[CVPR 2024]

WonderJourney: Going from Anywhere to Everywhere

Hong-Xing Yu[1] Haoyi Duan[1] Junhwa Hur[2] Kyle Sargent[1] Michael Rubinstein[2]
William T. Freeman[2] Forrester Cole[2] Deqing Sun[2] Noah Snavely[2] Jiajun Wu[1] Charles Herrmann[2]
[1]Stanford University [2]Google Research

Presenter: Gyeongsu Cho
Thu Dec 5, 2024

Contents

- Introduction
- Method
- Experiments
- Conclusion

Introduction

WonderJourney

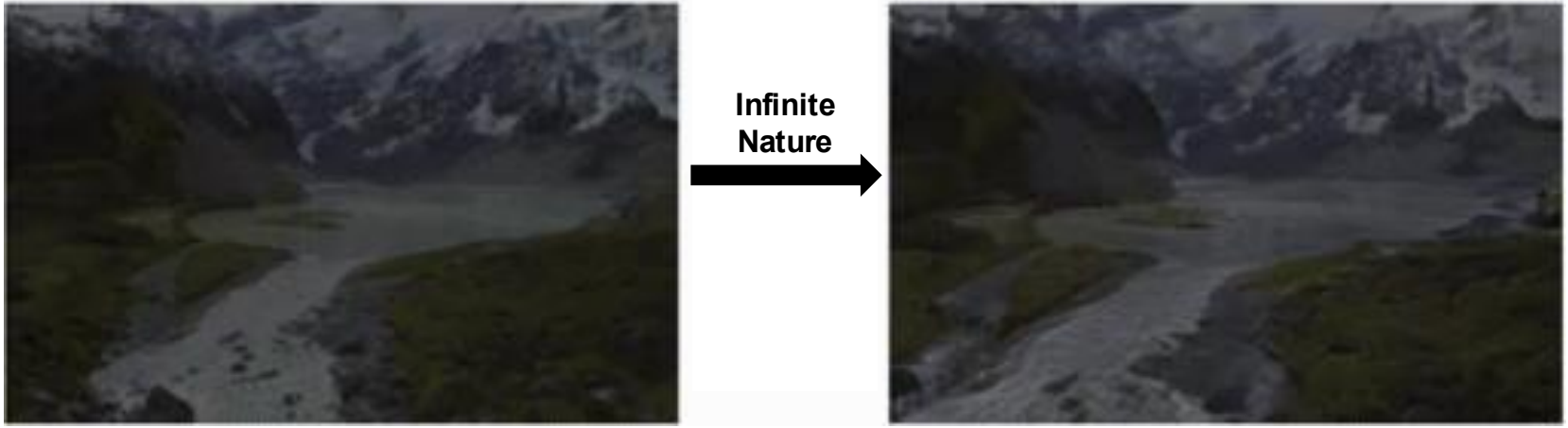
Input: a single image
OR
text (“Girl in wonderland”)

Wonder
Journey

Output: generated video

WonderJourney generates a sequence of diverse yet coherently connected 3D scenes along a camera trajectory.

Main Challenge: Generating diverse, plausible scene elements



Input: a single image **Output: generated video**

Prior works on perpetual view generation only focuses on a single type of scenes.

Liu, Andrew, et al. "Infinite nature: Perpetual view generation of natural scenes from a single image.", ICCV 2021

Main Challenge: Generating diverse, plausible scene elements

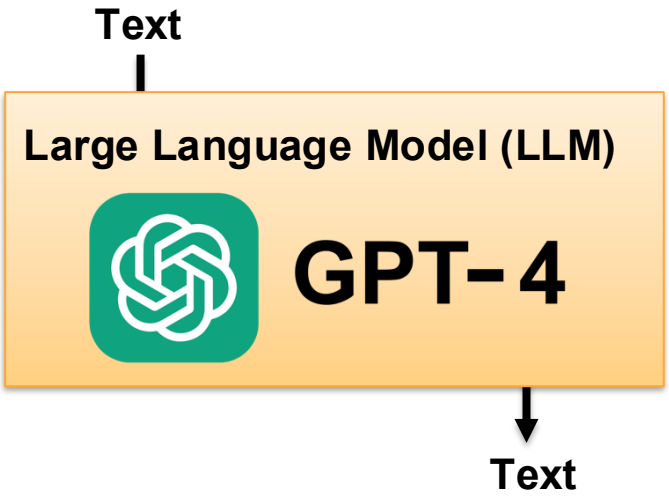
Input: a fairy-tale image

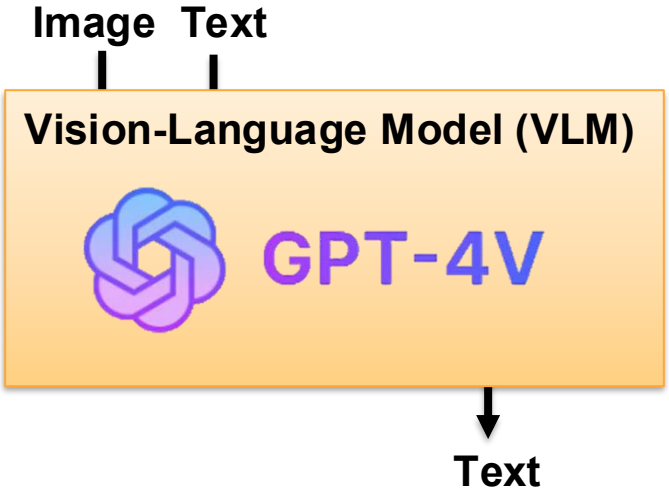
Input: a real image

Prior works on perpetual view generation only focuses on a single type of scenes.

- We start from any user-provided location, and end at any plausible locations.
- Generating diverse and plausible objects, backgrounds, and layouts, that fit into observed scenes and transit to next scene.

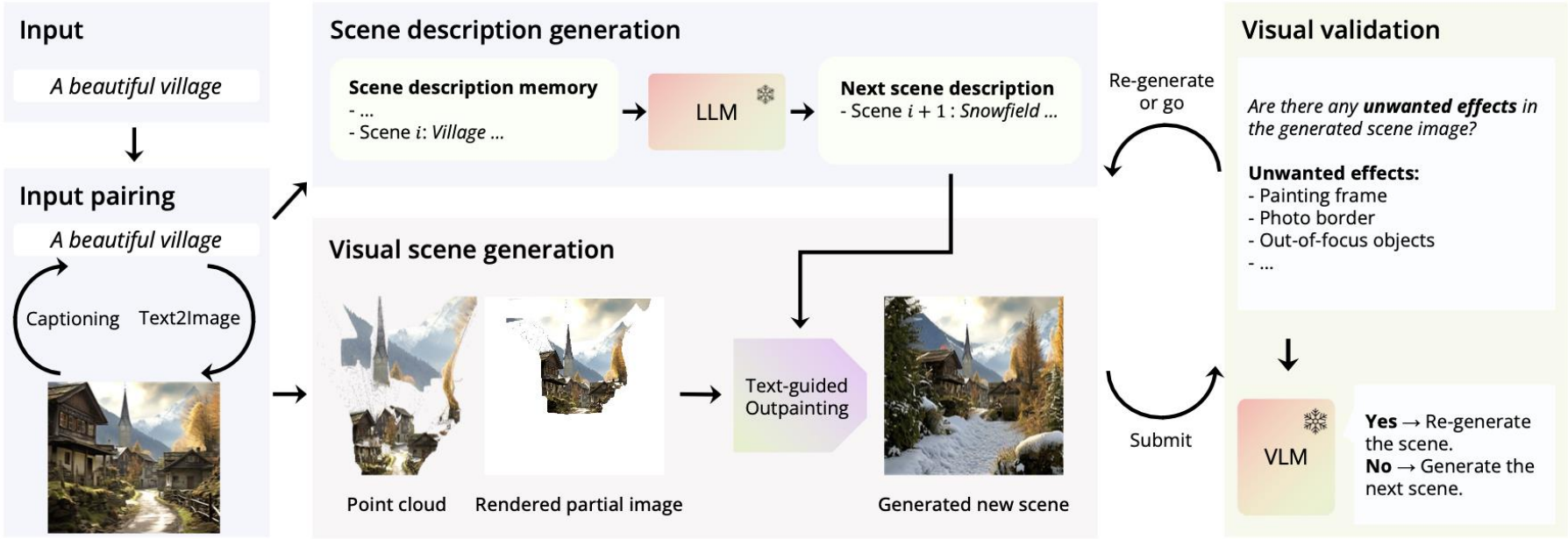
Key Idea Leveraging LLM and VLM to provide semantic and world knowledge.



