Supplemental Material: Deep Novel View Synthesis from Unstructured Input

Anonymous ECCV submission

Paper ID 3314

1 Introduction

In this supplementary document, we present additional experimental results in Section 2. This includes quantitative and qualitative results for our own recordings, as well as additional qualitative results for the Tanks and Temples sequences and for DTU view interpolation and extrapolation. In Section 3, we provide details on our recurrent mapping and blending network.

2 Additional Results

For an additional evaluation of novel view synthesis from unstructured input, we recorded new challenging sequences. To get unstructured input for the source images, we first recorded for each scene a video sequence with a duration of one too two minutes. In addition, we recorded for each scene up to two additional videos that serve as ground-truth targets for novel view synthesis and are not included in the source input.

We summarize the quantitative results on those recordings in Table 1. Note that all results are slightly worse than the Tanks and Temples results presented in the main paper. This is an indication for the challenging setup provided in our recordings. However, we also see that the results of our method are clearly better than those of the state-of-the-art methods. In Figures 1, 2, 3, 4, 5, and 6 we show qualitative results of all methods on a subset of our own recordings.

We also present additional qualitative results for our evaluations on the Tanks and Temples sequences in Figures 7, 8, 9, 10, 11, 12, 13, and 14 and for the DTU view interpolation and extrapolation in Figures 15, 16, 17, 18, 19, 20, 21, 22, and 23.

Table 1: Quantitative results on our own recordings. Mean over all scenes.

	↓LPIPS	↑SSIM	↑PSNR
EVS [2]	14.96	40.74	12.82
LLFF [3]	17.36	30.45	10.52
NPBG [1]	9.85	70.65	18.31
Our	**7.60**	**77.42**	**19.86**

