High-Performance Graphics (2025)
A. Knoll and C. Peters (Editors)

Supplemental Material: Collaborative Texture Filtering

T. Akenine-Möller, P. Ebelin, M. Pharr, and B. Wronski

NVIDIA

This supplemental material includes:

- Detailed code showing example implementations of the Box Sampling and Mask Sampling algorithms.
- Details on the edge remapping technique used at silhouette edges where not all of the wave's lanes are active.
- More information about the scenes, textures, and metrics used for evaluation.
- Additional evaluation and results.

S.1. Code

For simplicity, we omit clamping of out-of-bounds texture coordinates in the code below and we do not include an implementation of List Merge, since it performs poorly and is not a realistic alternative for use in real applications. (Recall that it was mostly included because it computes the number of texels needed for perfect filtering exactly, which was useful for evaluating the other approaches.)

S.1.1. Box Sampling

The following helper functions are used in the implementation of Box Sampling for bilinear filtering, which follows.

```
int2 LaneIdxToCoord(uint laneIdx,
          int2 waveUpperLeftIntCoords,
          uint bbWidth)
{
    uint laneY = laneIdx / bbWidth;
    uint laneX = laneIdx % bbWidth;
    return waveUpperLeftIntCoords + int2(laneX, laneY);
}

uint CoordToLaneIdx(int2 coord,
          int2 waveUpperLeftIntCoords,
          uint bbWidth)
{
    coord -= waveUpperLeftIntCoords;
    return coord.x + coord.y * bbWidth;
}

bool LanesLowerThanCountActive(uint count)
{
    uint activeLanesMask = WaveActiveBallot(true).x;
    uint desiredActiveMask = (count == 32) ? 0xFFFFFFFF :
        (1 << count) - 1;

    return (activeLanesBitMask & desiredActiveMask) ==
      desiredActiveMask;
}
```

Inside the shader that accesses the texture, the following code implements Box Sampling:

```
uint2 txDim;
texture.GetDimensions(txDim.x, txDim.y);
float2 floatCoords = uv * txDim - float2(0.5f);
int2 upperLeftIntCoords = int2(floor(floatCoords));
int2 lowerRightIntCoords = upperLeftIntCoords + int2(1, 1);
float2 stCoords = floatCoords - upperLeftIntCoords;

// Compute bounding box of texel integer coordinates
int2 waveUpperLeftCoords = WaveActiveMin(upperLeftIntCoords);
int2 waveLowerRightCoords = WaveActiveMax(lowerRightIntCoords);
int2 bbSize = waveLowerRightCoords - waveUpperLeftCoords + 1;

int activeTexelsNeeded = bbSize.x * bbSize.y;
bool requiredLanesActive = LanesLowerThanCountActive(
      activeTexelsNeeded);

if (activeTexelsNeeded > 32 || !requiredLanesActive)
  return fallBackMethod();

uint curLaneIdx = WaveGetLaneIndex();
float4 texelValue = float4(0.0f);

if (curLaneIdx <= activeTexelsNeeded) {
  uint2 texCoords = LaneIdxToCoord(curLaneIdx,
                                   waveUpperLeftCoords, bbSize.x);
  texelValue = texture[texCoords];
}

float4 bilinWeights = computeBilinearWeights(stCoords);

float4 filteredColor = float4(0.0f);
[unroll]
for (int i = 0; i < 4; i++) {
  int2 texelCoords = int2(upperLeftIntCoords.x + (i % 2),
                          upperLeftIntCoords.y + (i / 2));
  uint laneIdx = CoordToLaneIdx(texelCoords, waveUpperLeftCoords,
      bbSize.x);
  filteredColor += WaveReadLaneAt(texelValue, laneIdx) *
      bilinWeights[i];
}
return filteredColor;
```

The first six lines of code compute various coordinates from the texture dimensions and (u,v)-coordinates. Among these are the upper left coordinates of the 2×2 filter footprint needed for bilinear filtering and (s,t)-coordinates which are local coordinates inside the 2×2 region and are in $[0,1]$. Next, we use `WaveActiveMin()` and `WaveActiveMax()` to compute an axis-aligned bounding box over the texel coordinates that are needed for the entire wave.

Next, we check whether the number of needed texels is less than

or equal to the number of active lanes in the wave and if all of the lanes are active. For each texel-producing lane, we then use a simple linearly-ordered mapping `LaneIdxToCoord()` to compute its texel coordinates. Producing a texel in this example is done here via a regular texel fetch using `texture[]`. Finally, we iterate over the texture filtering footprint, gathering and accumulating the necessary texels using the `CoordToLaneIdx()` and `WaveReadLaneAt()` functions. We note that the same algorithm works for any texture filter: the only changes required are the calculations of `upperLeftIntCoords`, `lowerRightIntCoords`, `computeBilinearWeights`, and the final unrolled loop extent.