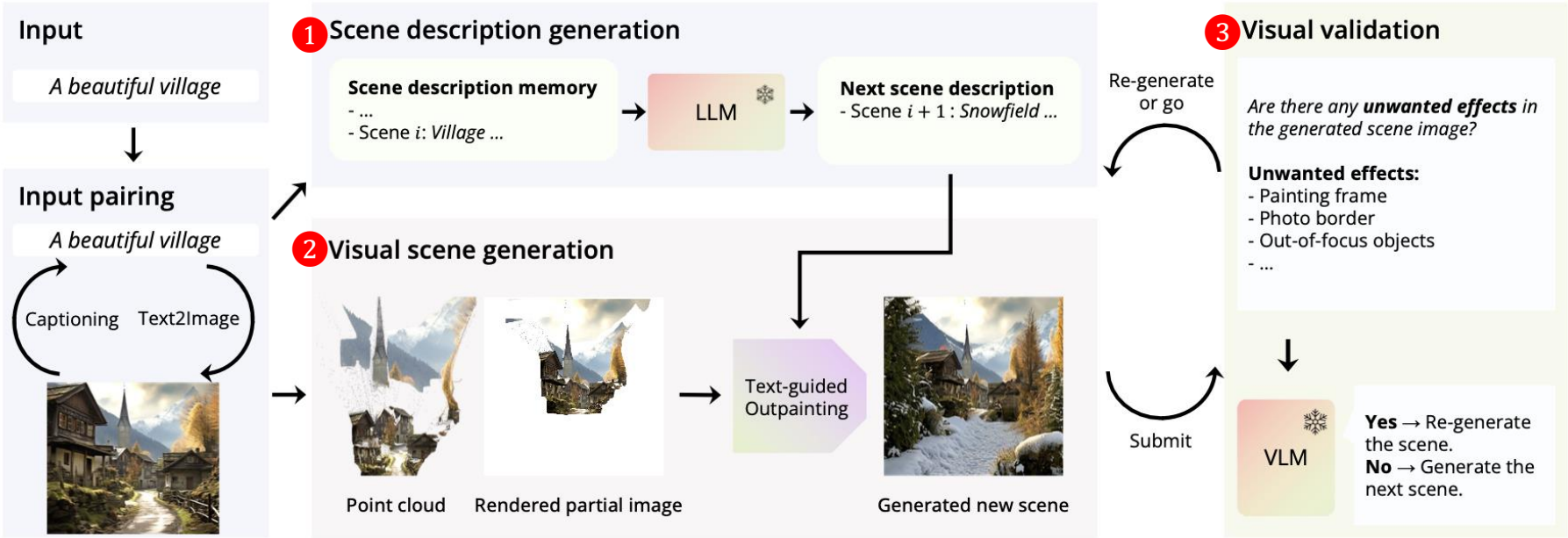


Method

WonderJourney: A modular framework for 3D scene generation.

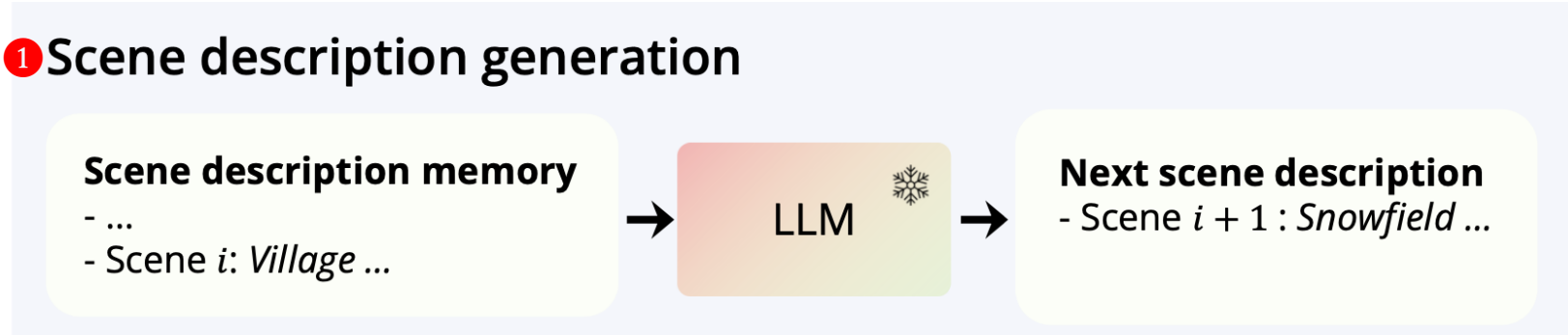


Overview



1. **Scene description generation**: A LLM to generate a long sequence of scene descriptions.
2. **Visual scene generation**: A text-driven pcd generation pipeline to synthesize 3D visual scenes.
3. **Visual validation**: A VLM to verify the generated scenes.

Scene description generation



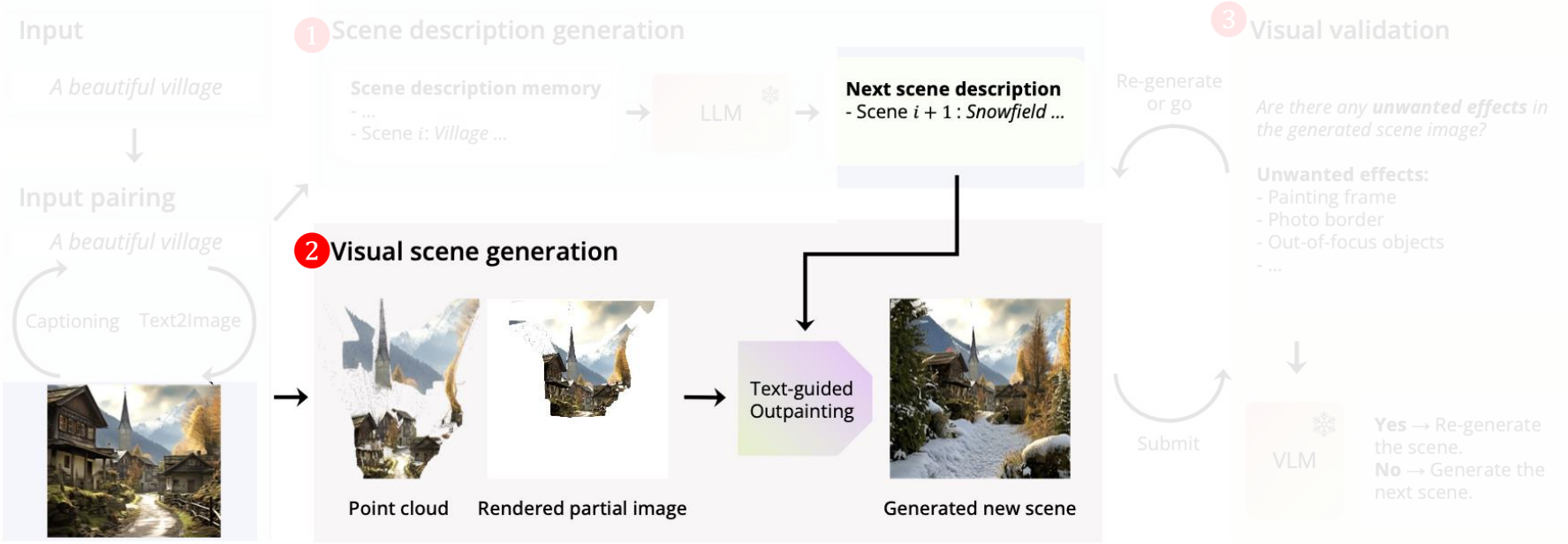
$$\mathcal{S}_{i+1} = g_{\text{description}}(\mathcal{J}, \mathcal{M}_i), \quad (1)$$

$$\mathcal{M}_{i+1} = \mathcal{M}_i \cup \{\mathcal{S}_{i+1}\}. \quad (2)$$

$\mathcal{J}$ = the task specification
$\mathcal{M}_i = \{\mathcal{S}_0, \mathcal{S}_1, \cdots, \mathcal{S}_i\}$
$g_{\text{description}}$ = LLM (GPT4)
$\mathcal{S}_{i+1}$ = $i{+}1^{\text{th}}$ scene description