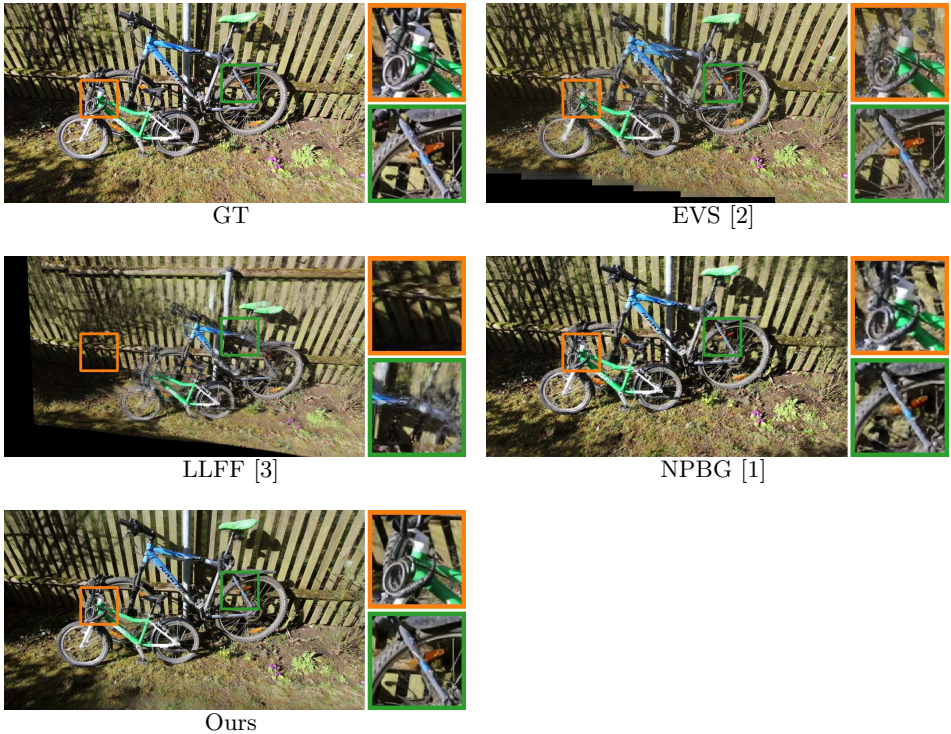


Fig. 1: Own recording *Bike*, view 6

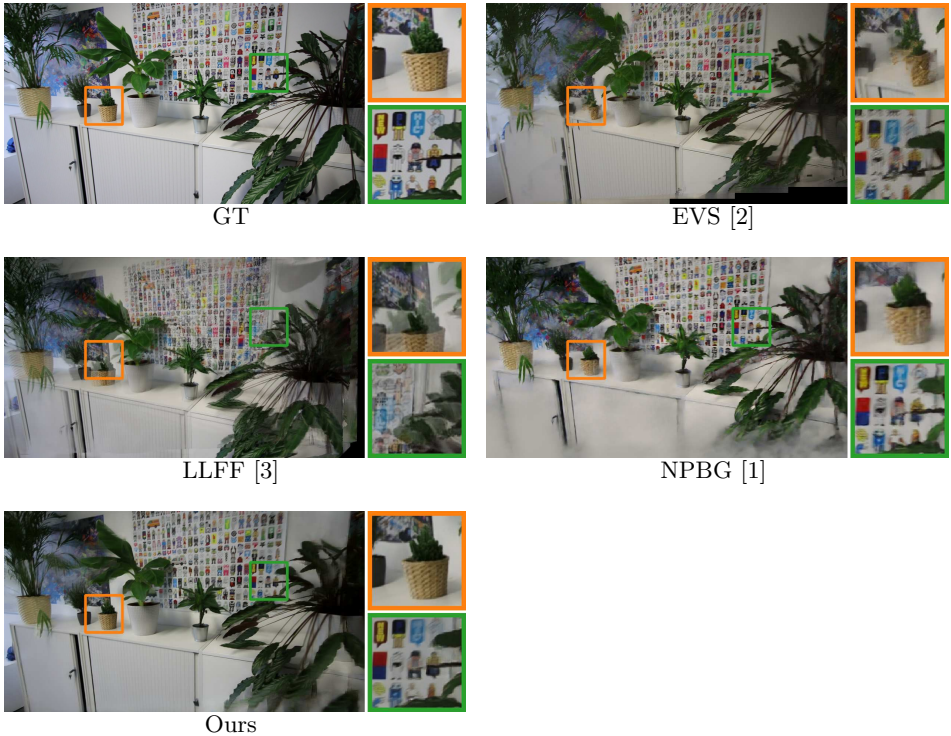


Fig. 2: Own recording *Flowers*, view 127

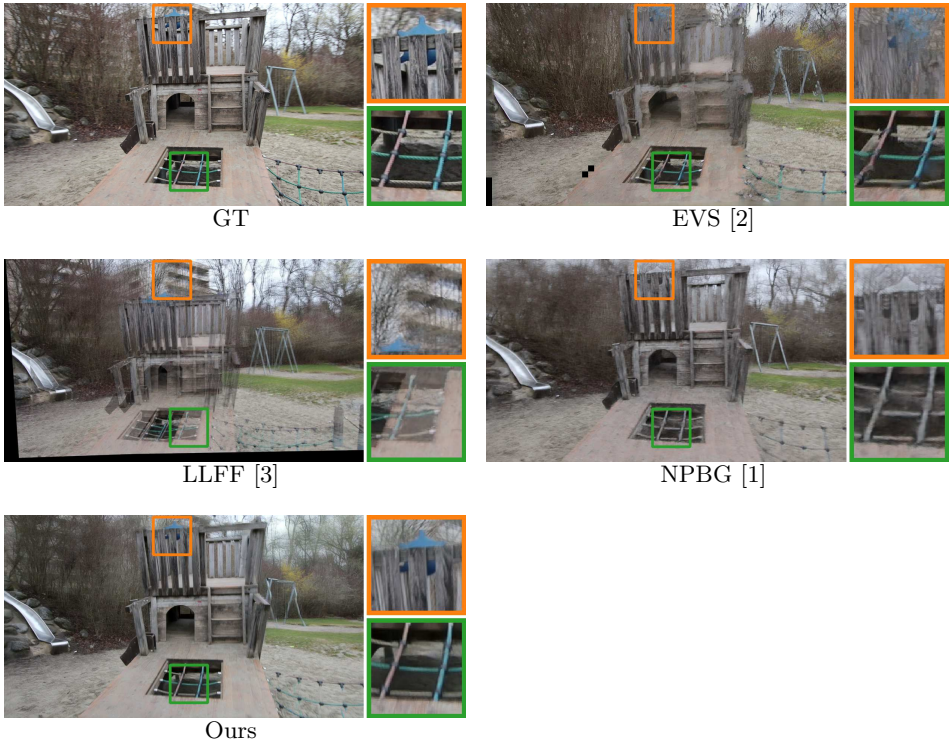


Fig. 3: Own recording *Pirate*, view 139

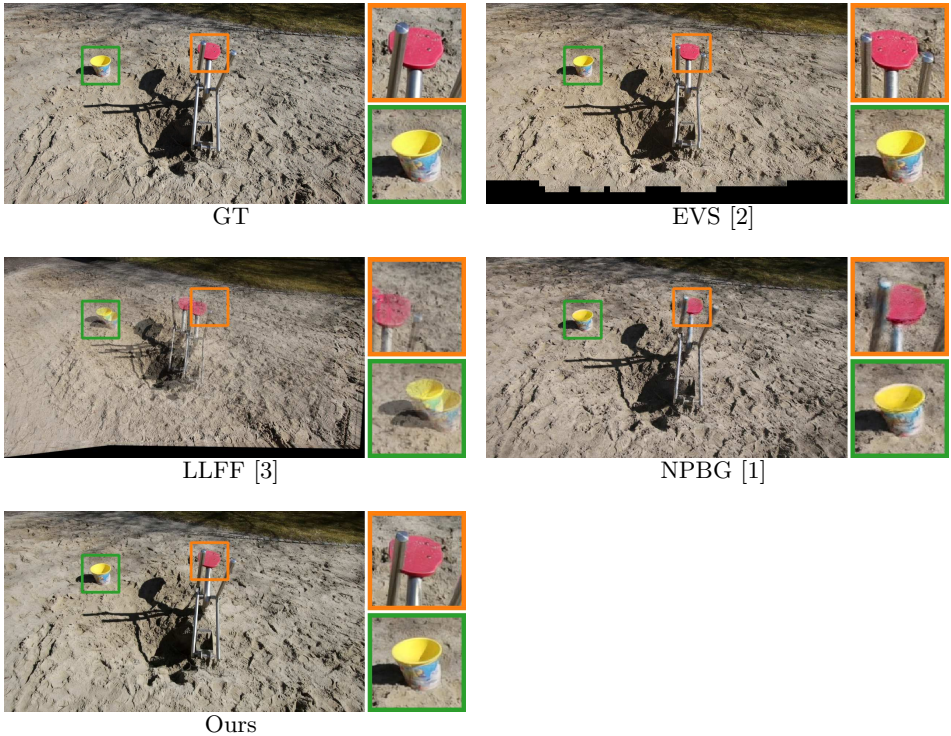


Fig. 4: Own recording *Playground*, view 43

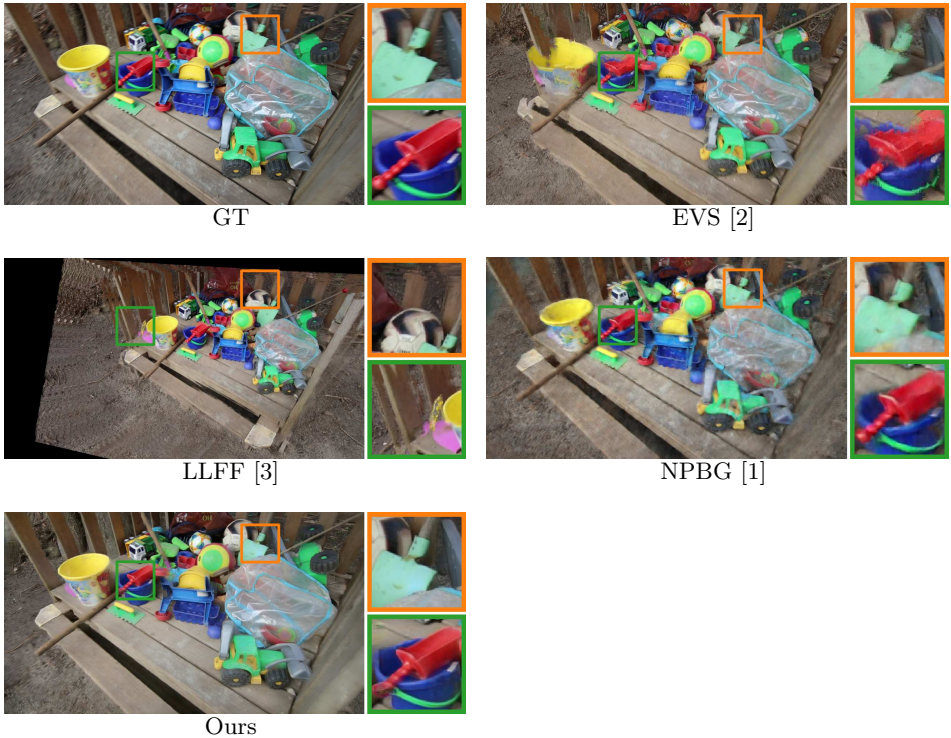


Fig. 5: Own recording *Sandbox*, view 102