S.1.2. Mask Sampling

Code for Mask Sampling with a 16×16 mask is shown below.

```
1  uint2 txDim;
2  texture.GetDimensions(txDim.x, txDim.y);
3  floatCoords = uv * txDim - float2(0.5f);
4  int2 upperLeftIntCoords = int2(floor(floatCoords));
5  int2 lowerRightIntCoords = upperLeftIntCoords + int2(1, 1);
6  float2 stCoords = floatCoords - upperLeftIntCoords;
7
8  // Compute bounding box of texel integer coordinates
9  int2 waveUpperLeftCoords = WaveActiveMin(upperLeftIntCoords);
10 int2 waveLowerRightCoords = WaveActiveMax(lowerRightIntCoords);
11 int2 bbSize = waveLowerRightCoords - waveUpperLeftCoords + 1;
12 uint2 deltaCoords = upperLeftIntCoords - waveUpperLeftIntCoords;
13
14 if (bbSize.x > 16 || bbSize.y > 16)
15   return fallBackMethod();
16
17 // Set the 2x2 bits corresponding to the 2x2 texels needed.
18 uint64_t4 mask = uint64_t4(0);
19 set2x2Bits(upperLeftIntCoords - waveUpperLeftIntCoords, mask);
20
21 // Compute the ORed mask across the entire wave.
22 uint64_t4 waveMask;
23 waveMask.x = WaveActiveBitOr(mask.x);
24 waveMask.y = WaveActiveBitOr(mask.y);
25 waveMask.z = WaveActiveBitOr(mask.z);
26 waveMask.w = WaveActiveBitOr(mask.w);
27
28 uint curLaneIdx = WaveGetLaneIndex();
29 uint activeTexelsNeeded = countbits(waveMask);
30 uint activeLanesMask = WaveActiveBallot(true).x; // # active lanes
31
32 if (activeTexelsNeeded > countbits(activeLanesBitMask))
33   return fallBackMethod();
34
35 float4 curPixelTexelValue;
36 int2 sampledTexelIntCoords;
37 float4 color = float4(0.0f);
38 float4 bilinWeights = computeBilinearWeights(stCoords);
39
40 // Does this lane needs to produce a texel?
41 if (curLaneIdx <= activeTexelsNeeded) {
42   uint2 deltaTexCoords = bijectiveFunctionH(curLaneIdx, waveMask);
43   sampledTexelIntCoords = waveUpperLeftIntCoords + deltaTexCoords;
44   curPixelTexelValue = texture[sampledTexelIntCoords];
45 }
46
47 // Compute 1D local texture coordinate:
48 uint t = (deltaCoords.y << 4) + deltaCoords.x;
49 // Read the 2x2 texels needed for this pixel and weight together.
50 uint i0 = inverseBijectiveFunctionH(t, waveMask);
51 color += bilinWeights.x * WaveReadLaneAt(curPixelTexelValue, i0);
52 uint i1 = inverseBijectiveFunctionH(t+1, waveMask);
53 color += bilinWeights.y * WaveReadLaneAt(curPixelTexelValue, i1);
54 uint i2 = inverseBijectiveFunctionH(t+16, waveMask);
55 color += bilinWeights.z * WaveReadLaneAt(curPixelTexelValue, i2);
56 uint i3 = inverseBijectiveFunctionH(t+16+1, waveMask);
57 color += bilinWeights.w * WaveReadLaneAt(curPixelTexelValue, i3);
58 return color;
```

The first six lines of code are the same as for Box Sampling. The mask is stored in an `uint64_t4`, which has $16 \cdot 16 = 256$ bits, and so lines 14–15 call the fallback method if either of the bounding box dimensions is larger than 16. Lines 18–26 first set the 2×2 bits of the texels needed by the current lane in `mask` using a function `set2x2Bits()`; then it is OR of all of these over the entire wave using `WaveActiveBitOr()` that gives the full wave mask, `waveMask`.

Lines 28–33 compute how many texels are needed by counting the set bits in `waveMask` and use `WaveActiveBallot(true)` to find a bitmask of the active lanes in the wave. As with Box Sampling, the fallback method is called if the number of texels needed is more than the number of active lanes in the wave. Past this point, we know there are a sufficient number of active lanes and that Mask Sampling will succeed at getting all 2×2 texels needed by each active lane in the wave to perform perfect bilinear filtering.

