6. Appendix

Here we present detailed algorithms for creating the texture hint, as well as additional evaluation.

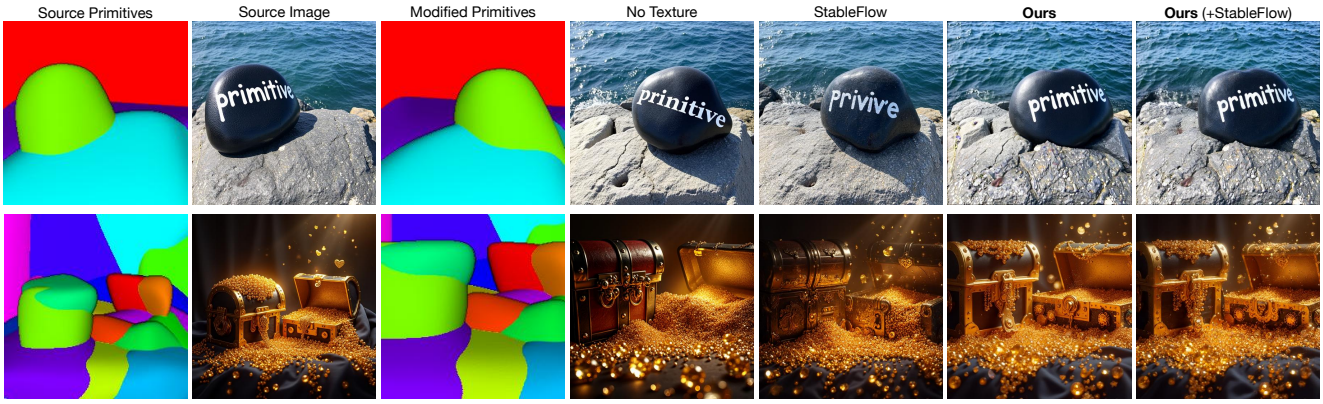
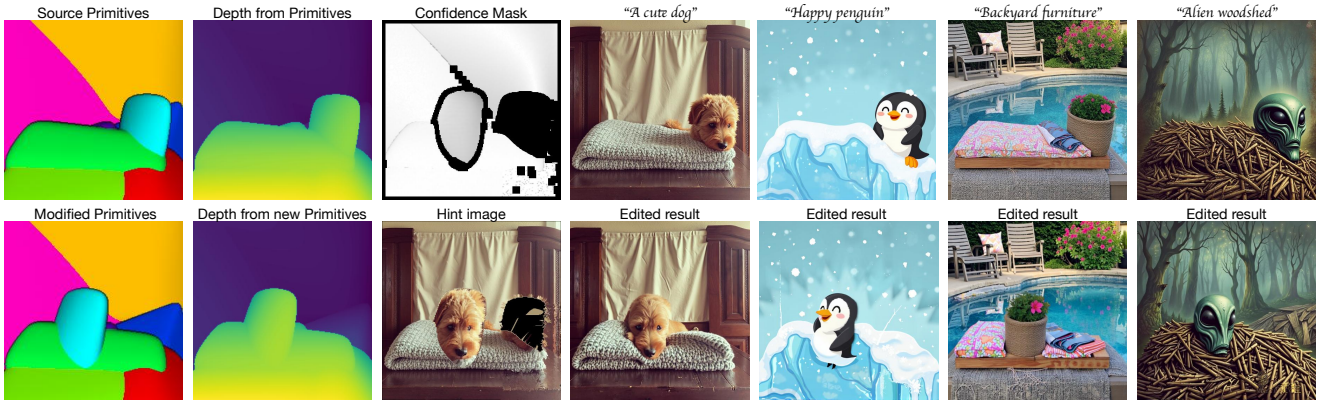


Figure 6. **Projection-Based Texture Hints Preserve Object Identity After Edits.** This figure compares our projection-based texture hints against StableFlow [1], which uses vital-layer key-value injection. **First two columns:** input primitives and image. **Third:** edited primitives. **Fourth:** synthesis from original depth, revealing consistent geometry but altered texture. **Fifth:** StableFlow's approach often changes texture or object identity. **Sixth:** our projection-based hints maintain texture fidelity despite edits. **Seventh:** combining both approaches can improve fine detail recovery (e.g., the treasure chest).