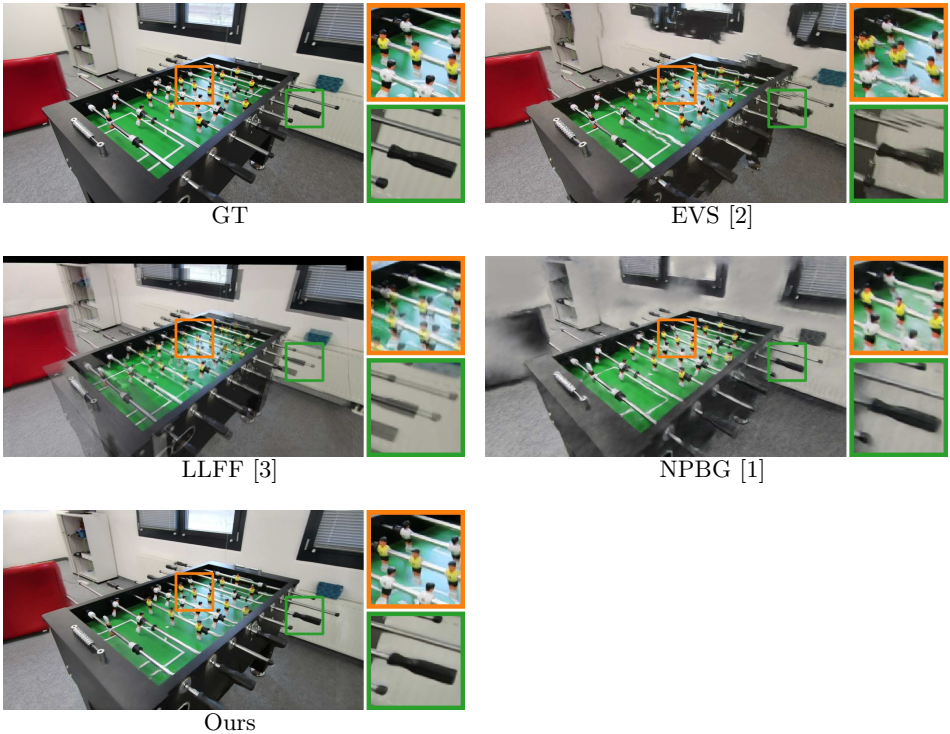


Fig. 6: Own recording *Soccertable*, view 127

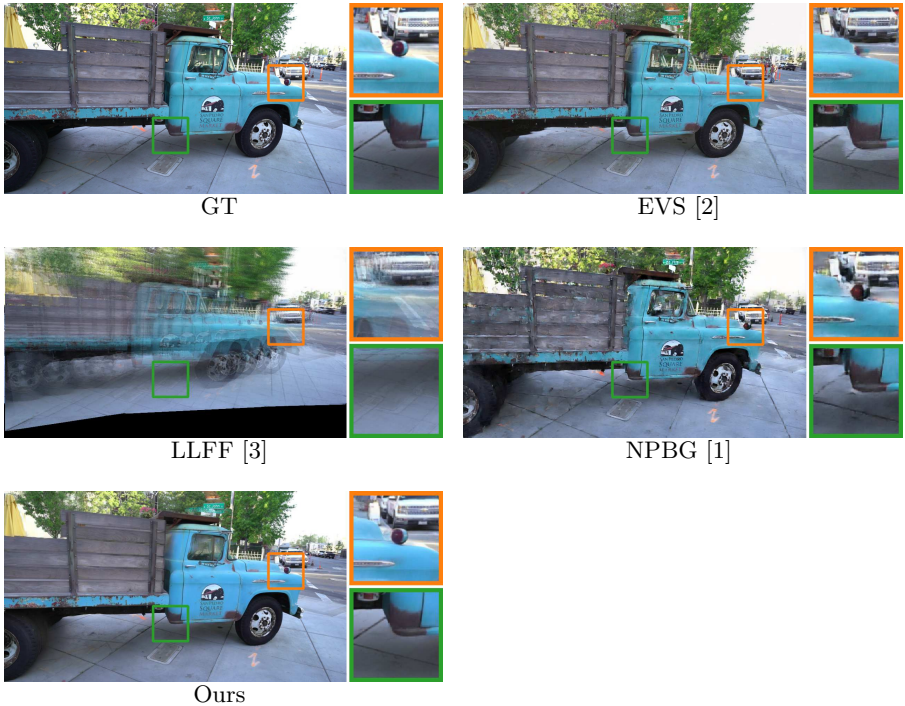


Fig. 7: Tanks and Temples *Truck*, view 0

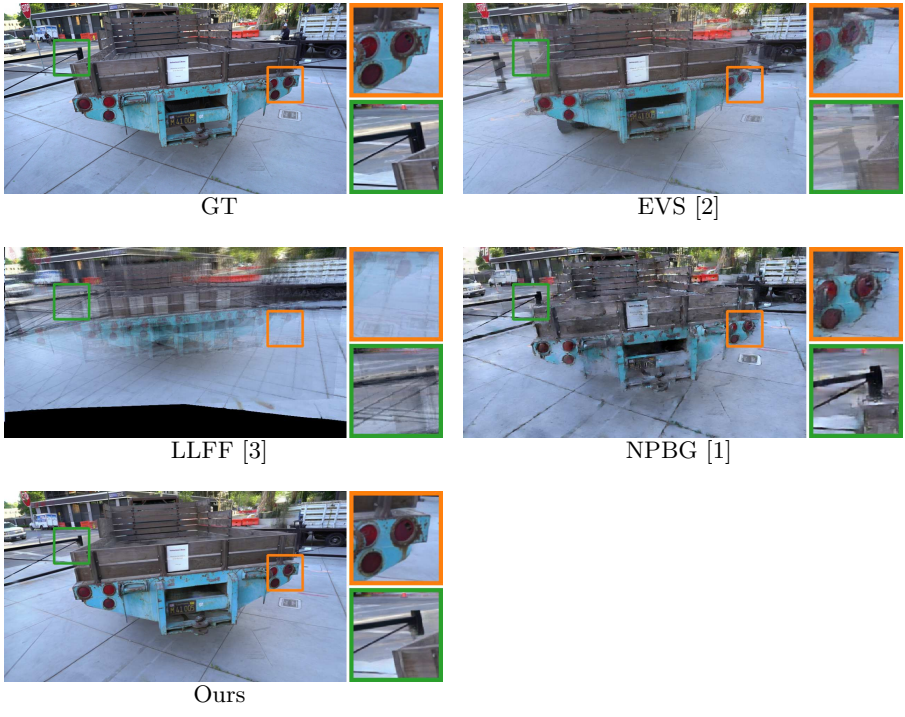


Fig. 8: Tanks and Temples *Truck*, view 16

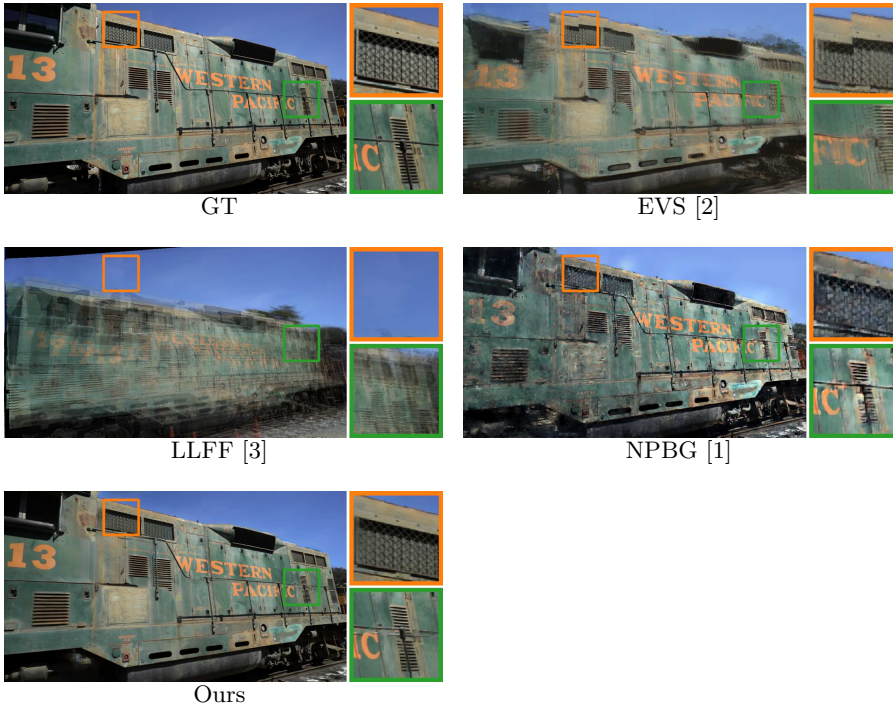


Fig. 9: Tanks and Temples *Train*, view 4

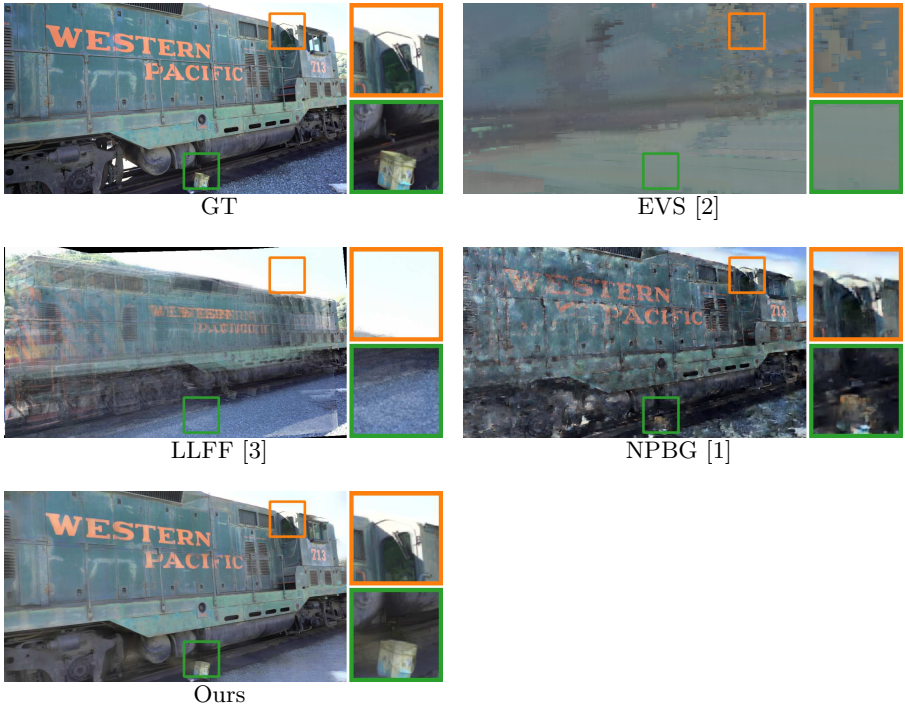


Fig. 10: Tanks and Temples *Train*, view 28