The scene description generation takes a set of past and current scene descriptions as input and predicts the subsequent scene description

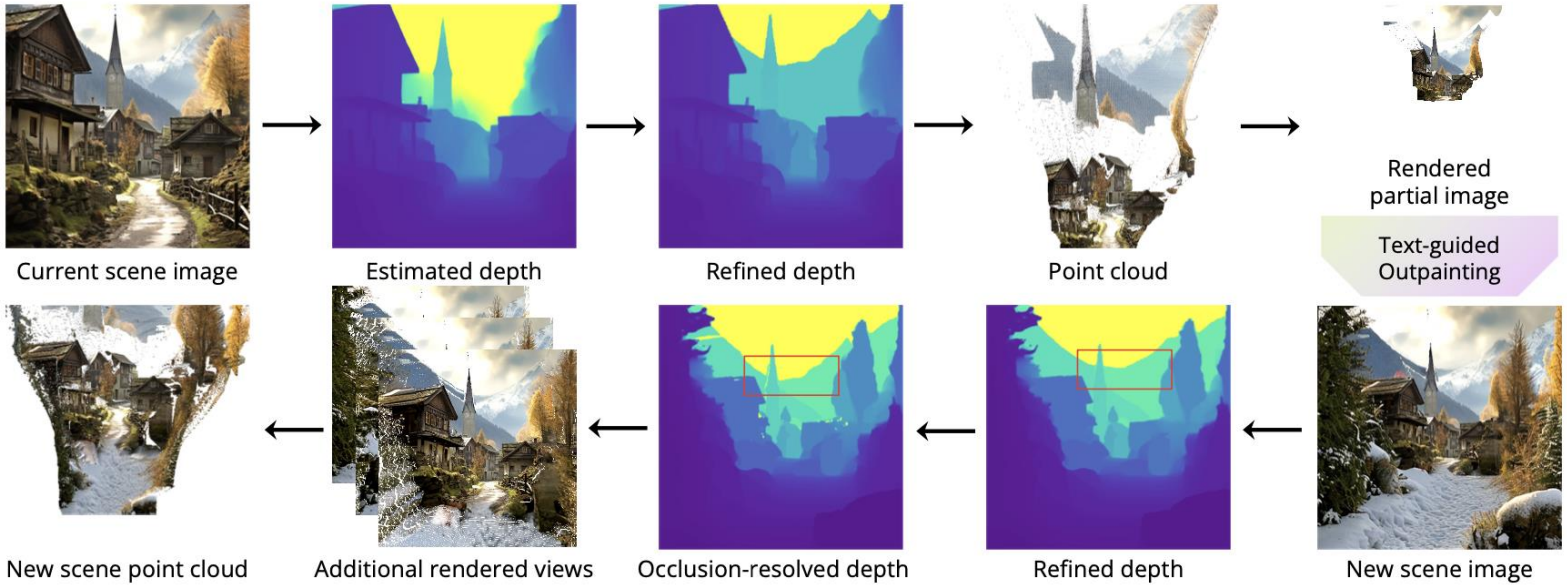
Visual scene generation



The goal of **visual scene generation** is to:

- produce a 3D point cloud representation of the next scene.
- ensure geometric and semantic coherence with previous scenes.
- leverage text and image inputs to guide the visual synthesis.

Visual scene generation

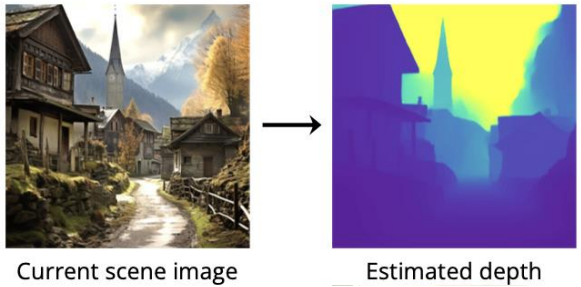


$$\mathcal{P}_{i+1} = g_{\text{visual}}(I_i, \mathcal{S}_{i+1}), \tag{3}$$

The goal of **visual scene generation** is to:

- produce a 3D point cloud representation of the next scene.
- ensure geometric and semantic coherence with previous scenes.
- leverage text and image inputs to guide the visual synthesis.

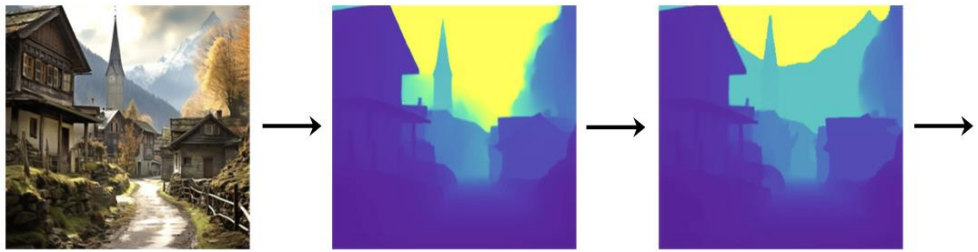
Visual scene generation Lifting image to point cloud



We use **MIDAS v3.1** as depth estimator to lift image to 3D point cloud. But, existing MDE have two issues.

1. Depth discontinuity is not well modeled.
2. The depth of the sky is always underestimated.

Visual scene generation Depth refinement



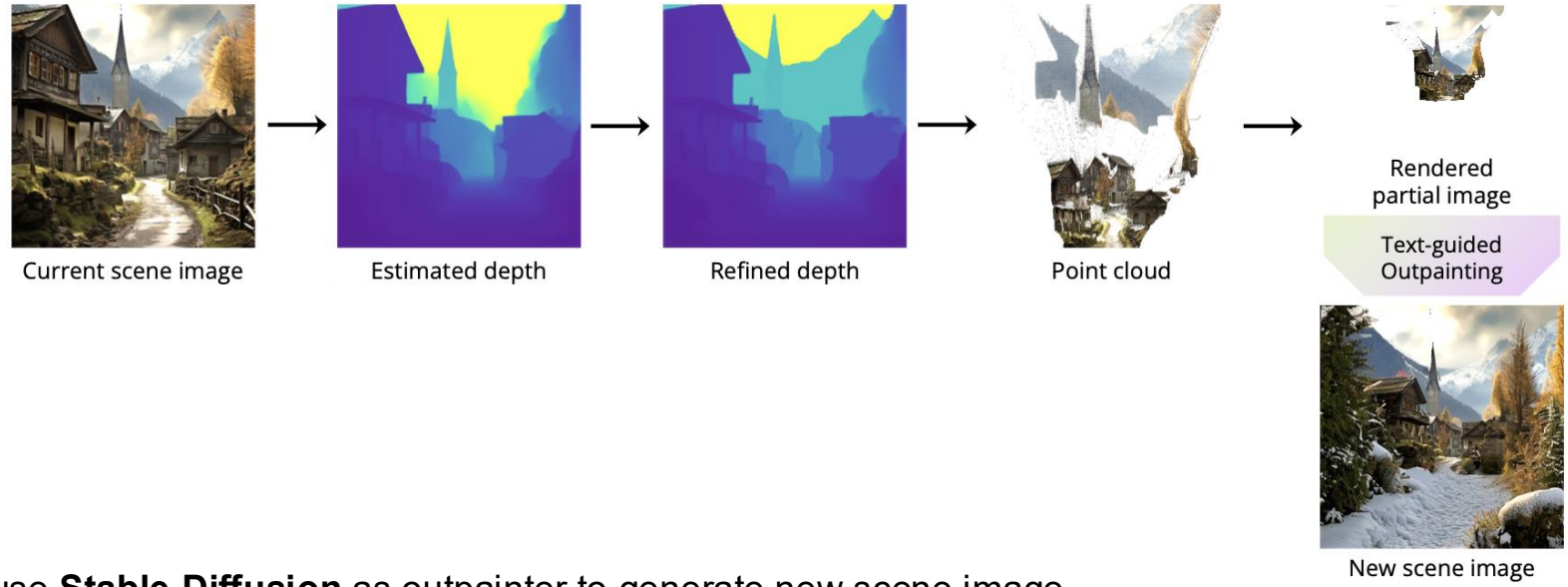
To refine depth at boundaries, we apply the equation utilizing segmentation:

$$\texttt{depth}[\texttt{seg}_j] \leftarrow \begin{cases} \texttt{median}(\texttt{depth}[\texttt{seg}_j]), & \text{if } \Delta D_j < T, \\ \texttt{depth}[\texttt{seg}_j], & \text{otherwise}, \end{cases} \quad (4)$$

To handle the sky depth, we use OneFormer to segment sky region and assign a high value.

