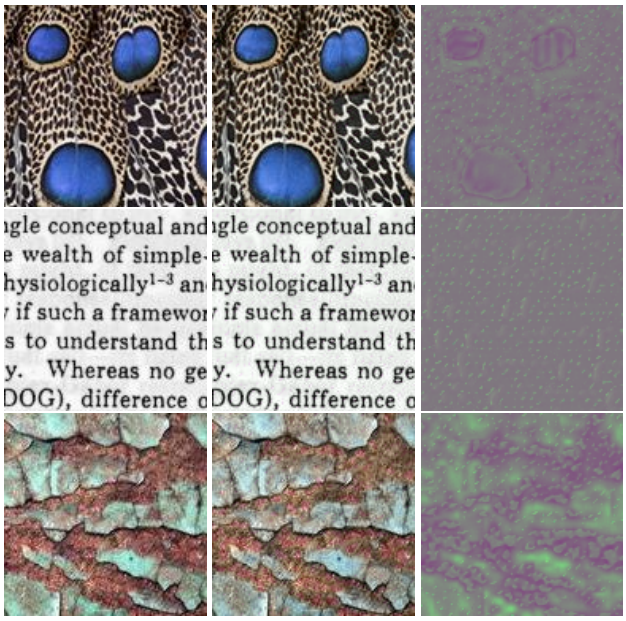
4. Bit Capacity

Fig. 4 shows the bit accuracy and the length of message bits under different encoder settings. For the experiments, we trained 10 times of the vertex-only, the texture-only, and the vertex + texture encoders. Then calculated the best, mean, and standard deviation of the bit accuracy. As we can see in Fig. 4a, vertex-only encoders can encode a few bits and cannot handle more than 8-bits (below 60% accuracy). On the other hand, texture-only encoders (Fig. 4b) can encode more bits, yet showed unstable training based on the standard deviation. The vertex + texture encoders (Fig. 4c) showed the nearly same bit accuracy, with narrower standard deviation area.



(a) Not Watermarked (b) Watermarked (c) Difference (4x)

Figure 2. The rendered images from input meshes (a), watermarked meshes (b) , and the difference images (c), with real textures.

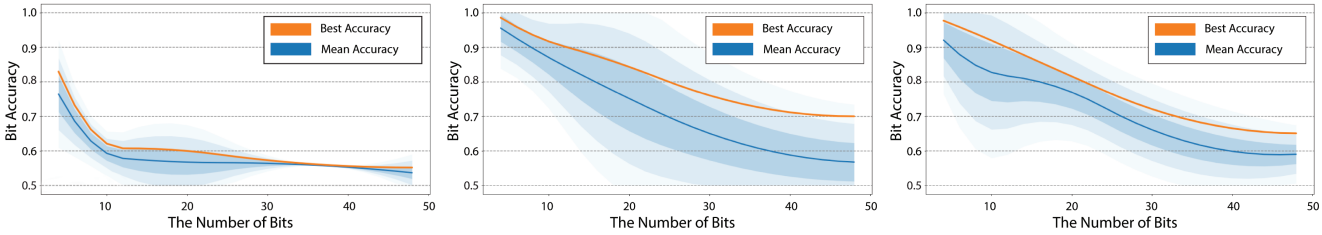


(a) Not Watermarked (b) Watermarked (c) Difference (4x)

Figure 3. Comparisons between not watermarked and watermarked real textures.