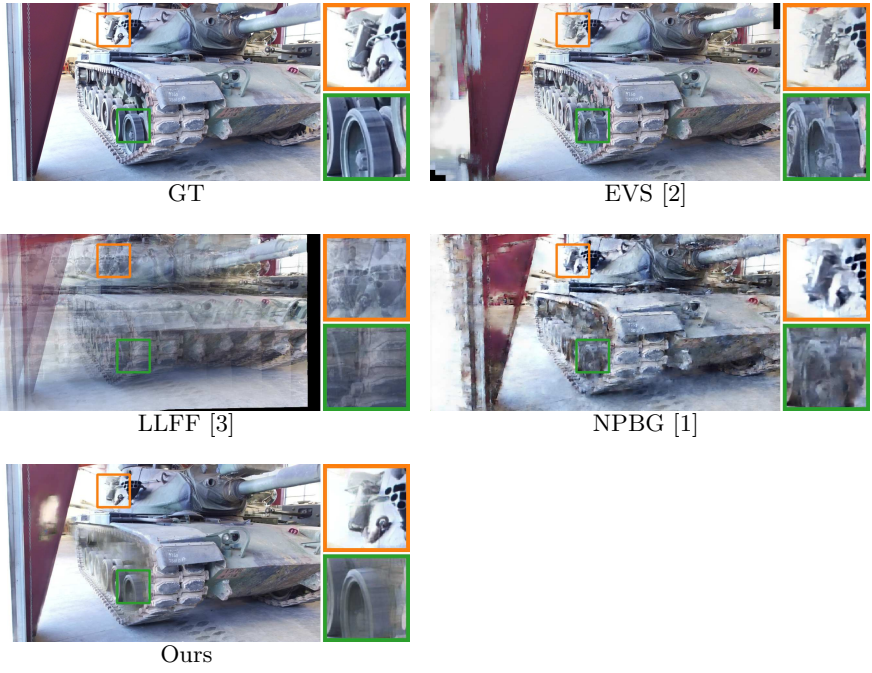


Fig. 11: Tanks and Temples *M60*, view 0

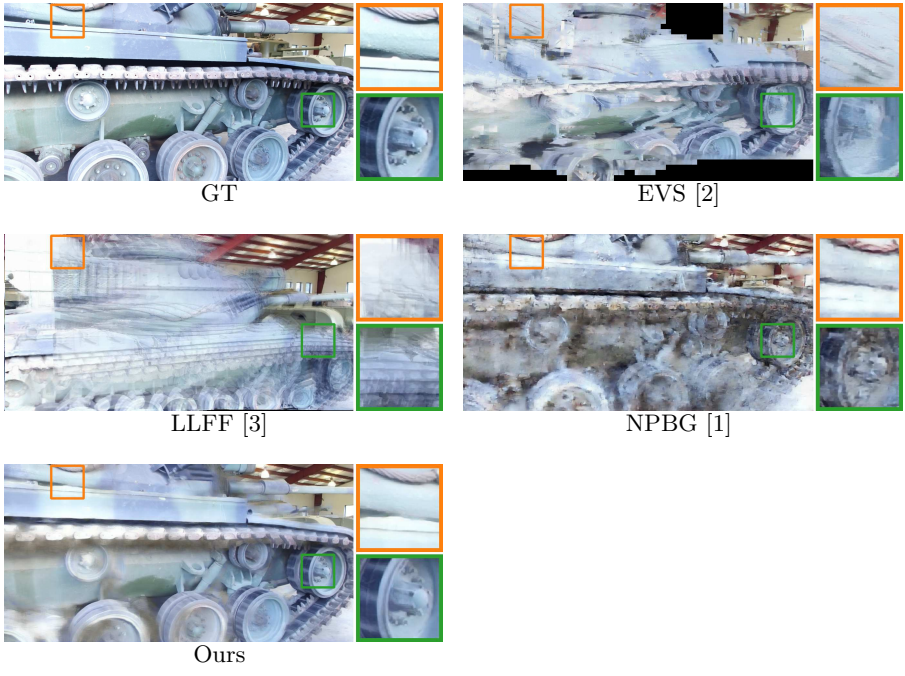
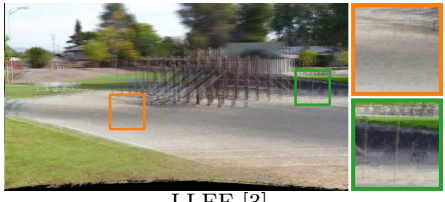


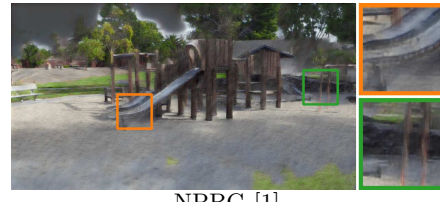
Fig. 12: Tanks and Temples *M60*, view 19

GT

EVS [2]



LLFF [3]



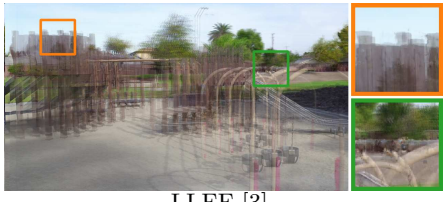
NPBG [1]

Ours

Fig. 13: Tanks and Temples *Playground*, view 10


GT


EVS [2]


LLFF [3]


NPBG [1]