Number of Parts (K)	**AbsRel Error**↓
4	0.0376
6	0.0330
8	0.0295
10	0.0282
12	0.0265
24	0.0223
36	0.0203
48	0.0202
60	0.0194
72	0.0195

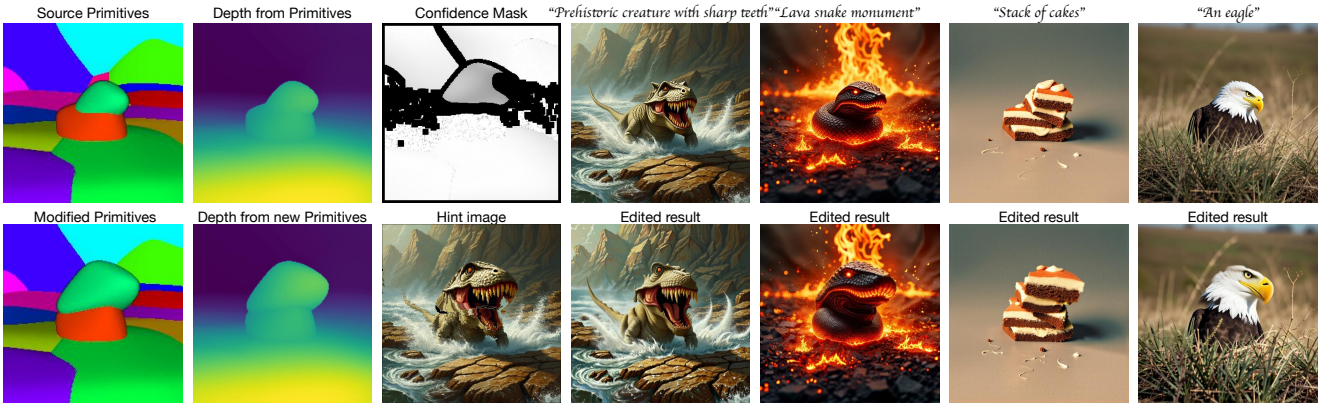
Table 2. AbsRel depth error metrics for varying numbers of 3D box primitives (12-face polytopes). Lower values indicate better depth map approximation quality. While theory would predict AbsRel $\to 0$ as $K \to \infty$ (e.g. one primitive per pixel), in practice we run into bias-variance problems fitting more than 60 primitives. Generating primitives is efficient (approx. 1-3 seconds per image on the GPU including finetuning and rendering) so it is feasible for the user to select from a few candidates based on the desired level of abstraction. No other primitive-conditioned image synthesis method offers variable abstraction.

(a) $K = 6$ parts.

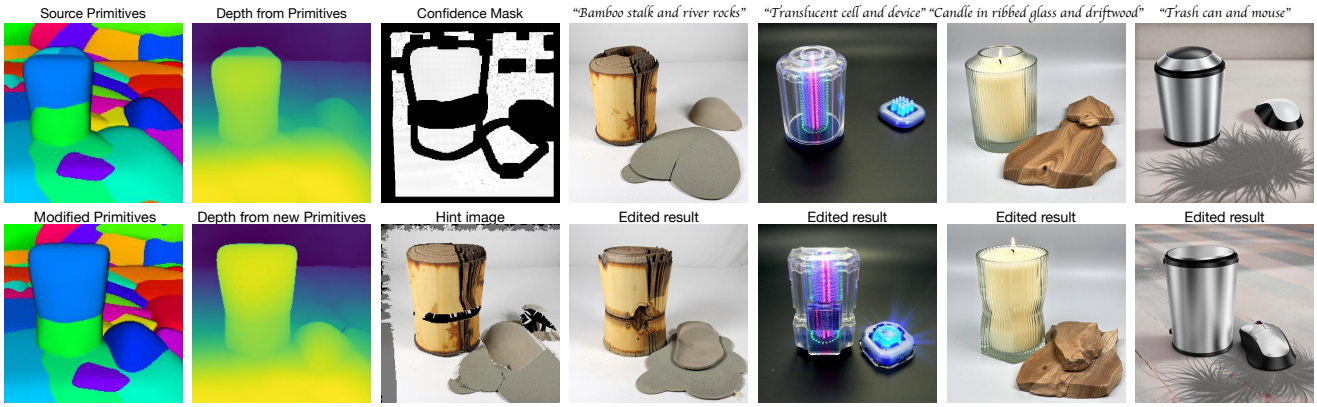


(b) $K = 8$ parts.

Figure 7. **Applying the same primitive edit for different text prompts at coarse scale ($K \in \{6, 8\}$ parts).** The first row in each subplot contains source primitives and depth (first two columns); the confidence mask for hint generation, followed by four source RGB images. The second row shows the modified primitives and depth, followed by the hint image x_{hint}, followed by the four corresponding edited images. At coarse scales, moving a primitive can move a lot of texture at once. Observe how our hint generation procedure automatically yields confidence masks and hints, assigning low confidence to boundaries of primitives that moved (e.g., the dog's hair) and reveals holes when moving objects. The image model cleans up the low-confidence regions and even handles blurry/aliased texture in the hint when $t_{end} > 0$, meaning that the hint is not used for some denoising steps.



(a) $K = 24$ parts.



(b) $K = 60$ parts.

