Supplementary Material

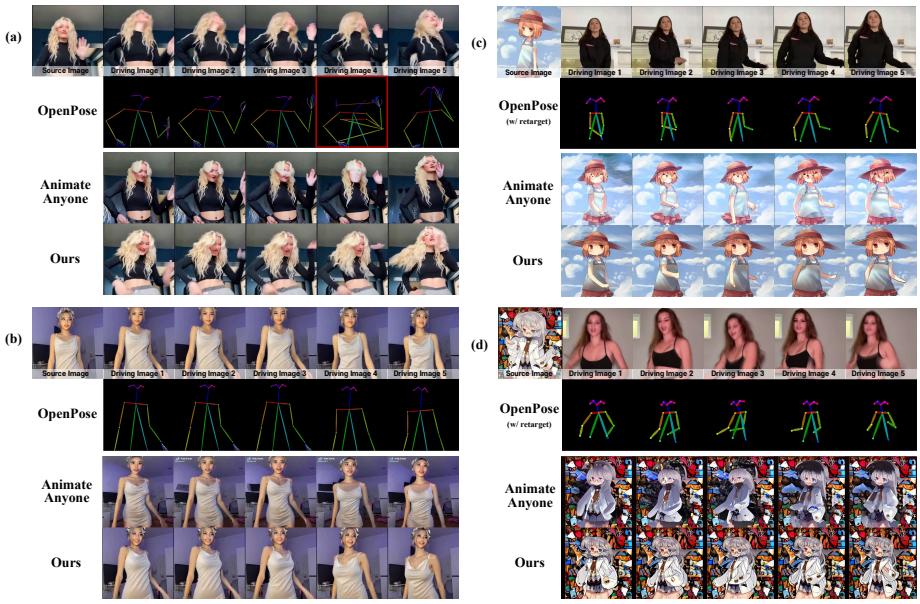

Fig. 10: Qualitative comparison of AnimateAnyone [15] and TCAN on TikTok dataset (a and b) and Bizarre dataset (c and d). The erroneous pose is highlighted in red.

A Comparison with AnimateAnyone

Method	L1 ↓	SSIM ↑	LPIPS ↓	FID ↓	FID-VID ↓	FVD ↓
AnimateAnyone [15]*	1.07E-04	0.692	0.366	28.75	42.40	341.07
TCAN (*w/ mm1*)	**8.85E-05**	**0.734**	**0.299**	**29.07**	**29.74**	**189.77**

Table 3: Quantitative comparison with AnimateAnyone [15] on TikTok dataset. To evaluate AnimateAnyone, we use re-implemented code*. All methods are trained on TikTok training dataset. We highlight the best performance in bold.

In this section, we compare our TCAN to the recent human image animation method, AnimateAnyone [15], which also uses OpenPose as the input pose condition. Since the AnimateAnyone does not have an official code, we trained re-implemented version of AnimateAnyone [5] on TikTok training dataset.

Table 3 shows the quantitative results of both methods. Our model outperforms the re-implemented AnimateAnyone in all performance metrics. Fig. 10 shows the qualitative comparison of TikTok and bizarre dataset. As for AnimateAnyone, we observe the distorted face region under inaccurate pose condition (Fig. 10 (a)), and the black object flickering in the background (Fig. 10

[5] AnimateAnyone(Re-implemented): https://github.com/MooreThreads/Moore-AnimateAnyone

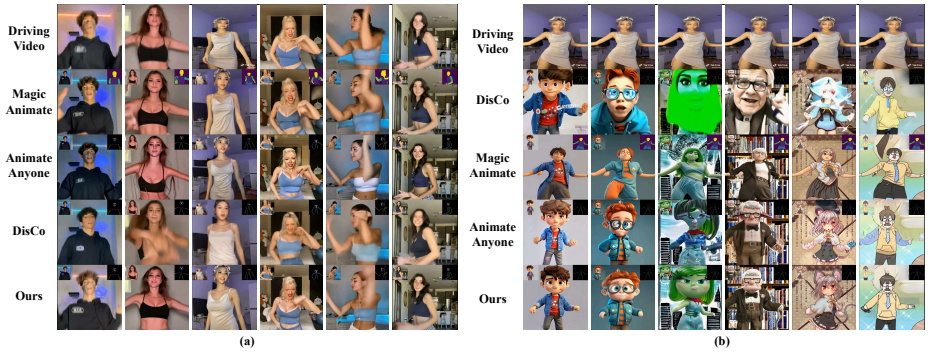

Fig. 11: Additional TCAN results on (a) TikTok dataset and (b) various animation characters. We select frames from each video at the same time and compare them. The full video can be viewed on our project page.

(b)). In contrast, the results of TCAN shows well-preserved faces and clean, stable backgrounds. Fig. 10 (c) and Fig. 10 (d) illustrates the results on Bizarre dataset. Fig. 10 shows AnimateAnyone fails to preserve identities such as faces or clothing, while our TCAN generalizes well to unseen domain.

B Additional Videos for Visual Comparison

Qualitative Comparison on TikTok Dataset To demonstrate the effectiveness of TCAN and to compare our approach with human image animation baselines, we upload videos on our project page [6] , which showcases generated videos where a source image is the first frame of each driving video. The selected frame from the video is attached in Fig. 11 (a). It is evident from the video that TCAN yields the most realistic results compared to the others. Among all baselines, MagicAnimate stands out as the closest competitor. MagicAnimate's advantage comes from its use of DensePose for guidance, providing detailed shape information from the driving video that aligns with the shape of the source image's foreground object. In contrast, our TCAN achieves remarkable performance by utilizing sparse pose information only.