Ours

Fig. 14: Tanks and Temples *Playground*, view 31

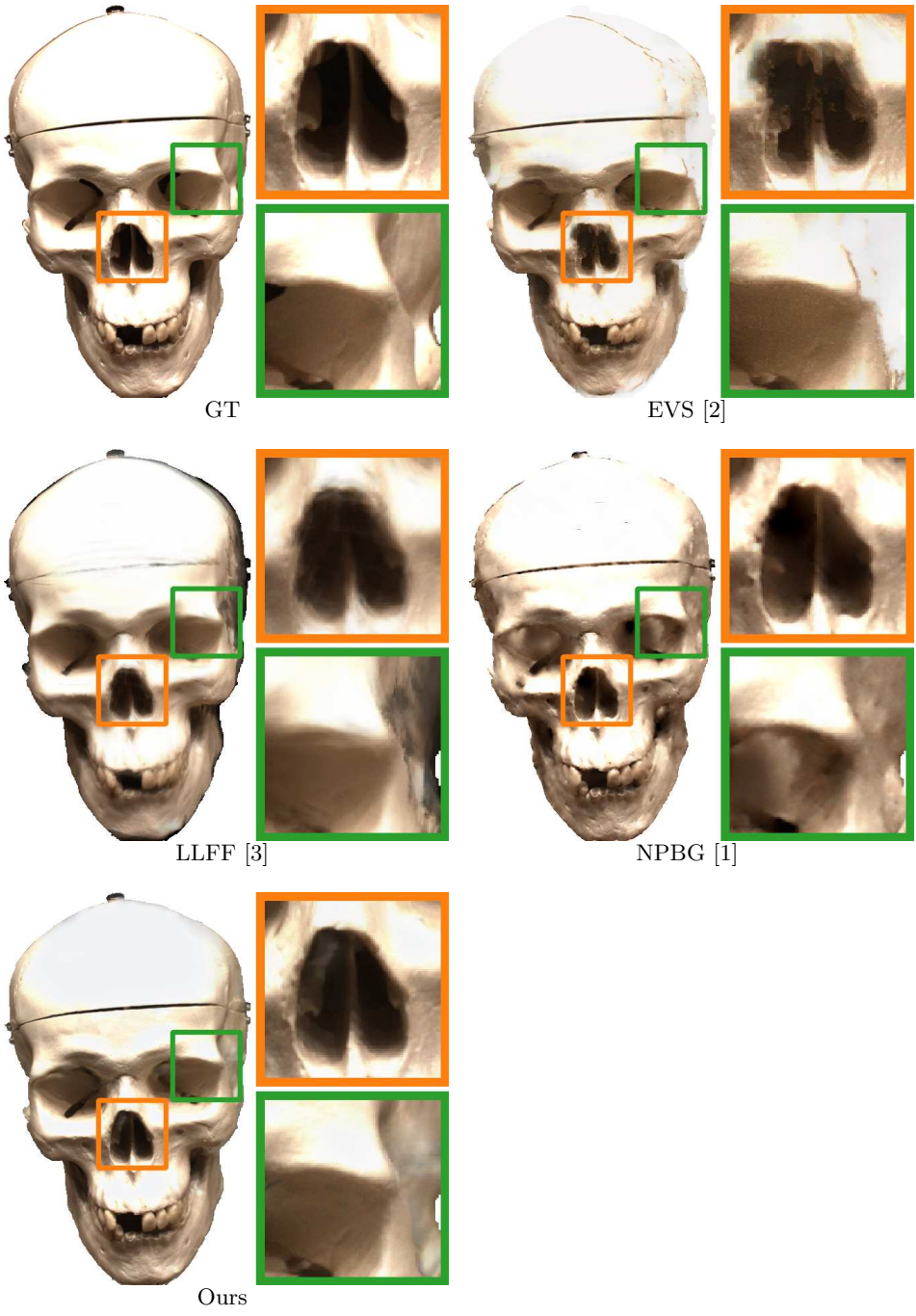


Fig. 15: DTU set 65, view 0, interpolation

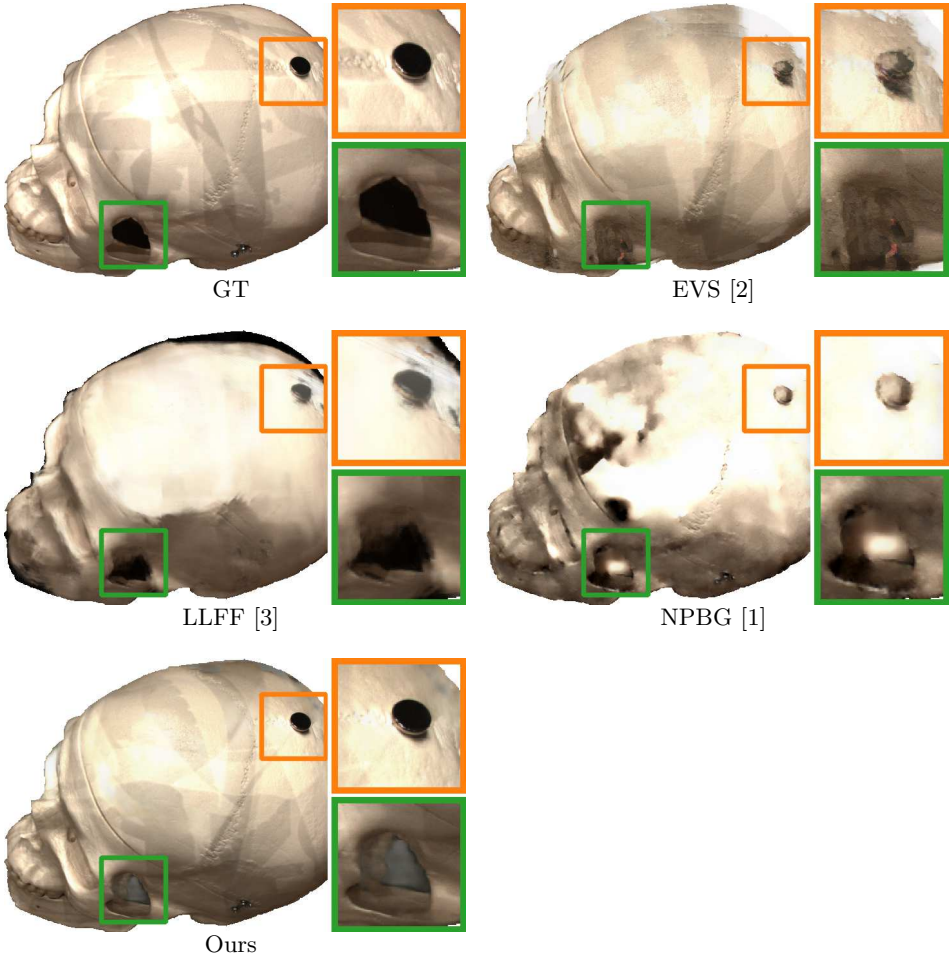


Fig. 16: DTU set 65, view 1, extrapolation

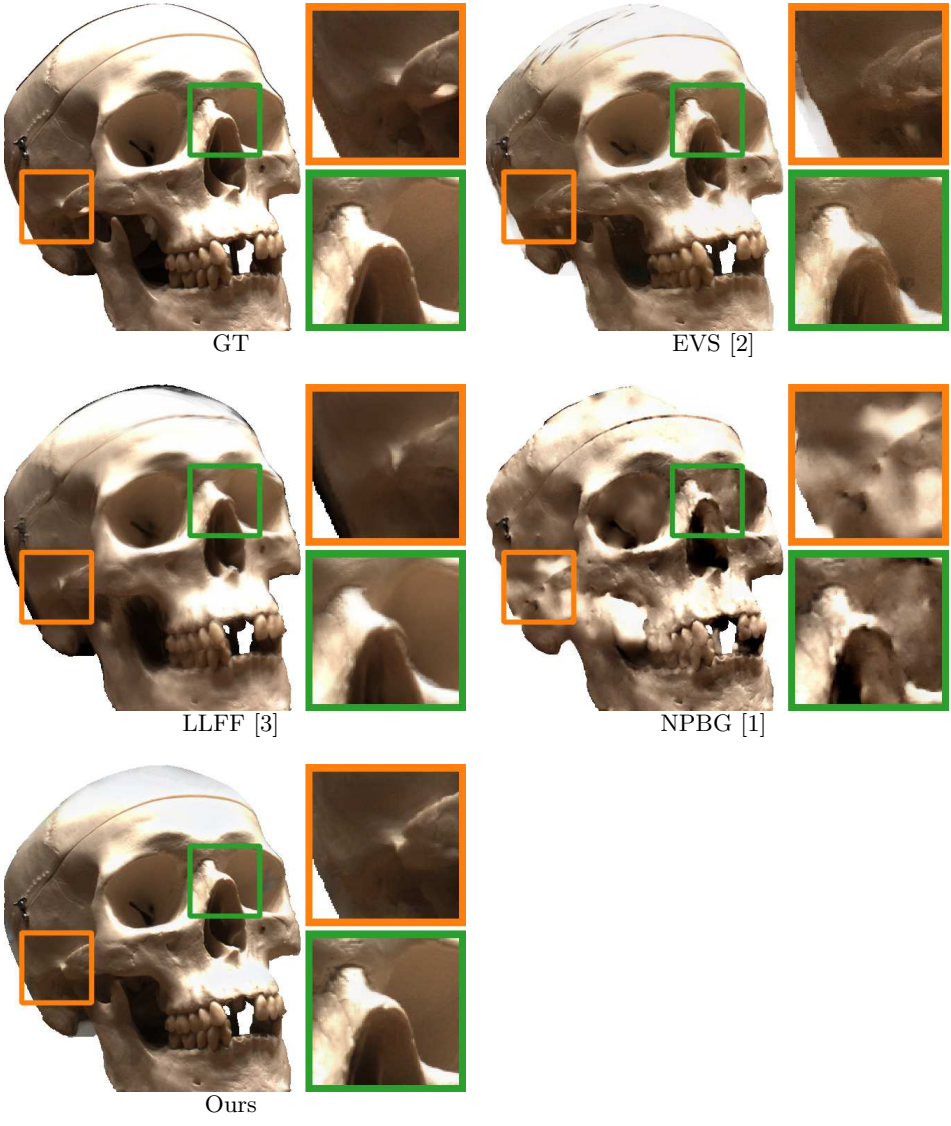


Fig. 17: DTU set 65, view 3, extrapolation