Figure 8. **Applying the same primitive edit for different text prompts at fine scale ($K \in \{24, 60\}$ parts).** Observe in the first two rows how all synthesized images respect the enlarged green primitive, while background texture is preserved. In the bottom two rows, we compose **several edits** using a large number of primitives ($K = 60$), enabling fine-scaled edits. We scale up the light blue primitive while scaling down the light green primitive on the left-hand side. We then translate the dark blue primitive on the right-hand side towards the bottom center of the image. We also slightly translate the camera upward. Observe how in the subsequent columns, the edited result respects the geometry specified by the primitives while following the high-level texture of the source image. However, notice how composing four edits challenges our procedure, as the texture preservation isn't as tight. For example, in the final column, a tiled pattern appears on the floor that wasn't in the source.

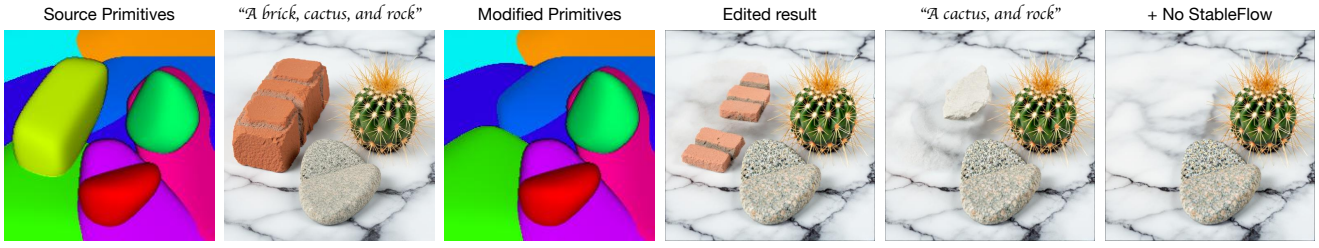


Figure 9. **Primitive edits can conflict with the text prompt.** Some geometric edits require changing the text prompt, for example, when removing an object. The **fourth column** mentions brick in the text prompt, but that primitive was removed, resulting in brick pieces in the inpainted region. In the **fifth column**, we remove the brick from the text prompt, which removes the brick pieces but it still leaves behind a white stone. In the **final column**, we use our texture hints but without StableFlow, getting a clean surface. The StableFlow key-value sharing approach placed brick and stone textures where we didn't want them. We conclude that our texture hints are critical, but combining them with StableFlow [1] key-value sharing can help in some cases, hurt in others.

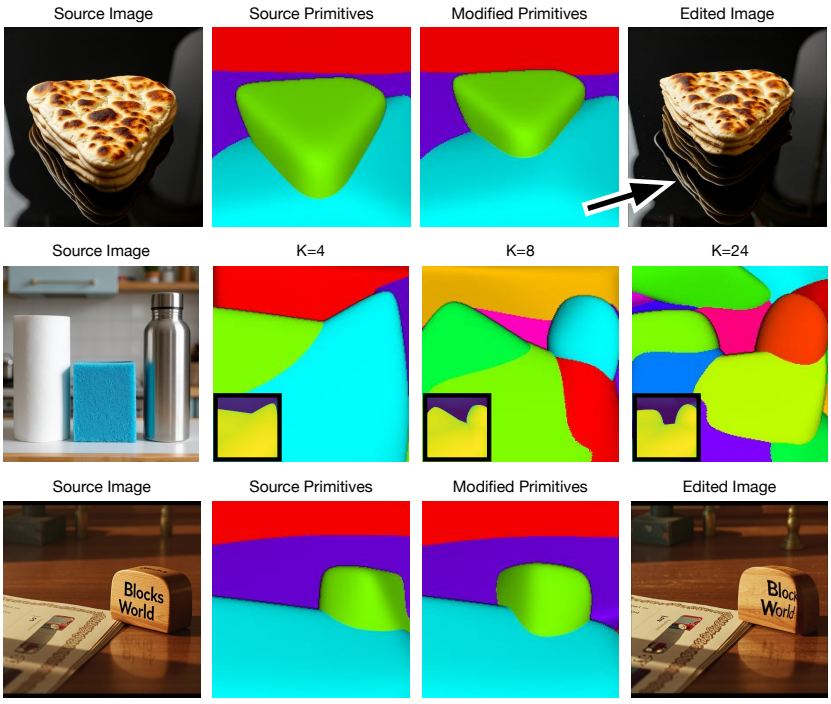


Figure 10. **Failure cases. Top: Illumination misalignments.** Our texture hints operate in pixel space and cannot model illumination effects outside primitive boundaries (e.g., reflections or cast shadows). As a result, moving or scaling objects may not consistently update their associated lighting effects. For example, the bread stack is translated correctly, but its reflection remains unchanged (see fourth column). **Middle: Poor decomposition.** Primitive fitting may fail in cluttered scenes or near image boundaries, where sparse depth points hinder the separation of adjacent objects (e.g., the bottle and paper towel are merged). This leads to inaccurate depth maps and poor control. **Bottom: Rotation artifacts.** Large object rotations (e.g., 50°) disrupt texture consistency and geometry, possibly due to distribution shift in the texture hints, resulting in distortions or hallucinated content (e.g., warped "Blocks World" text).