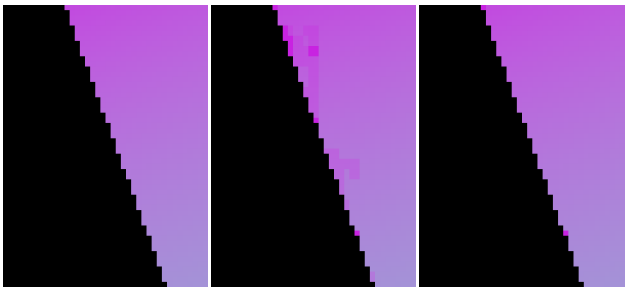
In lines 41–45, the lanes with lane number less than the number of needed texels produce a texel, which in this example simply does a texture lookup using the GPU's texture unit. The bijective function $h(i,B)$ (Figure 3) is used to compute local texture coordinates, which are added to the wave's upper left coordinates and the lookup is performed at the end. Line 48 computes the local one-dimensional texture coordinate of the upper left texel for the current lane. Since we know that the current lane wants the 2×2 texels starting from `upperLeftIntCoords`, we can transform that to 2×2 one-dimensional texture coordinates as $\{t, t+1, t+16, t+16+1\}$, where $t+1$ identifies the texel to the right of t, and $t+16$ the texel below t, since the mask is 16×16 bits. These coordinates are then fed to the inverse of the bijective function, i.e., $h^{-1}(t,B)$, and then the texel is produced from that lane and weighted, with perfect bilinear filtering as a result.

Similar to Box Sampling, we note that the algorithm above works for any texture filter, only requiring changes to the calculations of `upperLeftIntCoords`, `lowerRightIntCoords`, `computeBilinearWeights`, `set2x2Bits`, and the final unrolled loop.

S.2. Edge Remapping

Close to a triangle edge against the background, a wave will typically not have all of its lanes active. So far, we have assumed that at least the first n lanes are active when n texels are needed and so the mappings from lanes to texels in Box and Mask Sampling may end up with inactive lanes being assigned to generate texels but then not actually doing so. In such cases, we may use the fallback method introduced in Section 3.4 of the main paper, though that can give artifacts as shown in the middle in Figure S1.

Alternatively, we can map lanes to texels more carefully, accounting for inactive lanes. Assume for now that the size of a wave is 8: if all lanes are active, then we have the active lane mask `11111111`. In that case Mask Sampling, for example, can simply use its bijective function, $t = h(i,B)$, to map a lane index into a texel coordinate. This fails if the active lane mask is, for example, `11101010`. Still, if only four lanes are needed, then we can see that since the number of bits in the mask is ≥ 4, it should be possible to run the algorithm anyway.



bilinear ground truth | Mask Sampling w/o edge remapping | Mask Sampling with edge remapping

Figure S1: *Left: bilinear ground truth filtering near a triangle edge. Middle: if we require the first n lanes to be active if n texels are needed, then the fallback is often used and results may be noisy. Right: with the remapping technique described in Section S.2, we only need n active lanes across the wave and this problem is largely eliminated.*

If we enumerate the needed texels from 0 to $n-1$ with a number i, then we need a way to map that number into a lane number in the active lane mask where there is a `1`. This can be done with the same bijective function as is used in Mask Sampling. We simply use $h()$ once again, i.e., we remap so the texel number is remapped to lane index as $h(i,A)$, where A is the active lane mask. This technique can also be applied to Box Sampling.

On the right side of Figure S1, we present the result of our edge remapping, which substantially improves the image quality. However, it is still sometimes necessary to use the fallback method. This can happen, for example, when two texels are needed (e.g., due to clamping), but there is only a single active lane in the wave.

S.3. Additional Evaluation Details and Results

In this section, we provide specifics about the scenes and textures used in the paper as well as the image quality metrics we used. We also present additional quality results.

S.3.1. Scenes and Textures

We used two primary scenes and six textures for evaluation.

The first test scene consisted of a single quad, and a camera positioned directly above the quad's center, viewing it head on. The degrees of freedom used for this scene was the camera's distance to the quad, which set the effective magnification factor, the quad's rotation around the axis corresponding to the camera's viewing direction, as well as the texture used for the quad. The quad was shaded with a simple shading algorithm that only considered diffuse and normal textures, with lighting defined by a single light vector. This scene was used for Figures 4, 6, S4, S5, S7, S6, and S8, as well as Table S1.

To analyze the behavior of our algorithms with partially active waves and texture discontinuities, our second scene included 4×4 tessellated spheres in addition to the quad. The camera started some distance away from the quad so that a small amount of minification occurred, and then moved toward the spheres and back. The spheres and the quad used the same texture, though the version used on the spheres was a low-resolution version of the texture, so that we would have magnification also on the spheres. Each mesh in this scene was shaded using the same shading algorithm as the quad in the first scene. In total, the camera animation was 6 seconds long and rendered at 60 frames per second. It is shown in our supplemental video. This scene was used for Figures 5 and S3, as well as Table 1. A version similar to this scene, but with different camera animations and number of spheres, was used for Figures 1 and 8.