Jain, Jitesh, et al. “OneFormer: One Transformer to Rule Universal Image Segmentation.”, CVPR 2023

Visual scene generation Description-guided scene generation



We use **Stable Diffusion** as outpainter to generate new scene image.

$$I_{i+1} = g_{\text{outpaint}}(\hat{I}_{i+1}, \mathcal{S}_{i+1}), \qquad (5)$$

$\hat{I}_{i+1}$ = Rendered partial image

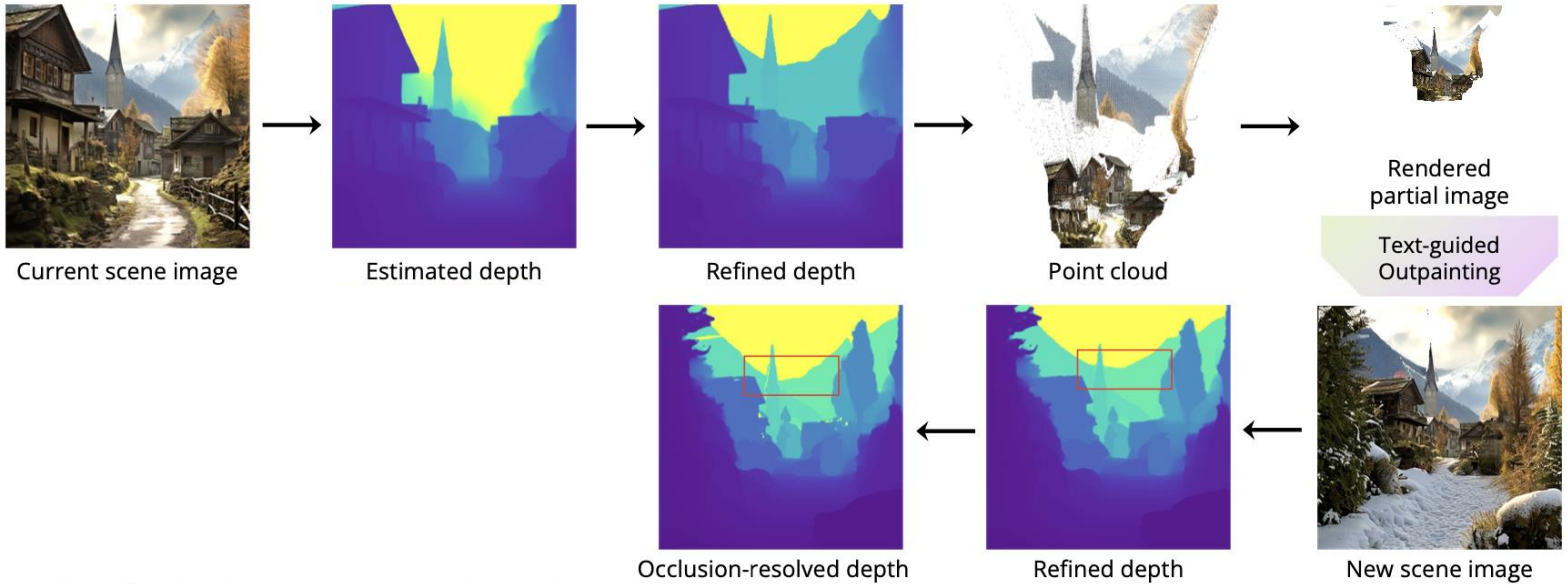
$\mathcal{S}_{i+1}$ = $i{+}1^{\text{th}}$ scene description

g_{outpaint} = Stable Diffusion Inpainting

I_{i+1} = New scene iamge

Visual scene generation

Occlusion handling by re-rendering consistency



To fix the underestimation of depth in disocclusion regions, we use a re-rendering consistency method.

Visual scene generation

New scene registration by depth consistency

The depth estimator is unaware of geometry constraints, we finetune the depth estimator by a depth alignment loss:

$$\mathcal{L}_{\text{depth}} = \max(0, \mathcal{D}^*_{\text{bg}} - \mathcal{D}'_{\text{bg}}) + \|\mathcal{D}^*_{\text{fg}} - \mathcal{D}'_{\text{fg}}\|, \qquad (6)$$

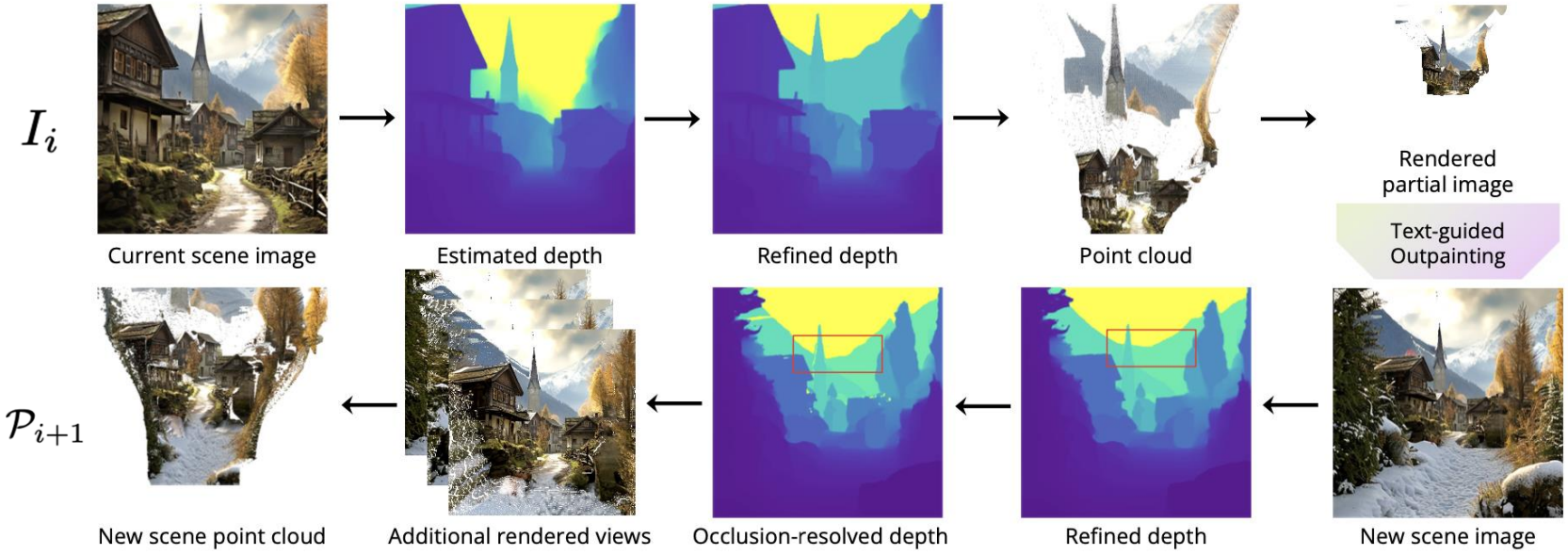
$\mathcal{D}^*_{\text{bg}}$ = the computed depth of back ground pixels from I_i

$\mathcal{D}^*_{\text{fg}}$ = the computed depth of fore ground pixels from I_i

$\mathcal{D}'_{\text{bg}}$ = the estimated depth for pixels corresponding to $\mathcal{D}^*_{\text{bg}}$