Figure 11. Our model is compatible with most depth-image synthesizers. While a pretrained FLUX works out of the box, LoRA weights on top of the base FLUX model are available (`FLUX.1 Depth [dev] LoRA`), exposing a new $lora_{weight}$ parameter (scaling the activations of the LoRA layers). This is intriguing in the context of our primitives, because they can either be used to coarsely model scene geometry (e.g. $lora_{weight}$ near 0.8, **second last column**), leaving details to the image synthesizer, or they can tightly control the result when $lora_{weight}$ is close to 1 (**final column**).

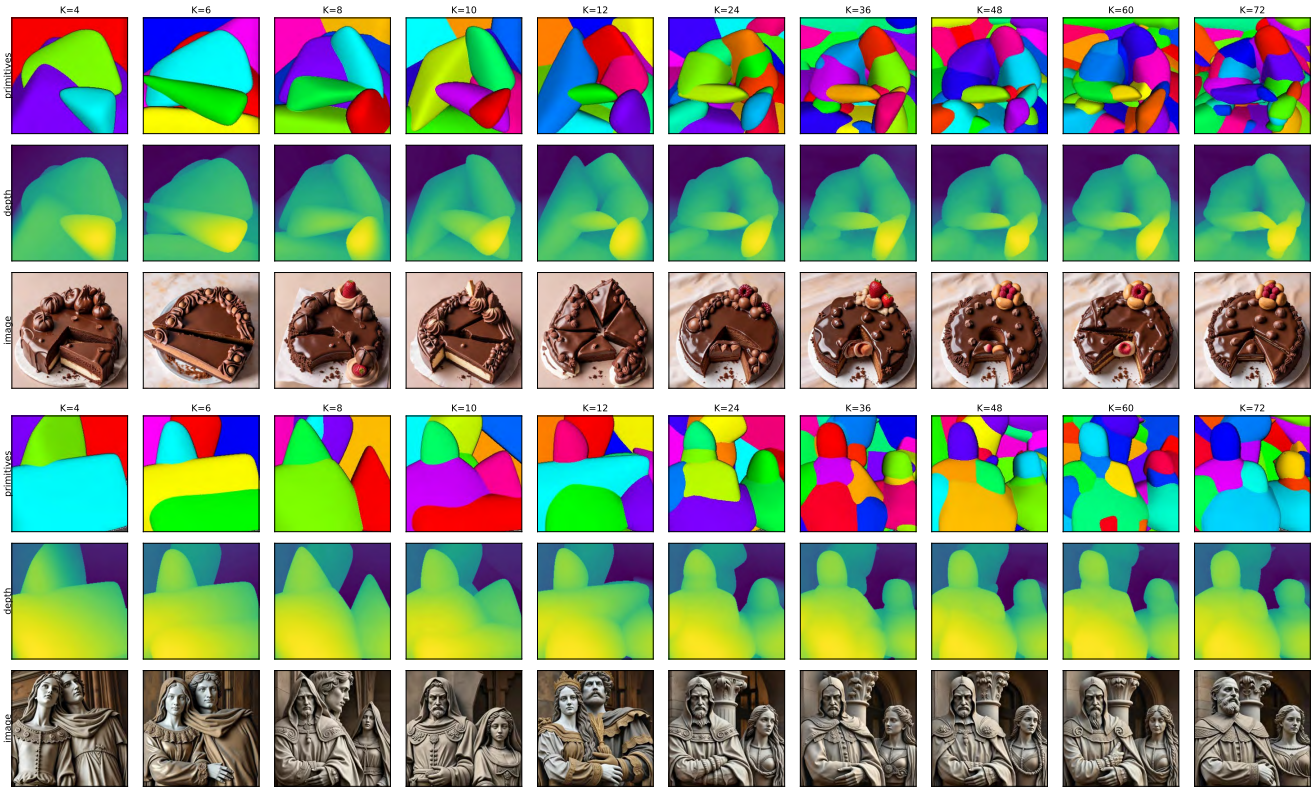


Figure 12. Given the same depth map, we extract primitives at variable resolution (from 4-72 parts). We show the depth maps in each second row, and synthesized result in each 3rd row. Observe how no matter the resolution, the FLUX-LoRA model (we use $lora_{weight} = 0.8$) gives an image that follows the primitive conditioning. We conclude that a wide array of primitive densities is tolerable to depth-to-image models, enabling meaningful artistic edits.