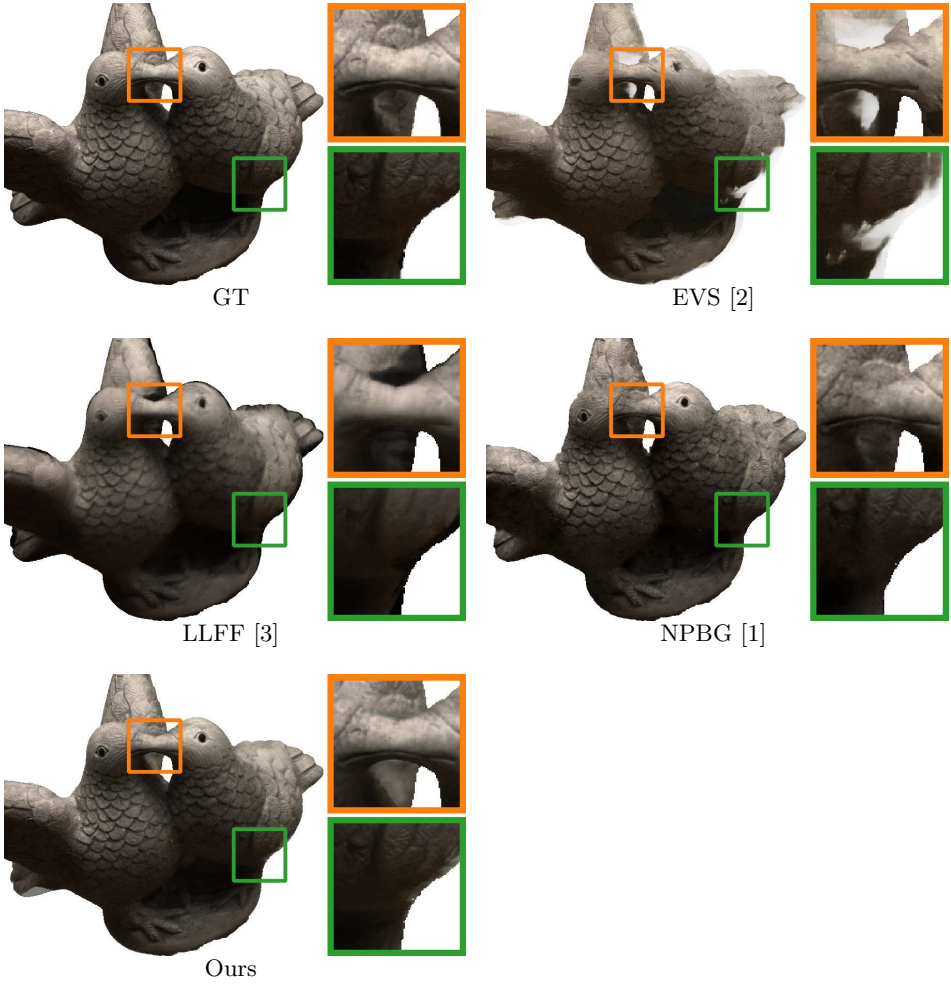


Fig. 18: DTU set 106, view 0, interpolation

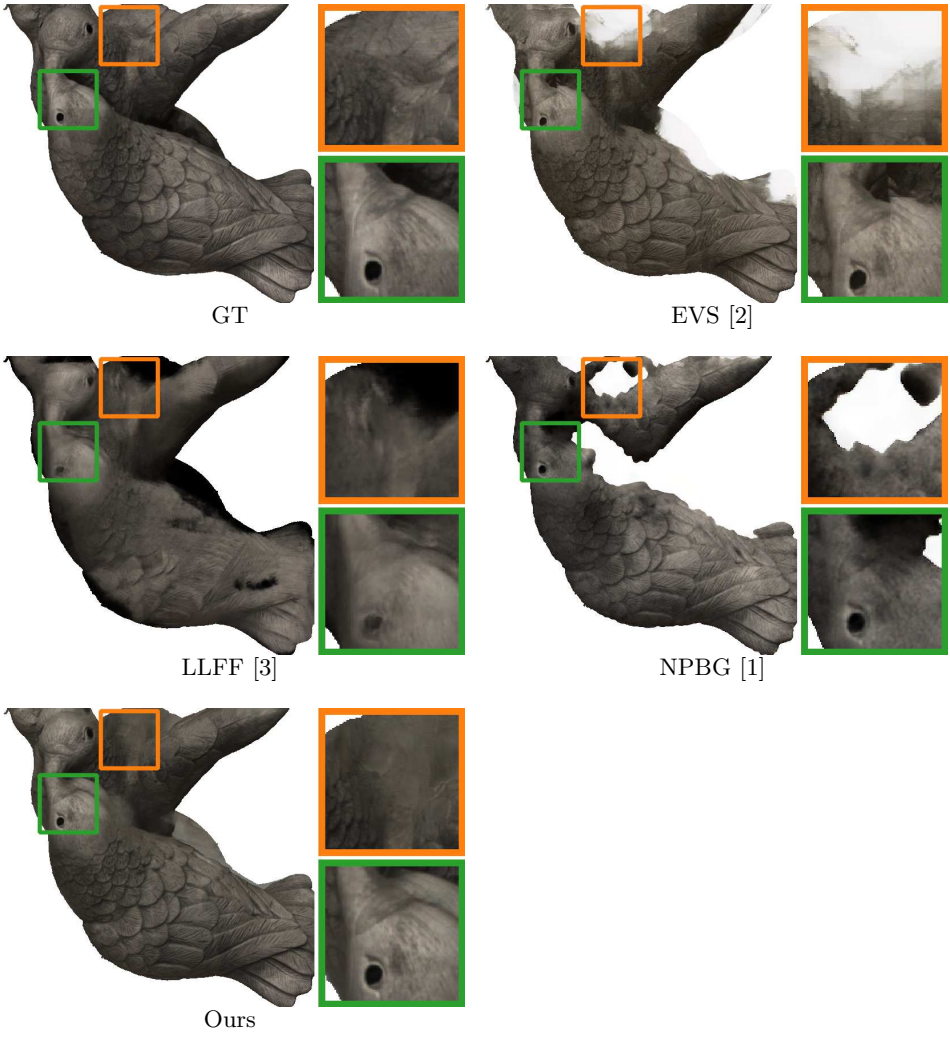


Fig. 19: DTU set 106, view 1, extrapolation

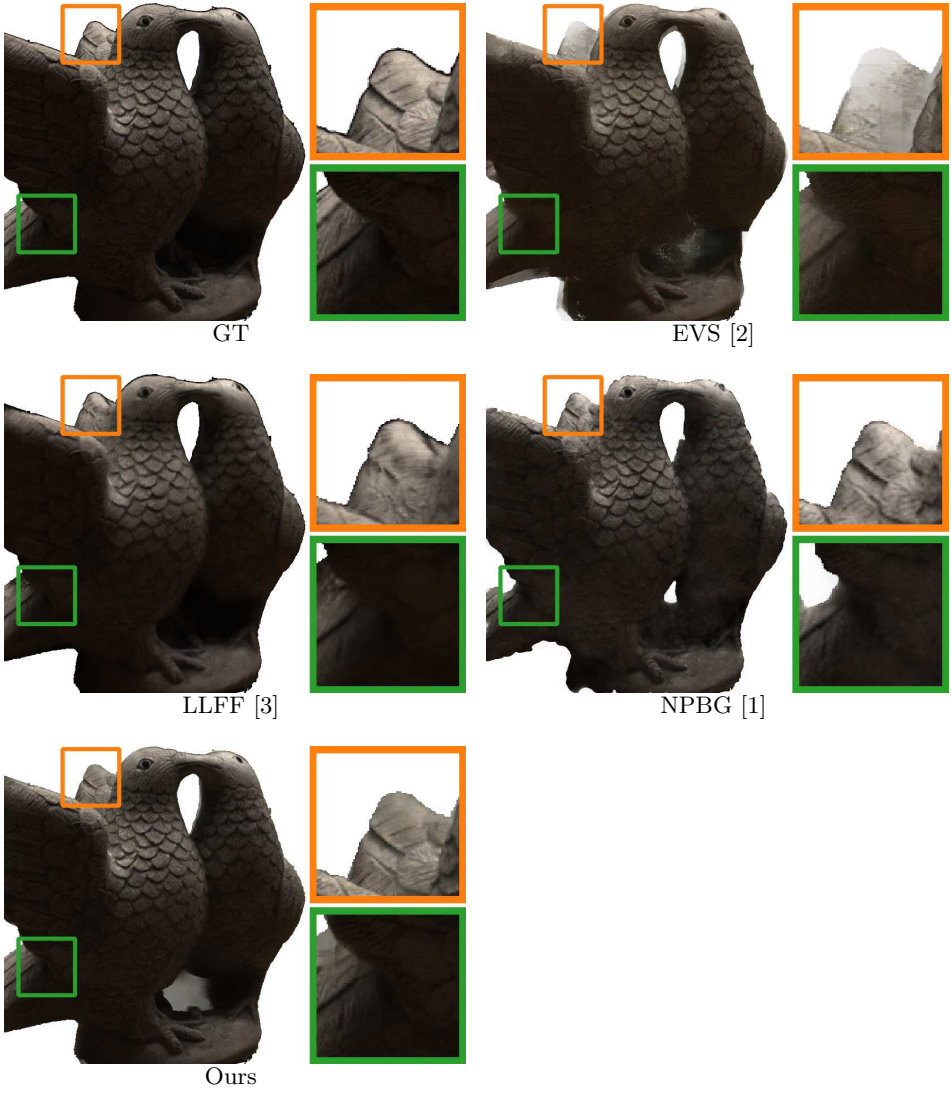
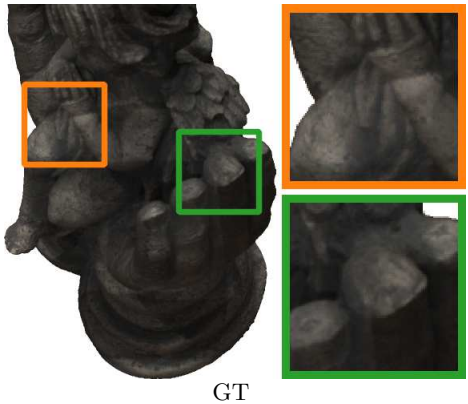
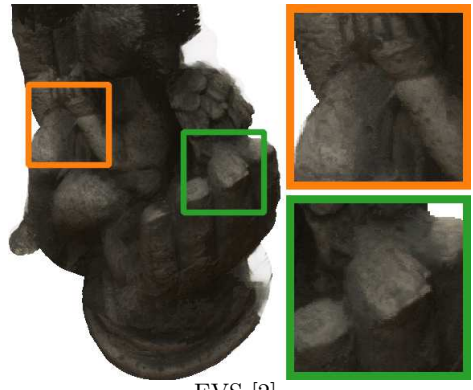
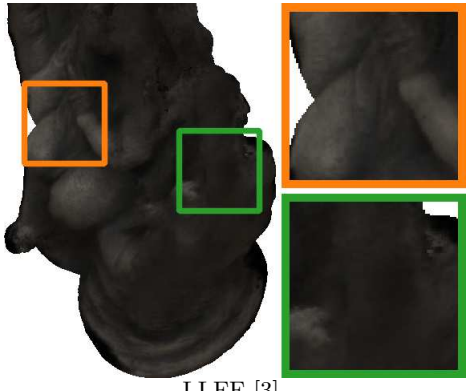