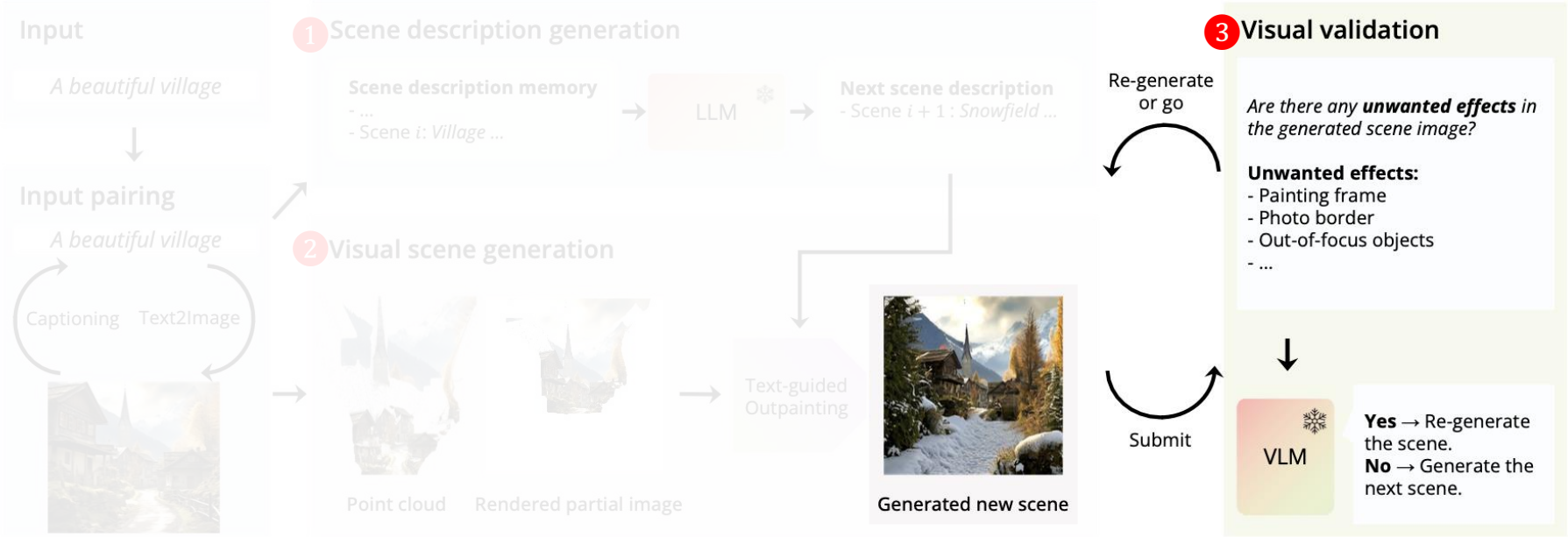
$\mathcal{D}'_{\text{fg}}$ = the estimated depth for pixels corresponding to $\mathcal{D}^*_{\text{fg}}$

Visual scene generation Scene completion



$$\mathcal{P}_{i+1} = g_{\text{visual}}(I_i, \mathcal{S}_{i+1}), \qquad (3)$$

Visual validation



Visual validation: A Vision-Language Model to verify the generated scenes.

Experiments

Experiments

- Dataset
  - Copyright-free photos from online, and generated examples
- Baseline
  - Image-based: InfiniteNature-Zero
  - Text-based: SceneScape
- Evaluation metrics
  - Human preference evaluation

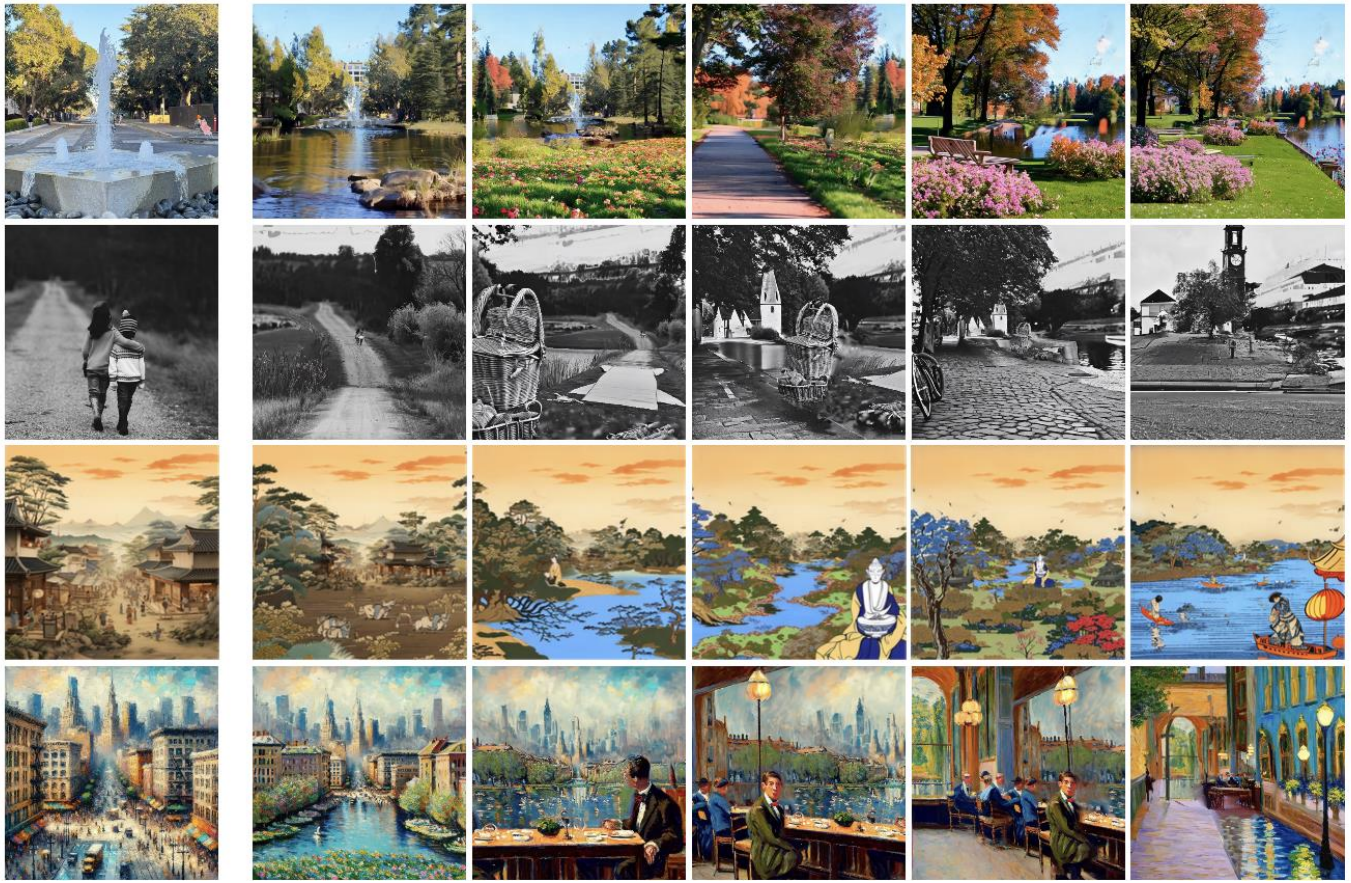
Li, Zhengqi, et al. "Infinitenature-zero: Learning perpetual view generation of natural scenes from single images.", ECCV 2022
Fridman, Rafail, et al. "Scenescape: Text-driven consistent scene generation.", NeurIPS 2023

Experiments

	Div.	Qual.	Compl.	Overall
Ours over InfiniteNature-Zero	92.7%	94.9%	91.5%	88.6%
Ours over SceneScape	88.8%	83.4%	80.0%	90.3%

Li, Zhengqi, et al. "Infinitenature-zero: Learning perpetual view generation of natural scenes from single images." ECCV 2022
Fridman, Rafail, et al. "Scenescape: Text-driven consistent scene generation.", NeurIPS 2023