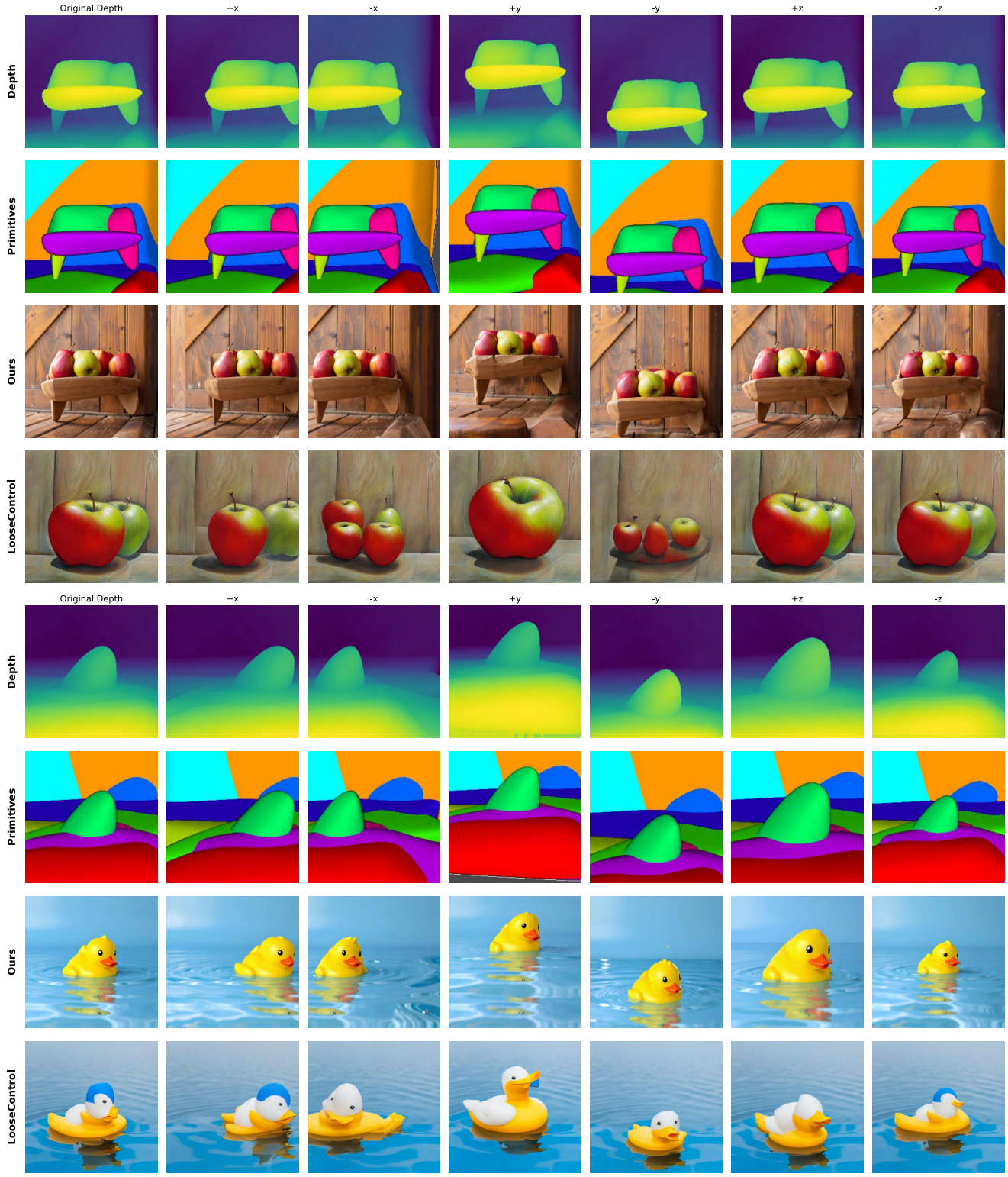


Figure 13. Additional move camera evaluations. Generative Blocks World can simultaneously adhere to source texture and requested primitives.