The resolution of the images we rendered was 1600×1600, and the field of view of the camera was 45 degrees.

The five textures used throughout the results section of this work were the diffuse and normal textures for the AERIALROCKS, BRICKS, DIRT, PAINTEDCONCRETE, and RAILS sets. This set of textures shows variety in both diffuse and normal content, including both high- and low-frequency information. Each texture image has a resolution of 4096×4096 and we present them in Figure S2.

S.3.2. Metric Specifics

Our results contain PSNR values as well as output from the ColorVideoVDP [MHA*24] and ꟻLIP metrics [ANA*20].

For single images, PSNR was computed as usual by

$$\mathrm{PSNR}(\mathbf{G},\mathbf{T}) = -10\log\left(\mathrm{MSE}(\mathbf{G},\mathbf{T})\right), \tag{S1}$$

where $\mathbf{G}$ is the ground-truth image, $\mathbf{T}$ is the test image, and MSE is mean-squared error. Both ground-truth and test images were in linear sRGB space. In cases where we have N image sequences, each containing M frames, we take the average of the individual frame pairs' MSEs before computing PSNR. That is, if $\hat{\mathbf{G}}^i = \{\mathbf{G}^i_j\}_{j=0}^{M-1}$ and $\hat{\mathbf{T}}^i = \{\mathbf{T}^i_j\}_{j=0}^{M-1}$ are the sets of M ground-truth and test frames for sequence i, $i \in \{0,1,\ldots,N-1\}$, we compute PSNR as

$$\begin{aligned}&\mathrm{PSNR}\left(\{\hat{\mathbf{G}}^i\}_{i=0}^{N-1}, \{\hat{\mathbf{T}}^i\}_{i=0}^{N-1}\right)\\ &= -10\log\left(\frac{1}{NM}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\mathrm{MSE}(\mathbf{G}^i_j,\mathbf{T}^i_j)\right).\end{aligned} \tag{S2}$$

Notice that the computation in Equation S2 gives the same result as that in Equation S1 when only a single frame pair is considered.

The ColorVideoVDP model requires information about the assumed observer's viewing conditions and display. We use the default settings, and set the assumed number of frames displayed per second to 60 for all our image sequences. The reproducibility information provided when running the metric was "ColorVideoVDP v0.4.2, 75.4 [pix/deg], Lpeak=200, Lblack=0.2, Lrefl=0.3979 [cd/m^2], (standard_4k)." ColorVideoVDP's output is in Just-Objectionable Differences (JOD) units, scaled to have a maximum value of 10. Only if the reference and test images/sequences are visually indistinguishable under the assumed viewing conditions is the JOD value of the test equal to 10. In cases where we compute one JOD over a set of image sequences, that JOD is the average of the JODs for each individual sequence. Using the

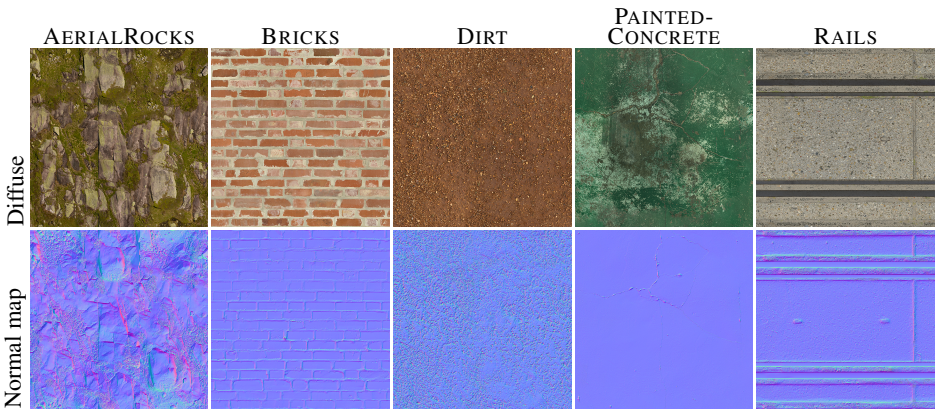


Figure S2: *The textures used in our image quality measurements.*

notation introduced above, we have

$$\begin{aligned}&\text{ColorVideoVDP}\left(\{\hat{\mathbf{G}}^i\}_{i=0}^{N-1},\{\hat{\mathbf{T}}^i\}_{i=0}^{N-1}\right)\\&=\frac{1}{N}\sum_{i=0}^{N-1}\text{ColorVideoVDP}\left(\hat{\mathbf{G}}^i,\hat{\mathbf{T}}^i\right).\end{aligned}\tag{S3}$$

Note that ColorVideoVDP is also able to compare single images. We use this capability for the results in Figure 6.