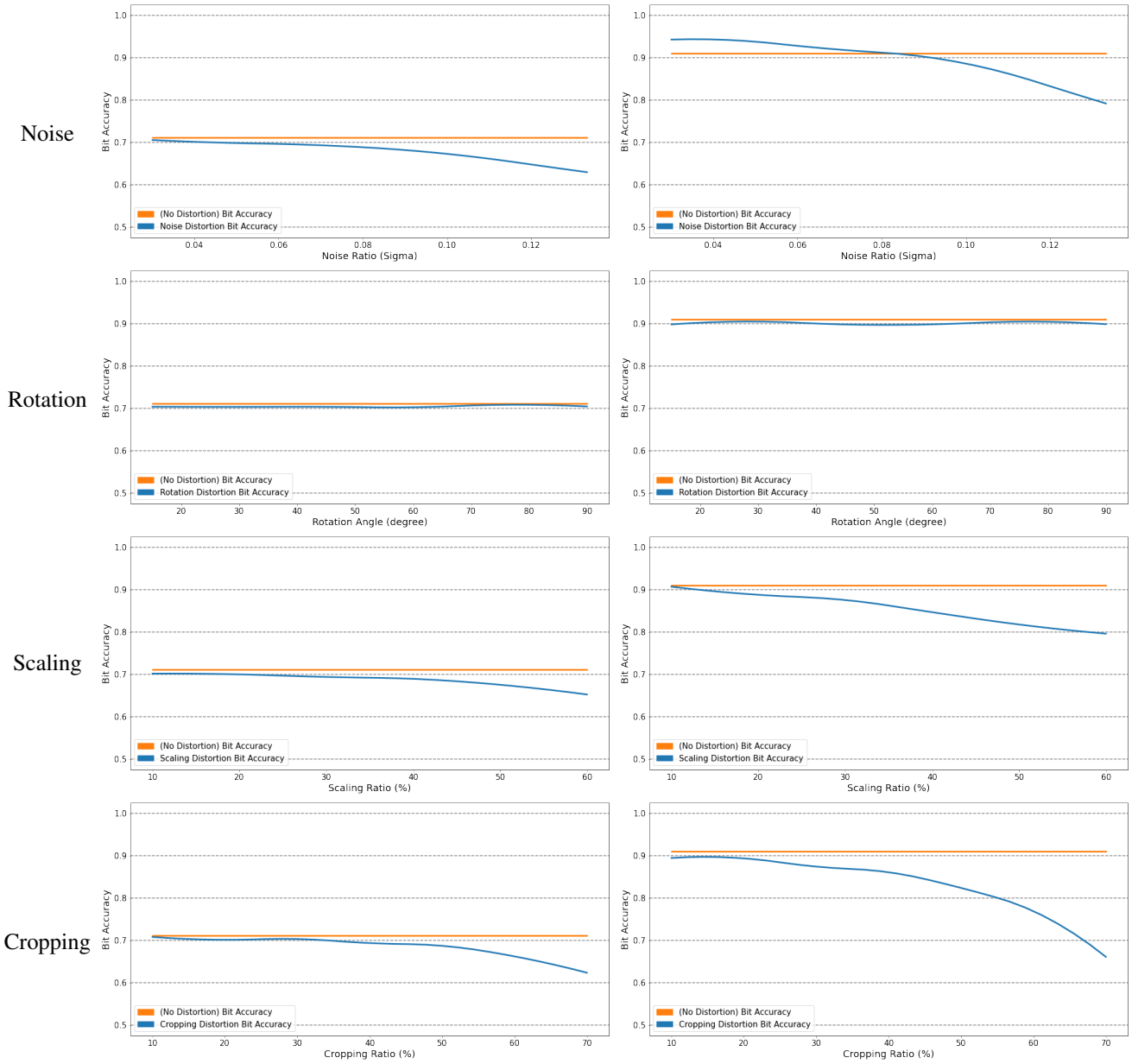
5. More Results

We provide more rendered results that rendered from original meshes, watermarked meshes, and the differences between the two images (Fig. 2). Also original textures, watermarked textures, and the difference are shown in Fig. 3.



(a) Mesh vertex encoder (b) Mesh texture encoder (c) Mesh vertex + texture encoders

Figure 4. The relationship between bit accuracy and the length of message.



(a) Real Texture (b) Noise Texture

Figure 5. Bit accuracy against distortion strength.

References

[1] Kyle Genova, Forrester Cole, Aaron Maschinot, Aaron Sarna, Daniel Vlasic, and William T Freeman. Unsupervised training for 3d morphable model regression. In *CVPR*, 2018. 1

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 1

[3] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, Version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016. 2

[4] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, 2017. 1

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015. 1

[6] Jiren Zhu, Russell Kaplan, Justin Johnson, and Li Fei-Fei. HiDDeN: Hiding Data with Deep Networks. In *ECCV*, 2018. 1