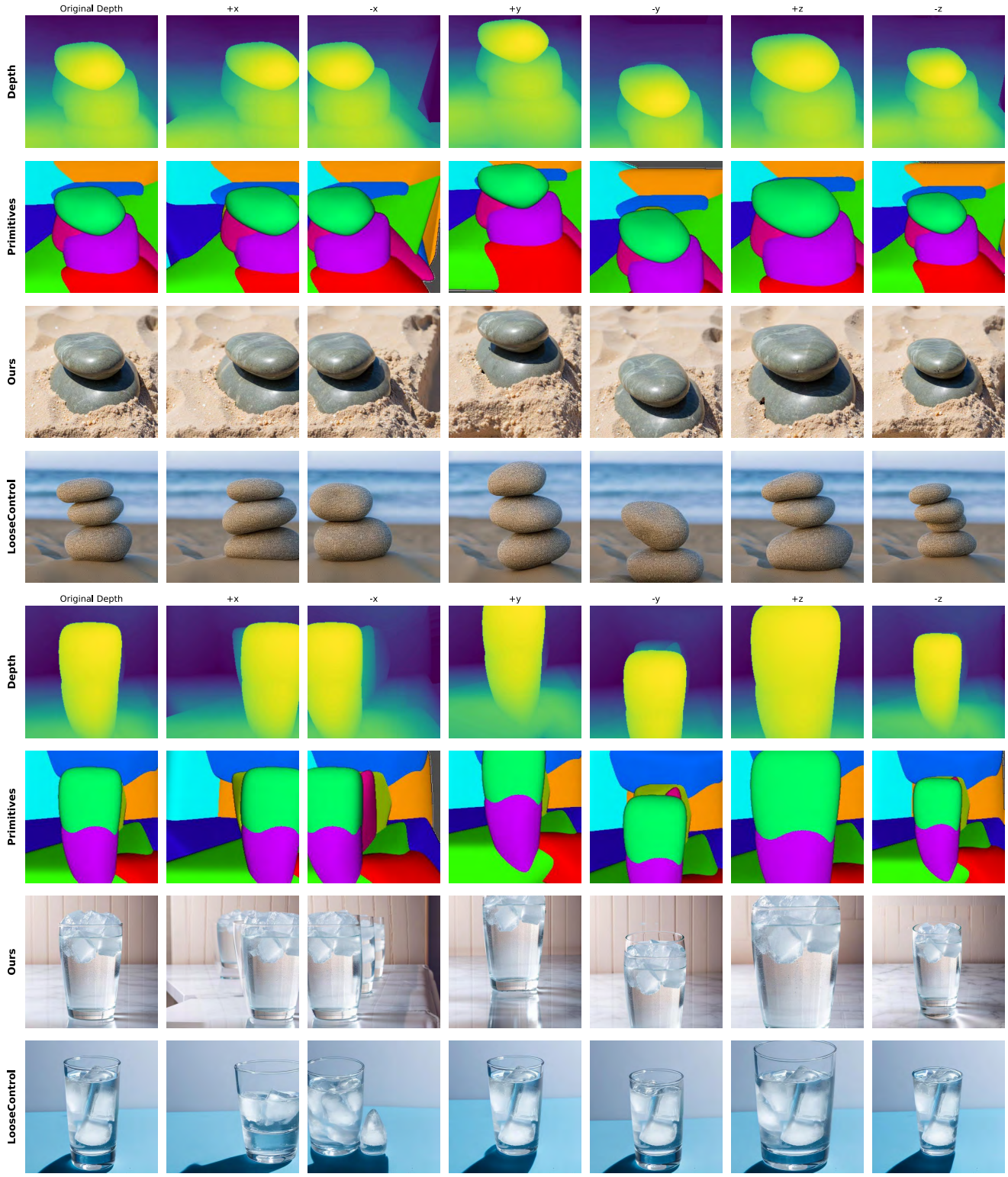


Figure 14. Additional move camera evaluations. Our method can simultaneously adhere to source texture and requested primitives.