Like ColorVideoVDP, ꟻLIP requires information about the assumed observer's viewing conditions. In particular, it takes the observer's distance to the display as well as the display's width (in meters and in pixels). To match the assumptions for ColorVideoVDP and ꟻLIP, we use the information provided by ColorVideoVDP for its `standard_4k` display, namely a distance to display of 0.7472 meters and a display width of 0.664 meters and 3840 pixels. As ꟻLIP acts on single images, and not image sequences, the ꟻLIP error we present for multiple image sequences is the average of each frame pairs' ꟻLIP error, i.e.,

$$\begin{aligned}&\text{ꟻLIP}\left(\{\hat{\mathbf{G}}^i\}_{i=0}^{N-1},\{\hat{\mathbf{T}}^i\}_{i=0}^{N-1}\right)\\&=\frac{1}{NM}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\text{ꟻLIP}(\mathbf{G}_j^i,\mathbf{T}_j^i).\end{aligned}\tag{S4}$$

S.3.3. Quality Comparison Without Denoising

In this section, we present diagrams showing quality/performance efficiency and the quality as a function of magnification corresponding to the main paper's Figures 5 and 6, but without denoising the evaluated sequences. The results are shown in Figures S3 and S4. Furthermore, in Figure S5 and Table S1, we examine how the maximum image error differs between the considered algorithms.

We start by noting that the same methods lie on the Pareto frontier, independent of whether the sequences are denoised (Figures 5 and S3). Furthermore, in Figure S4, as also shown in Figure 4, we again see that our algorithms provide perfect bilinear filtering above a magnification threshold (1.59 for Mask Sampling and 2.35 for Box Sampling), while the One-tap STF [PWSF24] and Wave Communication STF [WPAM25] algorithms are unable to achieve

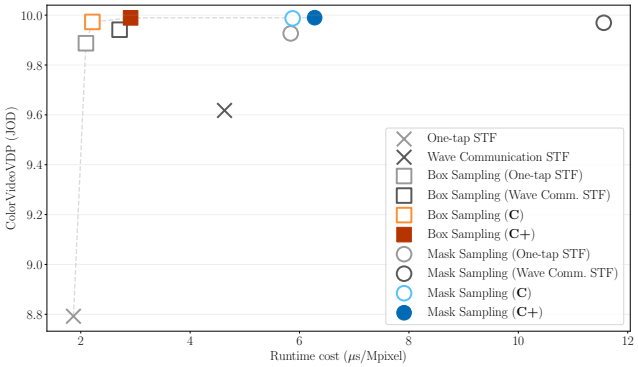


Figure S3: *Pareto frontier (dashed line) indicating the most quality/performance-efficient algorithms when rendering without denoising. Runtime cost was measured on an NVIDIA RTX 5090. Our new fallback algorithms are marked with* ***C*** *and* ***C+****. The PSNR range for this plot was 29–59 dB.*

that for any magnification factor due to their inherently stochastic nature. At lower magnification factors, where our algorithms are unable to reach perfect filtering and must use fallback alternatives, they still give higher quality than the prior techniques. (A comparison between our fallback methods and the state-of-the-art algorithms is included in Section S.3.5.) Furthermore, both Figures S3 and S4 show that Mask Sampling yields higher-quality results than Box Sampling. We also confirm our earlier findings that the Mask Sampling quality increase comes at a performance trade-off, as Box Sampling is the faster of the two.

Finally, we consider the plots in Figure S5 and the results in Table S1, where the plots show the *maximum* errors produced by the algorithms we consider, and the table shows the averages of the curves in those plots. We generated these results using the same simple magnification setup as explained in Section S.3.1, with the quad rotating between $[0,90]$ degrees for each measurement. For low magnification, Box and Mask Sampling perform the same (the Box Sampling curves in Figure S5a are offset slightly for visibil-