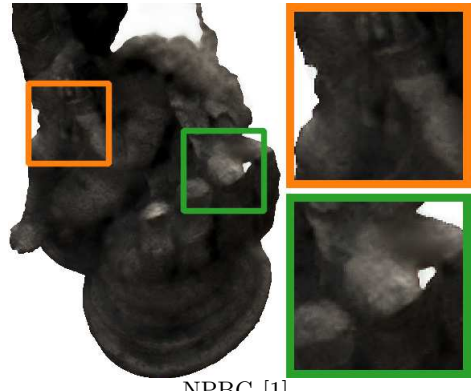
Fig. 20: DTU set 106, view 3, extrapolation

Fig. 21: DTU set 118, view 0, interpolation


GT


EVS [2]


LLFF [3]


NPBG [1]

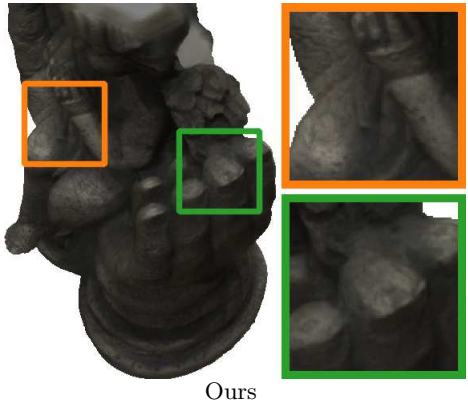

Ours

Fig. 22: DTU set 118, view 1, extrapolation

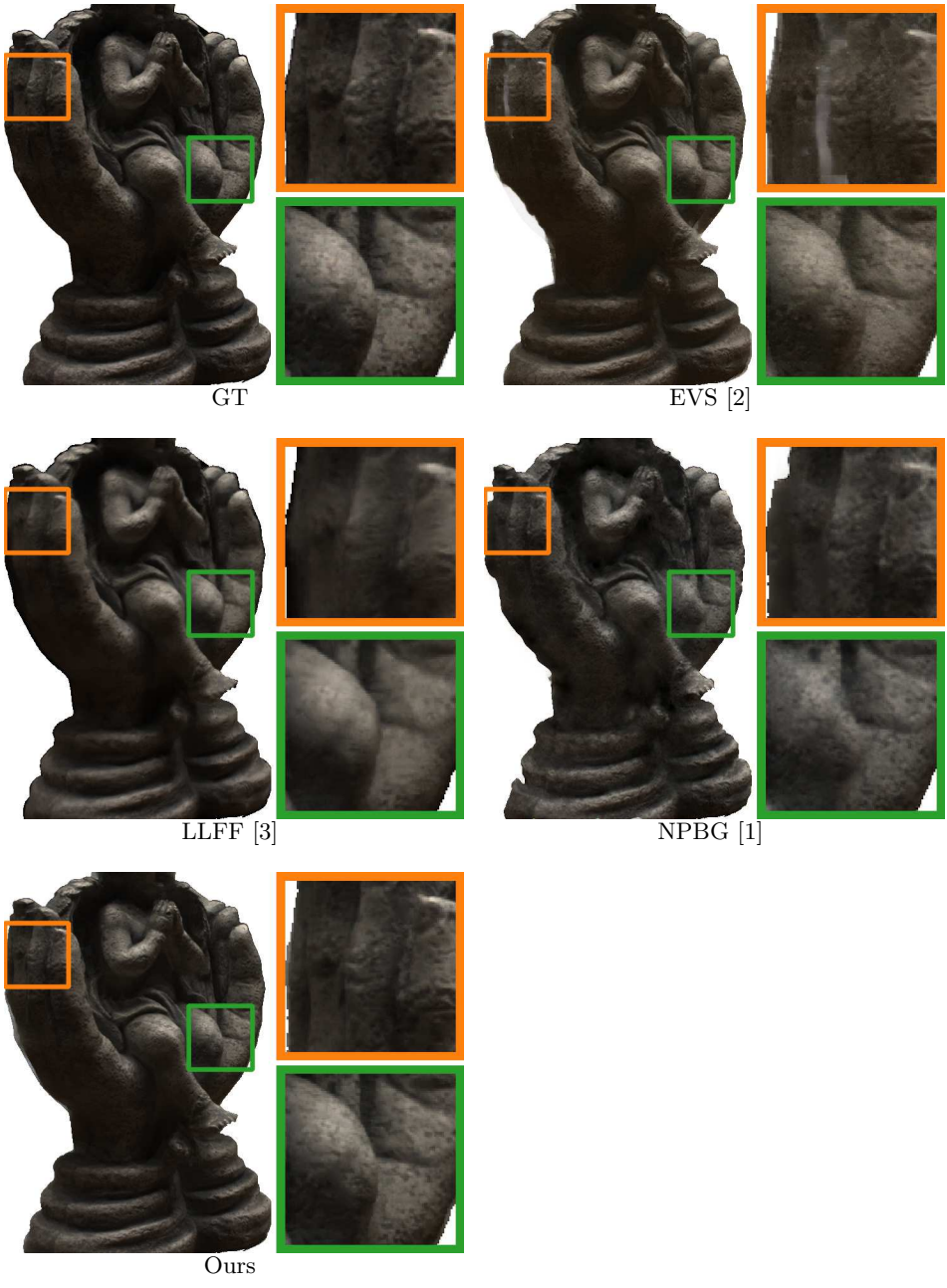


Fig. 23: DTU set 118, view 3, extrapolation

3 Network Details

Our recurrent mapping and blending network for novel view synthesis consists of two main components. First a VGG based encoder that extracts features from the source images, and a GRU based decoder that fuses the mapped features in the target view. Below, we present the PyTorch code of our network architecture. `MappingBlendingRNN` is the recurrent mapping and blending network, whereas `EncoderVGGUNet` is the source image encoder and `DecoderGRUUnet` is the GRU based decoder for blending. The code should give an overview of the number of feature maps, number of layers and layer connectivity. The complete source code of the method will be released upon acceptance.

```
class EncoderVGGUNet(nn.Module):
    def __init__(self, pool="average", n_encs=3, n_dec_convs=2):
        super().__init__()
        self.mean = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1)
        self.std = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1)
        vgg = torchvision.models.vgg19(pretrained=True).features

        encs = []
        enc = []
        encs_channels = []
        channels = -1
        for mod in vgg:
            if isinstance(mod, nn.Conv2d):
                channels = mod.out_channels

            if isinstance(mod, nn.MaxPool2d):
                encs.append(nn.Sequential(*enc))
                encs_channels.append(channels)
                n_encs -= 1
                if n_encs <= 0:
                    break
                if pool == "average":
                    enc = [
                        nn.AvgPool2d(
                            kernel_size=2, stride=2, padding=0, ceil_mode=False
                        )
                    ]
                elif pool == "max":
                    enc = [
                        nn.MaxPool2d(
                            kernel_size=2, stride=2, padding=0, ceil_mode=False
                        )
                    ]
                else:
                    raise Exception("invalid pool")
            else:
                enc.append(mod)
        self.encs = nn.ModuleList(encs)

        cin = encs_channels[-1] + encs_channels[-2]
        decs = []
        for idx, cout in enumerate(reversed(encs_channels[:-1])):
            decs.append(self._dec(cin, cout, n_convs=n_dec_convs))
            cin = cout + encs_channels[max(-idx - 3, -len(encs_channels))]
        self.decs = nn.ModuleList(decs)

    def _dec(self, channels_in, channels_out, n_convs=2):
        mods = []
        for _ in range(n_convs):
            mods.append(
```

```
                nn.Conv2d(
                    channels_in,
                    channels_out,
                    kernel_size=3,
                    stride=1,
                    padding=1,
                    bias=False,
                )
            )
            mods.append(nn.ReLU())
            channels_in = channels_out
        return nn.Sequential(*mods)

    def forward(self, x):
        x = (x + 1) / 2
        self.mean = self.mean.to(x.device)
        self.std = self.std.to(x.device)
        x = (x - self.mean) / self.std

        feats = []
        for enc in self.encs:
            x = enc(x)
            feats.append(x)

        for dec in self.decs:
            x0 = feats.pop()
            x1 = feats.pop()
            x0 = F.interpolate(
                x0, size=(x1.shape[2], x1.shape[3]), mode="nearest"
            )
            x = torch.cat((x0, x1), dim=1)
            x = dec(x)
            feats.append(x)

        x = feats.pop()
        return x


class ConvGRU2d(nn.Module):
    def __init__(
        self,
        channels_x,
        channels_out,
        kernel_size=3,
        padding=1,
        bias=True,
    ):
        super().__init__()
        self.channels_x = channels_x
        self.channels_out = channels_out

        self.conv_gates = nn.Conv2d(
            in_channels=channels_x + channels_out,
            out_channels=2 * channels_out,
            kernel_size=kernel_size,
            padding=padding,
            bias=bias,
        )
        self.conv_can = nn.Conv2d(
            in_channels=channels_x + channels_out,
            out_channels=channels_out,
            kernel_size=kernel_size,
            padding=padding,
            bias=bias,
        )

        self.nonlinearity = nn.ReLU()
```

```
    def forward(self, x, h=None):
        if h is None:
            h = torch.zeros(
                (x.shape[0], self.channels_out, x.shape[2], x.shape[3]),
                dtype=x.dtype,
                device=x.device,
            )
        combined = torch.cat([x, h], dim=1)
        combined_conv = torch.sigmoid(self.conv_gates(combined))
        r = combined_conv[:, : self.channels_out]
        z = combined_conv[:, self.channels_out :]

        combined = torch.cat([x, r * h], dim=1)
        n = self.nonlinearity(self.conv_can(combined))

        h = z * h + (1 - z) * n
        return h


class DecoderGRUUNet(nn.Module):
    def __init__(
        self,
        channels_in,
        enc=[32, 128, 128],
        dec=[128, 32],
        n_enc_convs=2,
        n_dec_convs=2,
        gru_all=False,
        bias=False,
    ):
        super().__init__()
        self.n_rnn = 0

        stride = 1
        cin = channels_in
        encs = []
        for cout in enc:
            encs.append(
                self._enc(
                    cin,
                    cout,