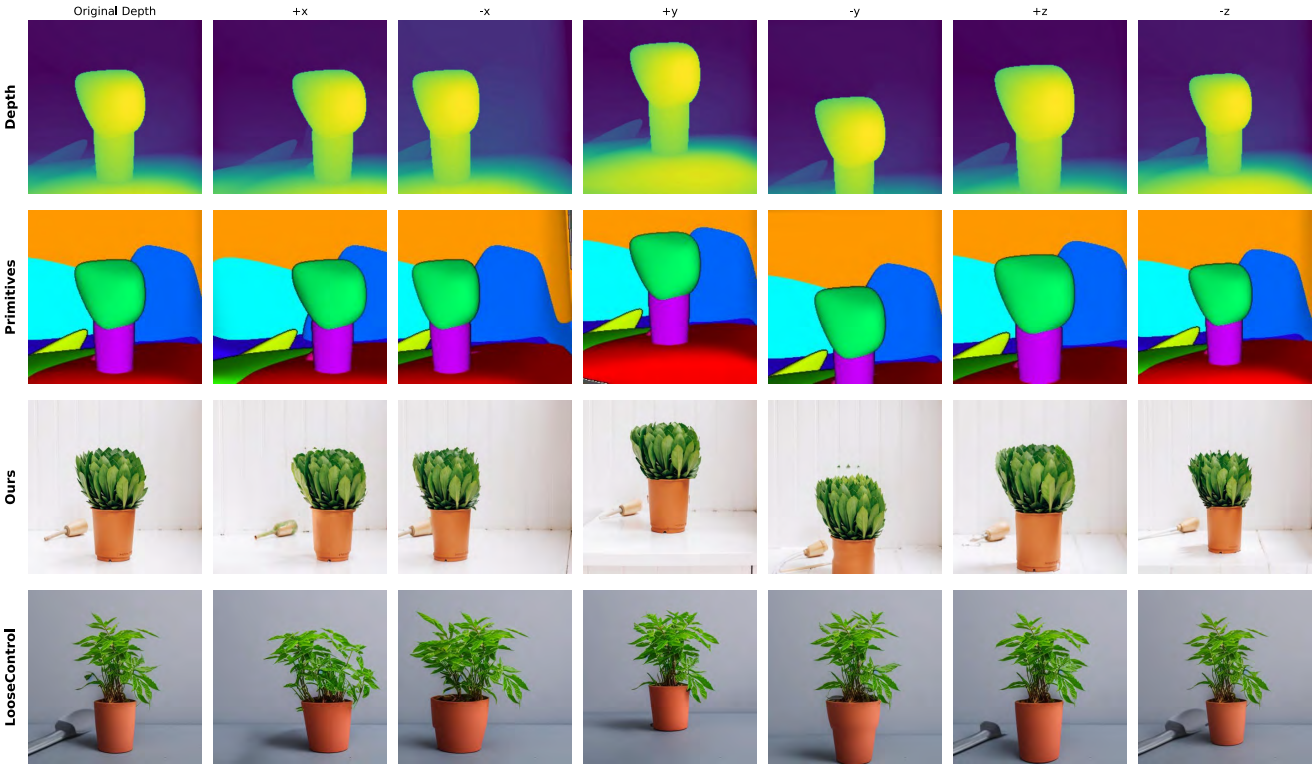


Figure 15. Additional move camera evaluations. Our method can simultaneously adhere to source texture and requested primitives.

Figure 16. We repeat the analysis of StableFlow [1], which applies U-Net based key-value caching of older-generation Diffusion models to newer Diffusion Transformers. Specifically, their work analyzes `FLUX.1 [dev]`; given that our work uses depth maps to communicate geometric information to our image generation model, we analyze Vital Layers in `FLUX.1 Depth [dev]` and `FLUX.1 Depth [dev] LoRA`, finding the top 5 multimodal and single modal layers to be essentially identical. We try using the vital layers we identified for texture transfer, finding this method to be inadequate (see Fig. 6).

ALGORITHM 1: Point Cloud Correspondence Generation

Input: $\mathcal{P}_1, \mathcal{P}_2$: point clouds; $\mathcal{M}_1, \mathcal{M}_2$: convex maps; $\mathcal{T}$: primitive transforms; $\mathcal{C}$: centers; $d_{\max} = 0.005$: max distance threshold

Output: $\mathcal{R}$: correspondence map; $\mathcal{W}$: confidence map

Function `ApplyTransform(`$\mathbf{p}$, $\mathbf{c}$, $\mathbf{T}$`)`**:**
- $\mathbf{p}' \leftarrow \mathbf{p} - \mathbf{c}$; `// Center the point`
- **if** $\mathbf{T}$ *contains translation* **then**
  - $\mathbf{p}' \leftarrow \mathbf{p}' - \mathbf{T}_{\text{trans}}$;
- **end**
- **if** $\mathbf{T}$ *contains rotation angle* θ **then**
  - $c, s \leftarrow \cos(-\theta), \sin(-\theta)$;
  - $x', z' \leftarrow x' \cdot c - z' \cdot s, x' \cdot s + z' \cdot c$; `// Y-axis rotation`
- **end**
- **if** $\mathbf{T}$ *contains scaling factor scale* **then**
  - $\mathbf{p}' \leftarrow \mathbf{p}'/scale$;
- **end**
- **return** $\mathbf{p}' + \mathbf{c}$;

$\mathcal{R} \leftarrow \mathbf{0}_{H \times W \times 2}$; `// Initialize correspondence map`

$\mathcal{W} \leftarrow \mathbf{0}_{H \times W}$; `// Initialize confidence map`

for $p \in$ *unique*$(\mathcal{M}_1)$ **do**
- **if** $p < 0$ ***or*** $p \geq |\mathcal{C}|$ ***or*** $p \notin \mathcal{M}_1$ ***or*** $p \notin \mathcal{M}_2$ **then**
  - **continue**;
- **end**
- $\mathcal{I}_1 \leftarrow \{(y, x) : \mathcal{M}_1[y, x] = p\}$; `// Pixel indices for primitive` p `in map 1`
- $\mathcal{I}_2 \leftarrow \{(y, x) : \mathcal{M}_2[y, x] = p\}$; `// Pixel indices for primitive` p `in map 2`
- $\mathcal{Q}_1 \leftarrow \{\mathcal{P}_1[y, x] : (y, x) \in \mathcal{I}_1\}$; `// 3D points for primitive` p
- **for** $(y_2, x_2) \in \mathcal{I}_2$ **do**
  - $\mathbf{q} \leftarrow \mathcal{P}_2[y_2, x_2]$; `// Query point from second cloud`
  - **if** $p \in \mathcal{T}$ **then**
    - $\mathbf{q} \leftarrow$ `ApplyTransform(`$\mathbf{q}, \mathcal{C}[p], \mathcal{T}[p]$`)` ; `// Apply transformation`
  - **end**
  - $\mathbf{d} \leftarrow \|\mathcal{Q}_1 - \mathbf{q}\|_2$; `// Compute distances to all points`
  - $i^* \leftarrow \arg\min_i \mathbf{d}[i]$; `// Find nearest neighbor`
  - $d_{\min} \leftarrow \mathbf{d}[i^*]$;
  - **if** $d_{\min} \leq d_{\max}$ **then**
    - $(y_1^*, x_1^*) \leftarrow \mathcal{I}_1[i^*]$; `// Get corresponding pixel coordinates`
    - $\mathcal{R}[y_2, x_2] \leftarrow [x_1^*, y_1^*]$;
    - $\mathcal{W}[y_2, x_2] \leftarrow 1 - \min(d_{\min}/d_{\max}, 1)$; `// Confidence score`
  - **end**
- **end**

end

return $\mathcal{R}, \mathcal{W}$;