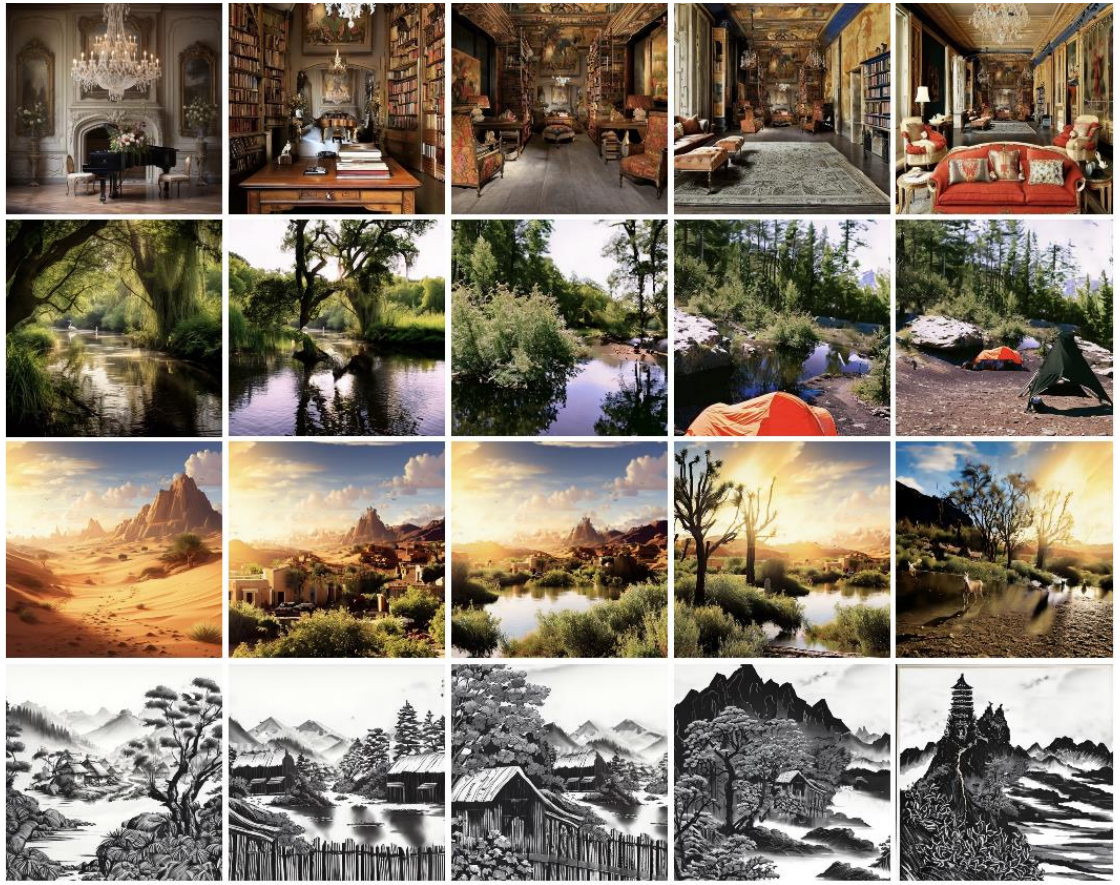
Experiments



Input

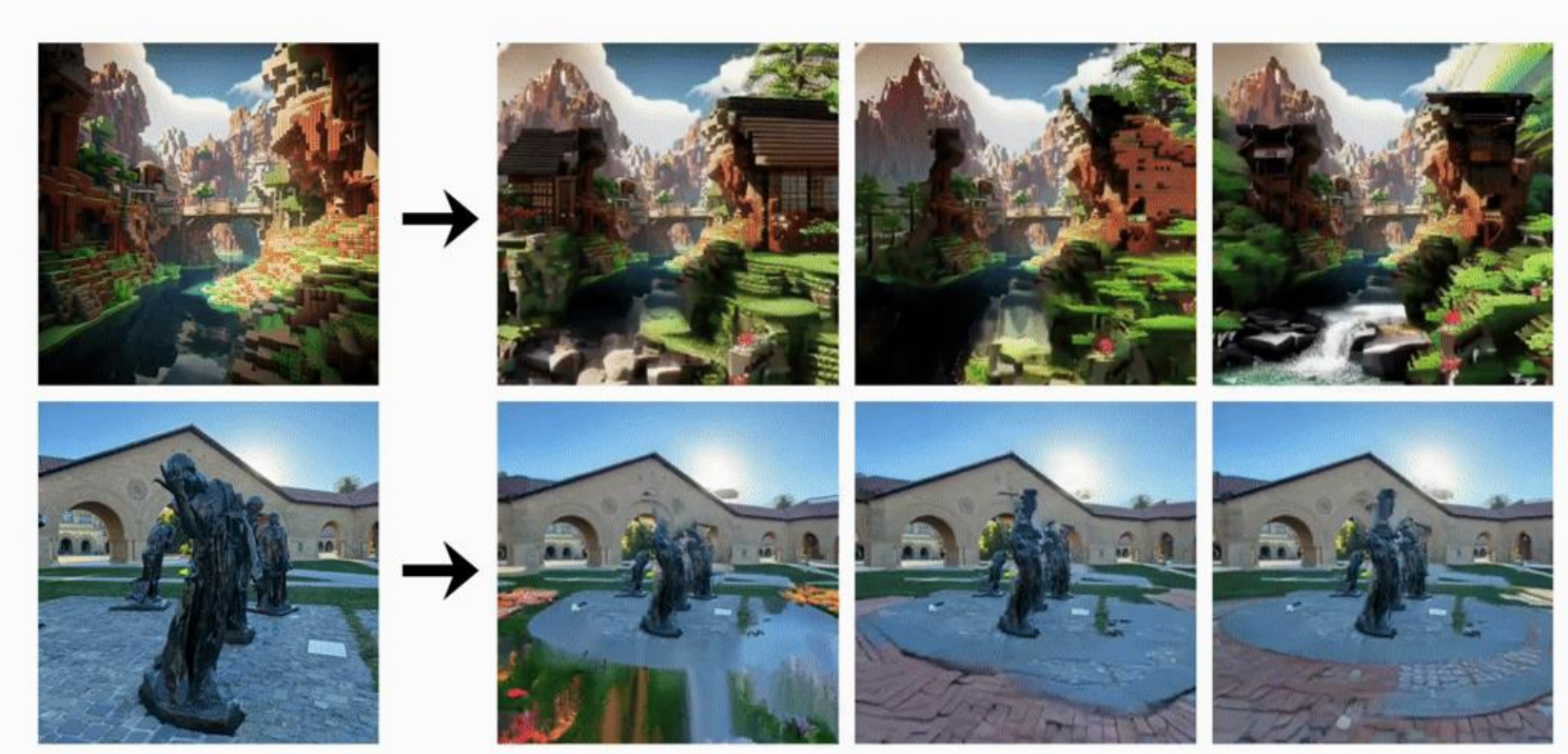
Generated wonderjourneys (rendering of generated sequences of 3D scenes)