Qualitative Comparison on Online-Collected Images In this section, we show additional results on the online-collected animation characters. We have collected characters with diverse body proportions and styles, and use them as source images. As depicted in Fig. 12, our methodology follows the motion of the foreground object in driving videos while effectively maintaining the identity and background of the source image. Furthermore, we upload generated videos using animation characters on our project page for qualitative comparisons with baseline methods (*i.e.,* DisCo, MagicAnimate, and re-implemented

[6] TCAN project page: https://eccv2024tcan.github.io

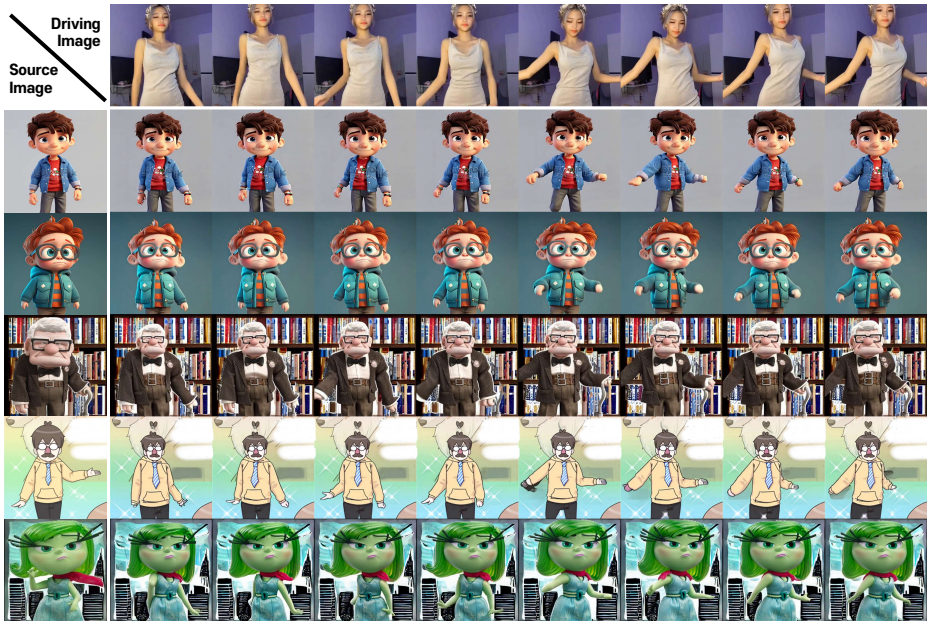

Fig. 12: Additional TCAN results on various animation characters.

AnimateAnyone). The sampled frames from the video are shown in Fig. 11 (b). As shown in the the figure, DisCo exhibits significant distortion of the source image's identity, and MagicAnimate changes the character's body proportions, leading to an identity shift. In contrast, our TCAN effectively follows the motion of driving videos with diverse body proportions and styles, even when trained solely on the TikTok dataset.

C Detailed Description of Pose Retargeting

As discussed in section 4.2, we apply the pose re-targeting algorithm to align the ratio of foreground objects of driving images to that of the source image. Specifically, we fix the 'neck' keypoint of the driving image and adjusted the position of the 'nose' so that the distance between the 'nose' and 'neck' keypoints in the source image matched the corresponding distance in the driving image. The keypoints connected to the 'nose' were moved by the same amount. Similarly, we traverse through the remaining keypoints, adjusting their positions to match the distances between the keypoints in the source image. We provide a detailed description of Pose-Retargeting in Algorithm 1 as pseudo-code.

Algorithm 1: Pseudo-Code for Pose-Retargeting

```
def pose_retarget(cur_kp, src_kp, init_kp, kp_mapper):
    # cur_kp : 18 keypoint coordinates from the current frame.
    # src_kp : 18 keypoint coordinates from the source image.
    # init_kp : 18 keypoint coordinates from the initial frame.
    # kp_mapper : A mapper storing the relationship between two connected keypoints.

    # Calculate the ratio of the lengths between the keypoints of the initial and the
     current frame.
    curinit_ratio = [1] * 18
    for kp1 in range(2, 18):
        kp2 = kp_mapper[kp1]
        cur_length = get_length(cur_kp[kp1], cur_kp[kp2])
        init_length = get_length(init_kp[kp1], init_kp[kp2])
        curinit_ratio[kp1] = cur_length / init_length

    # First adjusting the length between the neck(1)-nose(0).
    neck_kp = 1
    nose_kp = kp_mapper[neck_kp]
    src_length = get_length(src_kp[neck_kp], src_kp[nose_kp])
    cur_length = get_length(cur_kp[neck_kp], cur_kp[nose_kp])
    ratio = cur_length / src_length
    diff_vector = get_diff_with_ratio(cur_kp[neck_kp], cur_kp[nose_kp], ratio)
    cur_kp[nose_kp] += diff_vector

    # Move all subkeypoints of the nose keypoint (i.e., eyes and ears)
    cur_kp = move_subkeypoints(cur_kp, nose_kp, diff_vector)

    # Traverse and move the remaining keypoints and subkeypoints based on the length
     of the nose-neck.
    for kp1 in range(2, 18):
        kp2 = kp_mapper[kp1]
        src_length = get_length(src_kp[kp1], src_kp[kp2])
        cur_length = get_length(cur_kp[kp1], cur_kp[kp2])
        ratio = src_length / cur_length * curinit_ratio[kp2]
        diff_vector = get_diff_with_ratio(cur_kp[kp2], cur_kp[kp2], ratio)
        cur_kp[kp2] += diff_vector
        cur_kp = move_subkeypoints(cur_kp, kp2, diff_vector)
    retargeted_cur_kp = cur_kp
    return retargeted_cur_kp
```