ALGORITHM 2: Hint Generation from Correspondence Maps

Input: $\mathbf{I}_{\text{src}} \in \mathbb{R}^{C\times H_s\times W_s}$: source image; $\mathcal{R} \in \mathbb{R}^{H_r\times W_r\times 2}$: correspondence map; $\mathcal{W} \in \mathbb{R}^{H_r\times W_r}$: confidence map; $\mathcal{M}_{\text{hit}} \in \{0,1\}^{H_r\times W_r}$: hit mask

Output: $\mathcal{H} \in \mathbb{R}^{C\times H_s\times W_s}$: generated hint image

Function `BilinearSample(`$\mathbf{I}$, y, x`)`**:**

$C, H, W \leftarrow \text{shape}(\mathbf{I})$;

$x \leftarrow \text{clip}(x, 0, W - 1.001)$, $y \leftarrow \text{clip}(y, 0, H - 1.001)$;

$x_0, y_0 \leftarrow \lfloor x \rfloor, \lfloor y \rfloor$; `// Floor coordinates`

$x_1, y_1 \leftarrow \min(x_0 + 1, W - 1), \min(y_0 + 1, H - 1)$;

$w_x, w_y \leftarrow x - x_0, y - y_0$; `// Interpolation weights`

$\mathbf{v}_{\text{top}} \leftarrow \mathbf{I}[:, y_0, x_0] \cdot (1 - w_x) + \mathbf{I}[:, y_0, x_1] \cdot w_x$;

$\mathbf{v}_{\text{bot}} \leftarrow \mathbf{I}[:, y_1, x_0] \cdot (1 - w_x) + \mathbf{I}[:, y_1, x_1] \cdot w_x$;

return $\mathbf{v}_{\text{top}} \cdot (1 - w_y) + \mathbf{v}_{\text{bot}} \cdot w_y$;

$\lambda_h \leftarrow H_s/H_r, \lambda_w \leftarrow W_s/W_r$; `// Scale factors`

$\mathcal{H} \leftarrow \mathbf{0}_{C\times H_s\times W_s}$; `// Initialize hint image`

for $y \in [0, H_r)$ **do**

  for $x \in [0, W_r)$ **do**

    if $\mathcal{M}_{hit}[y, x] = 1$ **then**

      continue ; `// Skip hit pixels`

    end

    $(x_c, y_c) \leftarrow \mathcal{R}[y, x]$; `// Get correspondence`

    $w \leftarrow \mathcal{W}[y, x]$; `// Get confidence`

    if $w < 0.1$ **then**

      continue ; `// Skip low-confidence correspondences`

    end

    $y_{\text{src}} \leftarrow y_c \cdot \lambda_h, x_{\text{src}} \leftarrow x_c \cdot \lambda_w$; `// Scale to source resolution`

    $y_{\text{start}} \leftarrow \lfloor y \cdot \lambda_h \rfloor, y_{\text{end}} \leftarrow \lfloor (y + 1) \cdot \lambda_h \rfloor$;

    $x_{\text{start}} \leftarrow \lfloor x \cdot \lambda_w \rfloor, x_{\text{end}} \leftarrow \lfloor (x + 1) \cdot \lambda_w \rfloor$;

    for $y_s \in [y_{start}, y_{end})$ **do**

      for $x_s \in [x_{start}, x_{end})$ **do**

        if $y_s \notin [0, H_s)$ ***or*** $x_s \notin [0, W_s)$ **then**

          continue ; `// Boundary check`

        end

        $\alpha_y \leftarrow \frac{y_s - y_{\text{start}}}{\max(y_{\text{end}} - y_{\text{start}}, 1)}$; `// Normalized offset`

        $\alpha_x \leftarrow \frac{x_s - x_{\text{start}}}{\max(x_{\text{end}} - x_{\text{start}}, 1)}$;

        $y_{\text{sample}} \leftarrow y_{\text{src}} + \alpha_y \cdot \lambda_h$;

        $x_{\text{sample}} \leftarrow x_{\text{src}} + \alpha_x \cdot \lambda_w$;

        $\mathcal{H}[:, y_s, x_s] \leftarrow$ `BilinearSample(`$\mathbf{I}_{\text{src}}, y_{\text{sample}}, x_{\text{sample}}$`)`;

      end

    end

  end

end

return $\mathcal{H}$;