Experiments



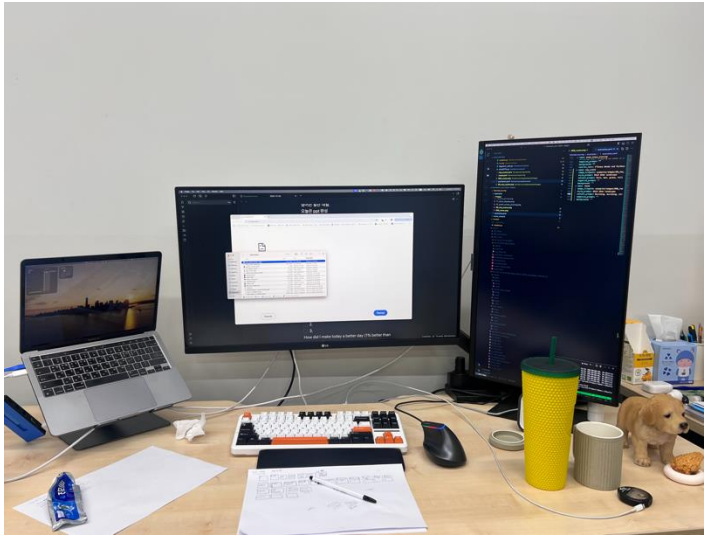
Experiments from anywhere

Experiments to everywhere

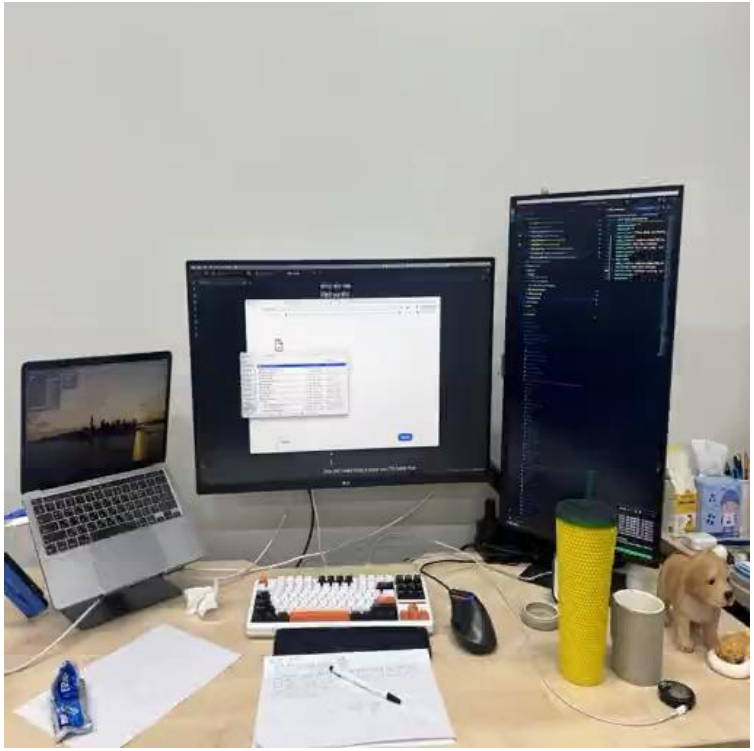


Conclusion

Conclusion

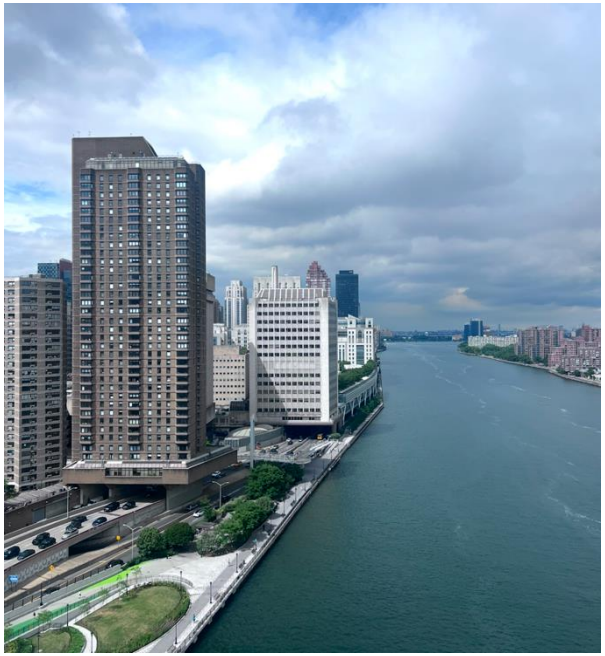


Wonder Journey



I tried **Wonderjourney** in an indoor environment, but **WonderJourney** struggles to generate plausible scene.

Conclusion



Input: a single image

Wonder Journey

Output: generated video

In my experiments, **Wonderjourney** doesn't always work well.
It cost 0.5 $ per long scene generation.

Conclusion

- **Modular framework (System) :**
  - **WonderJourney** uses a modular design with three main components: **description generation, scene generation,** and **visual validation**.
  - Each module works together to ensure scenes are diverse, coherent, and visually appealing.
- **Novel techniques:**
  - Instead of **human guidance**, the **visual validation** method of using **VLM** to regenerate images is innovative.
- **User Experience:**
  - The system generates visually stunning 3D scenes that flow smoothly, creating an immersive experience.
  - It adapts to creative inputs, whether starting from a photo or a text description.

Thank you
Q & A

Appendix – Depth refinement

```python
def refine_disp_with_segments(disparity, segments, keep_threshold=7*0.3):
    """
    Refine disparity values based on provided segmentations.

    Args:
    - disparity (numpy.ndarray): The disparity array of shape [H, W].
    - segments (list): List of segmentation masks represented by dicts.

    Returns:
    - numpy.ndarray: The refined disparity array.
    """

    # Initialize refined_disparity as a copy of disparity
    refined_disparity = disparity.copy()

    # Iterate over each segmentation
    for segment in segments:
        mask = segment['segmentation']  # Extracting the mask

        # 3.a. Query the values from refined_disparity using the mask
        disp_pixels = refined_disparity[mask]

        p70 = np.percentile(disparity[mask], 70)  # 20 for garden to reserve flag
        p30 = np.percentile(disparity[mask], 30)
        disparity_range = p70 - p30

        # Check if disparity range is too significant to be a valid object
        if disparity_range > keep_threshold:
            refined_disparity[mask] = disparity[mask]
        else:
            # 3.b. Find the median value of these disp_pixels
            median_val = np.percentile(disp_pixels, 50)

            # 3.c. Set refined_disparity[mask] to the median value
            refined_disparity[mask] = median_val

    return refined_disparity
```

Appendix – New scene registration by depth consistency

```python
def finetune_depth_model_step(self, target_depth, inpainted_image, mask_align=None, mask_cutoff=None, cutoff_depth=None):
    next_depth, _ = self.get_depth(inpainted_image.detach().cuda())

    # L1 loss for the mask_align region
    loss_align = F.l1_loss(target_depth.detach(), next_depth, reduction="none")
    if mask_align is not None and torch.any(mask_align):
        mask_align = mask_align.detach()
        loss_align = (loss_align * mask_align)[mask_align > 0].mean()
    else:
        loss_align = torch.zeros(1).to(self.device)

    # Hinge loss for the mask_cutoff region
    if mask_cutoff is not None and cutoff_depth is not None and torch.any(mask_cutoff):
        hinge_loss = (cutoff_depth - next_depth).clamp(min=0)
        hinge_loss = F.l1_loss(hinge_loss, torch.zeros_like(hinge_loss), reduction="none")
        mask_cutoff = mask_cutoff.detach()
        hinge_loss = (hinge_loss * mask_cutoff)[mask_cutoff > 0].mean()
    else:
        hinge_loss = torch.zeros(1).to(self.device)

    total_loss = loss_align + hinge_loss
    if torch.isnan(total_loss):
        raise ValueError("Depth FT loss is NaN")
    # print both losses and total loss
    print(f"(1000x) loss_align: {loss_align.item()*1000:.4f}, hinge_loss: {hinge_loss.item()*1000:.4f}, total_loss: {total_

    return total_loss
```

Appendix – Occlusion handling by re-rendering consiste

```python
@torch.no_grad()
def update_additional_point_cloud(self, rendered_depth, image, valid_mask=None, camera=None, points_2d=None, append_depth=False
    """
    args:
        rendered_depth: Depth relative to camera. Note that KF2 camera is represented in KF1 camera-centered coord frame.
        valid_mask: if None, then use inpaint_mask (given by rendered_depth == 0) to extract new points.
            if not None, then just valid_mask to extract new points.
    return:
        Does not really return anything, but updates the following attributes:
        - additional_points_3d: 3D points in KF1 camera-centered coord frame.
        - additional_colors: corresponding colors
    """

    inpaint_mask = rendered_depth == 0
    rendered_depth_filled = rendered_depth.clone()
    inpaint_mask_onthefly = inpaint_mask.clone()

    def nearest_neighbor_inpainting(inpaint_mask, rendered_depth, window_size=20):
        """
        Perform nearest neighbor inpainting with a local search window.

        Parameters:
        inpaint_mask (torch.Tensor): Binary mask indicating missing values.
        rendered_depth (torch.Tensor): Input depth image.
        window_size (int): Size of the local search window.

        Returns:
        torch.Tensor: Inpainted depth image.
        """

        # Step 1: Find coordinates of invalid and valid pixels
        invalid_coords = torch.nonzero(inpaint_mask.squeeze(), as_tuple=False)
        valid_coords = torch.nonzero(~inpaint_mask.squeeze(), as_tuple=False)

        # Step 4: Use indices to copy depth values from valid to invalid pixels
        rendered_depth_copy = rendered_depth.clone()

        # Define half window size
        hw = window_size // 2

        # Iterate through invalid coordinates
        for idx in range(invalid_coords.size(0)):
            x, y = invalid_coords[idx, 0], invalid_coords[idx, 1]

            # Define local search window
            x_start, x_end = max(0, x - hw), min(rendered_depth.size(2), x + hw + 1)
            y_start, y_end = max(0, y - hw), min(rendered_depth.size(3), y + hw + 1)

            # Extract valid coordinates within the window
            local_valid_coords = valid_coords[(valid_coords[:, 0] >= x_start) & (valid_coords[:, 0] < x_end) &
                                              (valid_coords[:, 1] >= y_start) & (valid_coords[:, 1] < y_end)]

            # Compute distances and find nearest neighbor
            if local_valid_coords.size(0) > 0:
                dists = torch.cdist(invalid_coords[idx, :].unsqueeze(0).float(), local_valid_coords.float())
                min_idx = torch.argmin(dists)
                rendered_depth_copy[0, 0, x, y] = rendered_depth[0, 0, local_valid_coords[min_idx, 0], local_valid_coords[min_i

        return rendered_depth_copy
```

```python
    while inpaint_mask_onthefly.sum() > 0:  # iteratively inpaint depth until all depth holes are filled
        rendered_depth_filled = nearest_neighbor_inpainting(inpaint_mask_onthefly, rendered_depth_filled, window_size=50)
        inpaint_mask_onthefly = rendered_depth_filled == 0

    current_camera = convert_pytorch3d_kornia(self.current_camera, self.config["init_focal_length"]) if camera is None else cam
    points_2d = self.points if points_2d is None else points_2d
    points_3d = current_camera.unproject(points_2d, rearrange(rendered_depth_filled, "b c h w -> (w h b) c"))
    points_3d[..., :2] = - points_3d[..., :2]
    inpaint_mask = rearrange(inpaint_mask, "b c h w -> (w h b) c")
    colors = rearrange(image, "b c h w -> (w h b) c")
    if valid_mask is None:
        extract_mask = inpaint_mask[:, 0].bool()
    else:
        extract_mask = rearrange(valid_mask, "b c h w -> (w h b) c")[:, 0].bool()
    additional_points_3d = points_3d[extract_mask]

    # original_points_3d = points_3d[~inpaint_mask[:, 0]]
    # save_point_cloud_as_ply(original_points_3d, "tmp/original_points_3d.ply", colors[~inpaint_mask[:, 0]])
    # save_point_cloud_as_ply(additional_points_3d, "tmp/additional_points_3d.ply", colors[inpaint_mask[:, 0]])

    additional_colors = colors[extract_mask]

    # remove additional points that are behind the camera
    backward_points = (- additional_points_3d[..., 2]) > current_camera.tz
    additional_points_3d = additional_points_3d[~backward_points]
    additional_colors = additional_colors[~backward_points]

    self.additional_points_3d = torch.cat([self.additional_points_3d, additional_points_3d], dim=0)
    self.additional_colors = torch.cat([self.additional_colors, additional_colors], dim=0)

    if append_depth:
        self.depths.append(rendered_depth_filled.cpu())
```