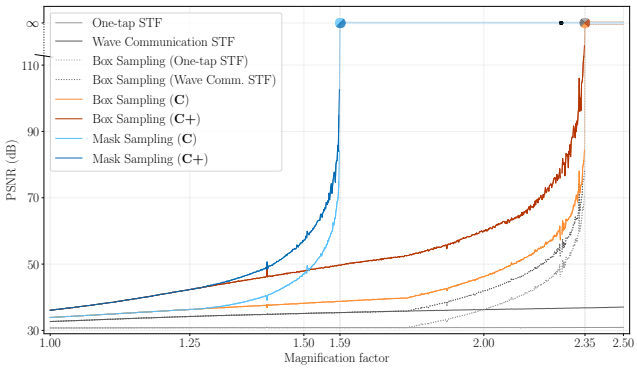


Figure S4: *Quality as a function of magnification, without denoising. As indicated by Figure 4, perfect bilinear filtering is achieved above magnification factor* 2.35 *for our methods. Colored discs mark that an algorithm has achieved perfect bilinear filtering at the corresponding magnification factor, resulting in infinite PSNR. At that and higher magnification factors, perfect bilinear filtering occurs for that algorithm. The small black circle marks a case when perfect bilinear filtering is achieved at a magnification lower than the limits indicated by the disks. This situation can arise for certain magnification and rotation combinations.*

ity). As magnification increases, Mask Sampling starts producing lower errors than Box Sampling, when both use the same fallback method. At a magnification factor of 1.55, Mask Sampling is able to produce perfect bilinear filtering for most rotations, while Box Sampling requires higher magnification to do so (see Figure 4). The more complex fallback method (**C+**) often give lower maximum errors than the simpler, faster one. These results agree with Figure S4.

S.3.4. Convergence Analysis

We also analyzed whether or not the output of our algorithms converges to the ground-truth image when we increase the number of samples drawn per pixel (SPP), or how large the bias is if they do not; see Figure S6, which shows PSNR as a function of SPP. For these results, we once again use the same simple, rotated quad and set of textures described in Section S.3.1. For a given SPP-value, PSNR was computed over a set of magnification factors in the $[1.0, 2.5]$ range, as that is approximately the range where one or more of our algorithms are unable to produce perfect bilinear filtering (see Figure 4). Figure S6 shows that our algorithms achieve better results, though they also show some bias, similar to One-tap STF [PWSF24] and Wave Communication STF [WPAM25]. This is a consequence of the difference between filtering after shading and filtering before shading, as discussed by Pharr et al. [PWSF24].

S.3.5. Fallback Evaluation

In extreme cases, every wave could contain a silhouette edge, which means that the fallback (Section 3.4) would be called for every pixel. We therefore present image quality results for our fallback methods as a function of magnification factor in Figure S7, both without and with denoising. Even our simplest proposed fallback (**C**) provides better image quality than both One-tap STF [PWSF24] and Wave Communication STF [WPAM25]. We also note that without DLSS (left diagram), our **C+** method continues to increase the image quality for higher magnification factors, while that happens to a lesser extent for the other methods. This is a consequence of **C+** leveraging unused lanes to produce more unique texels and as magnification increases there are more and more unused lanes that can be used to increase image quality. For the comparisons with denoised sequences, since we apply denoising both to the ground-truth and the test sequence and the latter almost is identical to the former (see the high PSNR values for high magnification in Figure S7a), the small errors become slightly larger, spatially, due to the slight blur caused by the denoiser. Presumably, this results in lower PSNR values for the **C+** method after denoising compared to before. Although the dB difference caused by these differences is large, the per-pixel errors in the denoised sequences are very small: at high magnification, the largest errors for the **C+** method are 1–2 when scaled to the $[0, 255]$ range. However, we note that the denoised results for **C+** show significantly worse quality than the non-denoised ones around magnification factor 7–8. While inspecting the data, we found that the quality decrease originated from the RAILS results, where errors became slightly larger for those magnification factors despite the image content not changing drastically. This was due to the denoised **C+** images being marginally more blurry than the ground truth at those magnification factors. For the other textures, this effect was not observed.