                    stride=stride,
                    n_convs=n_enc_convs,
                    gru_all=gru_all,
                )
            )
            stride = 2
            cin = cout
        self.encs = nn.ModuleList(encs)

        cin = enc[-1] + enc[-2]
        decs = []
        for idx, cout in enumerate(dec):
            decs.append(
                self._dec(
                    cin,
                    cout,
                    n_convs=n_dec_convs,
                    gru_all=gru_all,
                )
            )
            cin = cout + enc[max(-idx - 3, -len(enc))]
        self.decs = nn.ModuleList(decs)

    def _enc(
        self,
        channels_in,
        channels_out,
```

```
        stride=2,
        n_convs=2,
        gru_all=False,
    ):
        mods = []
        if stride > 1:
            mods.append(nn.AvgPool2d(kernel_size=2))
        for idx in range(n_convs):
            if gru_all or idx == n_convs - 1:
                self.n_rnn += 1
                mods.append(
                    ConvGRU2d(
                        channels_in,
                        channels_out,
                        kernel_size=3,
                        padding=1,
                        bias=False,
                    )
                )
            else:
                mods.append(
                    nn.Conv2d(
                        channels_in,
                        channels_out,
                        kernel_size=3,
                        padding=1,
                        bias=False,
                    )
                )
                mods.append(nn.ReLU())
            channels_in = channels_out
            stride = 1
        return nn.Sequential(*mods)

    def _dec(
        self,
        channels_in,
        channels_out,
        n_convs=2,
        gru_all=False,
    ):
        mods = []
        for idx in range(n_convs):
            if gru_all or idx == n_convs - 1:
                self.n_rnn += 1
                mods.append(
                    ConvGRU2d(
                        channels_in,
                        channels_out,
                        kernel_size=3,
                        padding=1,
                        bias=False,
                    )
                )
            else:
                mods.append(
                    nn.Conv2d(
                        channels_in,
                        channels_out,
                        kernel_size=3,
                        padding=1,
                        bias=False,
                    )
                )
                mods.append(nn.ReLU())
            channels_in = channels_out
        return nn.Sequential(*mods)
```

```
    def forward(self, x, hs=None):
        if hs is None:
            hs = [None for _ in range(self.n_rnn)]

        hidx = 0
        feats = []
        for enc in self.encs:
            for mod in enc:
                if isinstance(mod, ConvGRU2d):
                    x = mod(x, hs[hidx])
                    hs[hidx] = x
                    hidx += 1
                else:
                    x = mod(x)
            feats.append(x)

        for dec in self.decs:
            x0 = feats.pop()
            x1 = feats.pop()
            x0 = F.interpolate(
                x0, size=(x1.shape[2], x1.shape[3]), mode="nearest"
            )
            x = torch.cat((x0, x1), dim=1)
            for mod in dec:
                if isinstance(mod, ConvGRU2d):
                    x = mod(x, hs[hidx])
                    hs[hidx] = x
                    hidx += 1
                else:
                    x = mod(x)
            feats.append(x)

        x = feats.pop()


class MappingBlendingRNN(nn.Module):
    def __init__(self, enc, mapper, dec, merge_channels_out):
        super().__init__()
        self.enc = enc
        self.mapper = mapper
        self.dec = dec

        self.rgb_conv = nn.Conv2d(
            merge_channels_out,
            3,
            kernel_size=1,
            stride=1,
            padding=0,
            bias=False,
        )
        self.alpha_conv = nn.Conv2d(
            merge_channels_out,
            1,
            kernel_size=1,
            stride=1,
            padding=0,
            bias=False,
        )

    def forward(self, x, K0, R0, t0, K, R, t, dm0, nb_dms):
        bs, nv, channels, height, width = x.shape

        x = x.view(bs * nv, *x.shape[-3:])
        x = self.enc(x)
        x = x.view(bs, nv, *x.shape[-3:])

        with torch.no_grad():
            K0i = torch.inverse(K0)
```

```
            P0i = torch.bmm(
                R0.transpose(2, 1), torch.cat((K0i, -t0.view(bs, 3, 1)), dim=2)
            )
            P = torch.matmul(K, torch.cat((R, t.view(bs, nv, 3, 1)), dim=3))
        x = self.mapper(x, dm0, nb_dms, P, P0i)

        hs = None
        rgbs = []
        alphas = []
        for vidx in range(nv):
            y, hs = self.dec(x[:, vidx], hs)
            rgbs.append(self.rgb_conv(y))
            alphas.append(self.alpha_conv(y))

        rgbs = torch.stack(rgbs)
        alphas = torch.stack(alphas)
        alphas = torch.softmax(alphas, dim=0)
        x = (alphas * rgbs).sum(dim=0)
        del rgbs, alphas
        return {"rgb": x}
```

References

1. Aliev, K.A., Ulyanov, D., Lempitsky, V.: Neural Point-Based Graphics. arXiv preprint arXiv:1906.08240 (2019)
2. Choi, I., Gallo, O., Troccoli, A., Kim, M.H., Kautz, J.: Extreme View Synthesis. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV) (2019)
3. Mildenhall, B., Srinivasan, P.P., Ortiz-Cayon, R., Kalantari, N.K., Ramamoorthi, R., Ng, R., Kar, A.: Local Light Field Fusion: Practical View Synthesis with Prescriptive Sampling Guidelines. ACM Transactions on Graphics (SIGGRAPH) (2019)