Appendix – Occlusion handling by re-rendering consiste

```python
@torch.no_grad()
def update_additional_point_cloud(self, rendered_depth, image, valid_mask=None, camera=None, points_2d=None, append_depth=False
    """
    args:
        rendered_depth: Depth relative to camera. Note that KF2 camera is represented in KF1 camera-centered coord frame.
        valid_mask: if None, then use inpaint_mask (given by rendered_depth == 0) to extract new points.
            if not None, then just valid_mask to extract new points.
    return:
        Does not really return anything, but updates the following attributes:
        - additional_points_3d: 3D points in KF1 camera-centered coord frame.
        - additional_colors: corresponding colors
    """

    inpaint_mask = rendered_depth == 0
    rendered_depth_filled = rendered_depth.clone()
    inpaint_mask_onthefly = inpaint_mask.clone()

    def nearest_neighbor_inpainting(inpaint_mask, rendered_depth, window_size=20):
        """
        Perform nearest neighbor inpainting with a local search window.

        Parameters:
        inpaint_mask (torch.Tensor): Binary mask indicating missing values.
        rendered_depth (torch.Tensor): Input depth image.
        window_size (int): Size of the local search window.

        Returns:
        torch.Tensor: Inpainted depth image.
        """

        # Step 1: Find coordinates of invalid and valid pixels
        invalid_coords = torch.nonzero(inpaint_mask.squeeze(), as_tuple=False)
        valid_coords = torch.nonzero(~inpaint_mask.squeeze(), as_tuple=False)

        # Step 4: Use indices to copy depth values from valid to invalid pixels
        rendered_depth_copy = rendered_depth.clone()

        # Define half window size
        hw = window_size // 2

        # Iterate through invalid coordinates
        for idx in range(invalid_coords.size(0)):
            x, y = invalid_coords[idx, 0], invalid_coords[idx, 1]

            # Define local search window
            x_start, x_end = max(0, x - hw), min(rendered_depth.size(2), x + hw + 1)
            y_start, y_end = max(0, y - hw), min(rendered_depth.size(3), y + hw + 1)

            # Extract valid coordinates within the window
            local_valid_coords = valid_coords[(valid_coords[:, 0] >= x_start) & (valid_coords[:, 0] < x_end) &
                                              (valid_coords[:, 1] >= y_start) & (valid_coords[:, 1] < y_end)]

            # Compute distances and find nearest neighbor
            if local_valid_coords.size(0) > 0:
                dists = torch.cdist(invalid_coords[idx, :].unsqueeze(0).float(), local_valid_coords.float())
                min_idx = torch.argmin(dists)
                rendered_depth_copy[0, 0, x, y] = rendered_depth[0, 0, local_valid_coords[min_idx, 0], local_valid_coords[min_i

        return rendered_depth_copy
```

```python
    while inpaint_mask_onthefly.sum() > 0:  # iteratively inpaint depth until all depth holes are filled
        rendered_depth_filled = nearest_neighbor_inpainting(inpaint_mask_onthefly, rendered_depth_filled, window_size=50)
        inpaint_mask_onthefly = rendered_depth_filled == 0

    current_camera = convert_pytorch3d_kornia(self.current_camera, self.config["init_focal_length"]) if camera is None else cam
    points_2d = self.points if points_2d is None else points_2d
    points_3d = current_camera.unproject(points_2d, rearrange(rendered_depth_filled, "b c h w -> (w h b) c"))
    points_3d[..., :2] = - points_3d[..., :2]
    inpaint_mask = rearrange(inpaint_mask, "b c h w -> (w h b) c")
    colors = rearrange(image, "b c h w -> (w h b) c")
    if valid_mask is None:
        extract_mask = inpaint_mask[:, 0].bool()
    else:
        extract_mask = rearrange(valid_mask, "b c h w -> (w h b) c")[:, 0].bool()
    additional_points_3d = points_3d[extract_mask]

    # original_points_3d = points_3d[~inpaint_mask[:, 0]]
    # save_point_cloud_as_ply(original_points_3d, "tmp/original_points_3d.ply", colors[~inpaint_mask[:, 0]])
    # save_point_cloud_as_ply(additional_points_3d, "tmp/additional_points_3d.ply", colors[inpaint_mask[:, 0]])

    additional_colors = colors[extract_mask]

    # remove additional points that are behind the camera
    backward_points = (- additional_points_3d[..., 2]) > current_camera.tz
    additional_points_3d = additional_points_3d[~backward_points]
    additional_colors = additional_colors[~backward_points]

    self.additional_points_3d = torch.cat([self.additional_points_3d, additional_points_3d], dim=0)
    self.additional_colors = torch.cat([self.additional_colors, additional_colors], dim=0)

    if append_depth:
        self.depths.append(rendered_depth_filled.cpu())
```

Scene description generation

"You are an intelligent scene generator. Imaging you are flying through a scene or a sequence of scenes, and there are 3 most significant common entities in each scene. Please tell me what sequentially next scene would you likely to see? You need to generate the scene name and the 3 most common entities in the scene. The scenes are sequentially interconnected, and the entities within the scenes are adapted to match and fit with the scenes. You also have to generate a brief background prompt about 50 words describing the scene. You should not mention the entities in the background prompt. If needed, you can make reasonable guesses."

$$\mathcal{S}_{i+1} = g_{\text{description}}(\mathcal{J}, \mathcal{M}_i), \quad (2)$$

$$\mathcal{M}_{i+1} = \mathcal{M}_i \cup \{\mathcal{S}_{i+1}\}.$$

$$\mathcal{S}_i = \{S, O_i, B_i\}$$

$g_{\text{description}}$ = LLM (GPT4)

$\mathcal{S}_{i+1}$ = i+1$^{\text{th}}$ scene description

S = a style

O_i = objects in the scene

B_i = the background in the scene

The scene description generation takes a set of past and current scene descriptions as input and predicts the subsequent scene description

Visual validation

Input

A beautiful village

1 Scene description generation

Scene description memory
- ...
- Scene *i*: *Village* ...

LLM

Next scene description
- Scene *i* + 1 : *Snowfield* ...

"Along the four borders of this image, is there anything that looks like thin border, thin stripe, photograph border, painting border, or painting frame? Please look very closely to the four edges and try hard, because the borders are very slim and you may easily overlook them. If you are not sure, then please say yes."

Point cloud

Rendered partial image

Generated new scene

Re-generate or go

Submit

3 Visual validation

*Are there any **unwanted effects** in the generated scene image?*

Unwanted effects:
- Painting frame
- Photo border
- Out-of-focus objects
- ...

VLM

Yes → Re-generate the scene.
No → Generate the next scene.