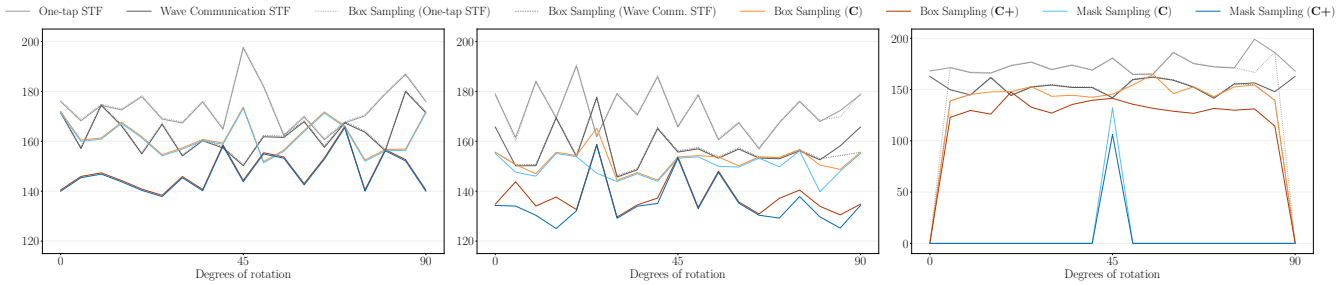
S.3.6. Additional Bicubic Results

Finally, we measured the amount of magnification that is needed to achieve perfect bicubic filtering with each of List Merge, Box Sampling, and Mask Sampling, given 32 lanes per wave. Figure S8a is the bicubic counterpart to Figure 4. That is, it shows how much magnification is needed at different rotations of a quad to achieve perfect bicubic filtering with 32 or fewer texture lookups per wave. Figure S8b is similar, but assumes that we are able to do two texture lookups per lane, so that we instead need 64 or fewer lookups per wave. The figures show that perfect bicubic filtering with our methods requires significantly more magnification if we only allow one lookup per pixel. If two lookups are allowed, we see that the required magnification factor is closer to that for perfect bilinear filtering, despite the four times larger filter area.

Acknowledgments

Many thanks to Tomáš Davidovič for help with details of Falcor internals and to Johannes Deligiannis for help with integrating our techniques with the NTC SDK. Magnus Andersson, Rasmus Barringer, Anders Lindqvist, James Player, and Robert Toth all helped with code optimizations. Thanks also to Aaron Lefohn and NVIDIA Research for supporting this work.

Thanks to *PolyHaven* for the Aerial Rocks, Dirt, and Painted Concrete texture sets and to *ambientCG* for the Bricks and Rails texture sets.



(a) *Magnification factor: 1.15* **(b)** *Magnification factor: 1.35* **(c)** *Magnification factor: 1.55*

Figure S5: *Maximum errors (scaled to* $[0,255]$*) produced by the compared algorithms for rotations by* $r \in [0,90]$ *degrees, given three different levels of magnification. The legend is included above the plots. Note that the y-axis for the rightmost plot differs from that in the other two. The errors from Box and Mask Sampling in Figure S5a are the same, but the curves for Box Sampling are offset by 0.5 for visibility. As implied by Figure 4, at magnification factors larger than 1.59, the maximum error of Mask Sampling is zero for all rotations, while Box Sampling requires more magnification (2.35 or higher) to produce perfect bilinear filtering.*

Table S1: *Average* maximum *errors (scaled to* $[0,255]$*) across the rotations used in Figure S5, for each of the different magnification factors used in that figure. One-tap STF is the algorithm by Pharr et al. [PWSF24], Wave Comm. is short for Wave Communication STF and is the algorithm by Wronski et al. [WPAM25], Box is short for Box Sampling, and Mask is short for Mask Sampling.*

Zoom	One-tap STF	Wave Comm.	Box (One-tap STF)	Box (Wave Comm.)	Box (**C**)	Box (**C+**)	Mask (**C**)	Mask (**C+**)
1.15	174	163	174	163	161	**147**	161	**147**
1.35	172	157	172	156	152	137	150	**135**
1.55	174	153	154	135	132	117	7	**6**

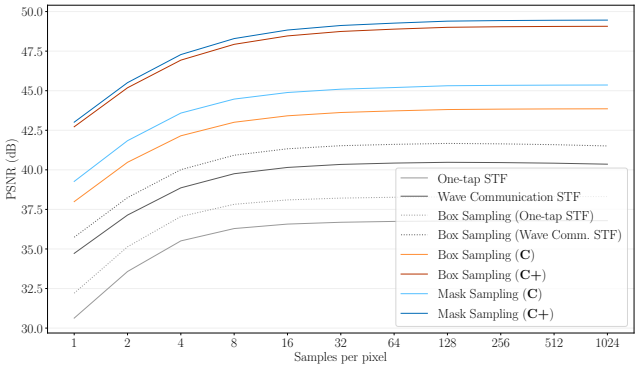


Figure S6: *Plot of the error convergence of the considered algorithms when compared to perfect bilinear filtering. The scene used was the first scene described in Section S.3.1. Ten magnification factors in the range* $[1.0,2.5]$ *were used for each PSNR value.*

References

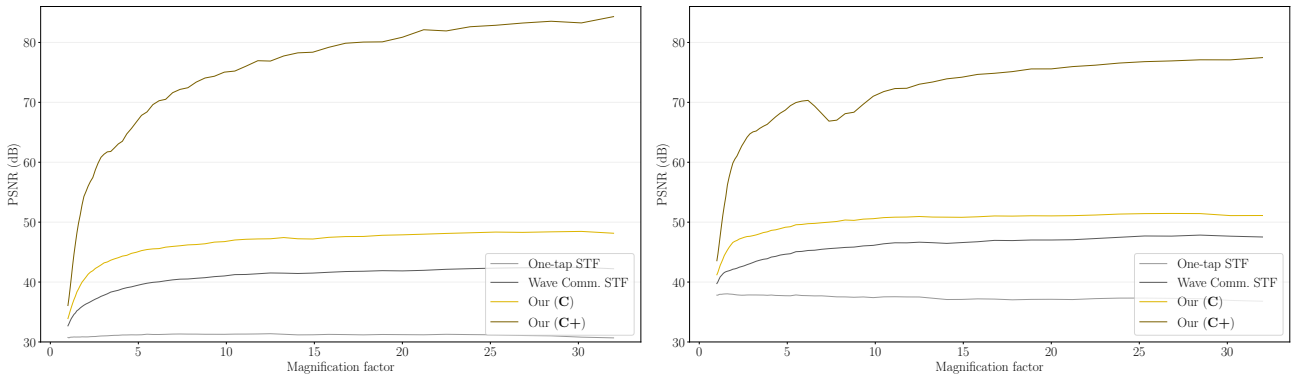
[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–23. 3

[MHA*24] MANTIUK R. K., HANJI P., ASHRAF M., ASANO Y., CHAPIRO A.: ColorVideoVDP: A Visual Difference Predictor for Image, Video and Display Distortions. *ACM Transactions on Graphics 43*, 4 (2024), 129:1–20. 3

[NVI25] NVIDIA: DLSS 4: Transforming Real-Time Graphics with AI. https://research.nvidia.com/labs/adlr/DLSS4/, 2025. Technical Report. 7

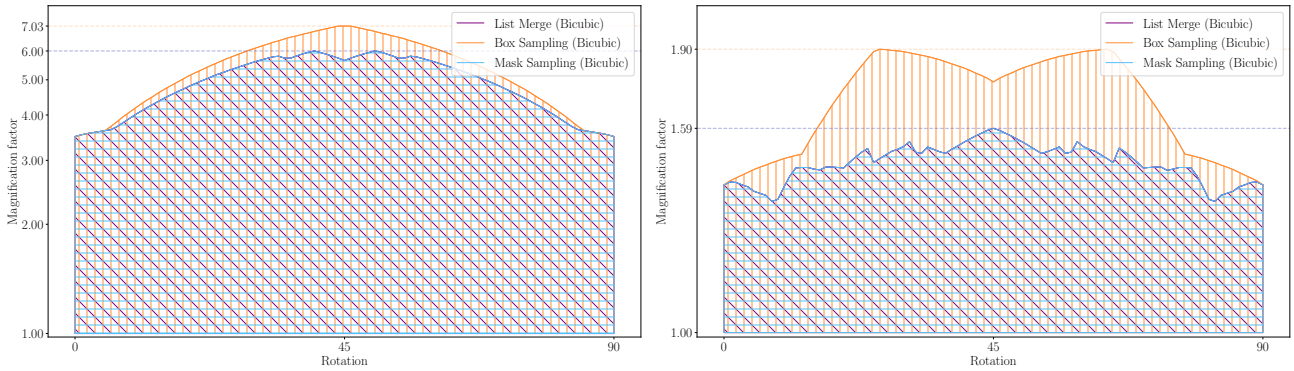
[PWSF24] PHARR M., WRONSKI B., SALVI M., FAJARDO M.: Filtering After Shading with Stochastic Texture Filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 7*, 1 (2024), 14:1–20. 4, 5, 6

[WPAM25] WRONSKI B., PHARR M., AKENINE-MÖLLER T.: Improved Stochastic Texture Filtering Through Sample Reuse. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 8*, 1 (2025). arXiv:2504.05562. 4, 5, 6



(a) *Not denoised.* **(b)** *Denoised.*

Figure S7: *Evaluation of image quality as a function of magnification factor for all the different fallback methods. For both without and with denoising (DLSS [NVI25]), our algorithms provide superior quality compared to the other methods.*



(a) *Texel lookups per lane: 1.* **(b)** *Texel lookups per lane: 2.*

Figure S8: *Illustration of the minimum degree of magnification necessary to achieve perfect filtering under different rotations of the quad for perfect bicubic filtering. Below those magnification factors, our algorithms need to rely on fallback methods (Section 3.4). Cases where a fallback was necessary are indicated by colored areas. The scene used was the first one described in Section S.3.1. The results in the left diagram are based on producing no more than one texel per lane, while those in the right diagram allow up to two texels per lane. Notice that List Merge and Mask Sampling both cover identical areas in both figures. Like in the bilinear case, the success rate of Box Sampling is lower than that of the other two techniques.*