Supplementary Material

In this supplementary material, we first present further implementation details corresponding to the three main components of our pipeline, *i.e.*, the voxel world generation stage (in Sec. A), the world-guided video generation stage (in Sec. B), and the dynamic 3DGS scene generation stage (in Sec. C). Additional details about the large-scale generation and user study can be found in Sec. D and Sec. E.

A. Additional Details of the Voxel World Generation

A.1. Voxel Diffusion Model Training

Following XCube [42], we first train a sparse structure Variational Autoencoder (VAE) to encode the semantic sparse voxel grid $\mathbf{D}^{\text{vx}}$ into a dense latent feature cube $\mathbf{X}^{\text{vx}}$, and then train an HD map and 3D bounding box conditioned diffusion model on the latent representation $\mathbf{X}^{\text{vx}}$. Here, we do not apply the hierarchical generation since one diffusion is enough for a voxel size of 0.2m in a range of $51.2\text{m} \times 51.2\text{m}$. The diffusion loss is defined with a $\boldsymbol{v}$-parameterization:

$$\mathcal{L}_{\text{Diffusion}} = \mathbb{E}_{t,\mathbf{X}^{vx},\boldsymbol{\epsilon}\sim\mathcal{N}(0,\mathbf{I})}\left[\left\|\boldsymbol{v}\left(\sqrt{\bar{\alpha}_t}\mathbf{X}^{vx} + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right) - \left(\sqrt{\bar{\alpha}_t}\boldsymbol{\epsilon} - \sqrt{1-\bar{\alpha}_t}\mathbf{X}^{vx}\right)\right\|_2^2\right], \tag{S.2}$$

where $\boldsymbol{v}(\cdot)$ is the diffusion network, t is the randomly sampled diffusion timestamp from $[0, 1000]$, and $\bar{\alpha}_t$ is the scheduling factor for the diffusion process. More details can be found in [42].

A.2. Voxel Diffusion Model Sampling

We use DDIM [48] as our sampler for the distribution of the latent feature cube $\mathbf{X}^{\text{vx}}$ given HD maps and 3D bounding boxes as conditions. We set the denoising step to 100 and the classifier-free guidance [22] weight to 2.0 during inference. To avoid inconsistency from VAE decoding, we do not decode the latent feature cube $\mathbf{X}^{\text{vx}}$ until all the chunks are generated and fused during our outpainting procedure. Then, we use the decoder from the sparse structure VAE to recover the 3D sparse voxel world with the semantic logit for each voxel.

B. Additional Details of the World-Guided Video Generation

B.1. Semantic Buffer Construction

We show the RGB value of each semantic category in the semantic buffer in Tab. S4. We cluster similar semantic categories together for the coloring.

Semantic Categories	RGB Values
SIGN, TRAFFIC_LIGHT, CONSTRUCTION_CONE	(0.4, 0.7608, 0.6471)
MOTORCYCLIST, BICYCLIST, PEDESTRIAN, BICYCLE, MOTORCYCLE	(0.9882, 0.5529, 0.3843)
CAR, TRUCK, BUS, OTHER_VEHICLE	Varying Colors
CURB, LANE_MARKER	(1.0, 0.8510, 0.1843)
VEGETATION, TREE_TRUNK	(0.3020, 0.6863, 0.2902)
WALKABLE, SIDEWALK	(0.5529, 0.6275, 0.7961)
BUILDING	(0.8980, 0.7686, 0.5804)
ROAD, OTHER_GROUND	(0.7020, 0.7020, 0.7020)
UNDEFINED	(0.1216, 0.4706, 0.7059)
POLE	(0.8000, 0.9216, 0.7725)

Table S4. **Semantic categories and their RGB values in the semantic buffer**. The RGB values in the table range from 0 to 1. In practice, we rescale the above values from -1 to 1 for the encoder.

Note that we use a varying RGB value across different instances, as mentioned in the main paper for the semantic categories of `CAR`, `TRUCK`, `BUS`, `OTHER_VEHICLE`. We randomly select a color in the *PuRd* color map from Matplotlib [26], as shown in Fig. S13.

Figure S13. **Color map for vehicle instances.** We randomly pick a color in *PuRd* color map from Matplotlib for each vehicle instance in the semantic buffer.

B.2. First Frame Initialization with ControlNet

We implement the semantic buffer conditioned FLUX [1] ControlNet with the `diffusers` [51] library. We train the model with an equivalent batch size of 64 for 48 GPU days using NVIDIA A100 GPUs. We show the results of our semantic buffer conditioned ControlNet in Fig. S14.

C. Additional Details of Dynamic 3D Gaussian Scene Generation

C.1. Voxel Branch

In this work, we adopt a voxel size of 0.2m in the voxel world generation stage, which is coarser than the voxel size of 0.1m used in SCube [43] but is sufficient for video model conditioning in our application. To further enhance the detail in the scene generation stage, we subdivide the generated voxels into voxel size of 0.1m before the per-voxel Gaussian attribute decoding in the voxel branch. To encode image features, we use several convolutional layers to transform the RGB images $\mathbf{I}$ into $2\times$ downsampled feature maps with a channel of 64. We then unproject the feature maps to the voxel world to assign each voxel a voxel feature by max-pooling, and use a 3D UNet to process the 3D voxel feature grid. The final output channel of the 3D UNet model is 56 for 4 Gaussians per voxel, where each 3D Gaussian uses 14 channels for RGB (3), rotation (4), scale (3),

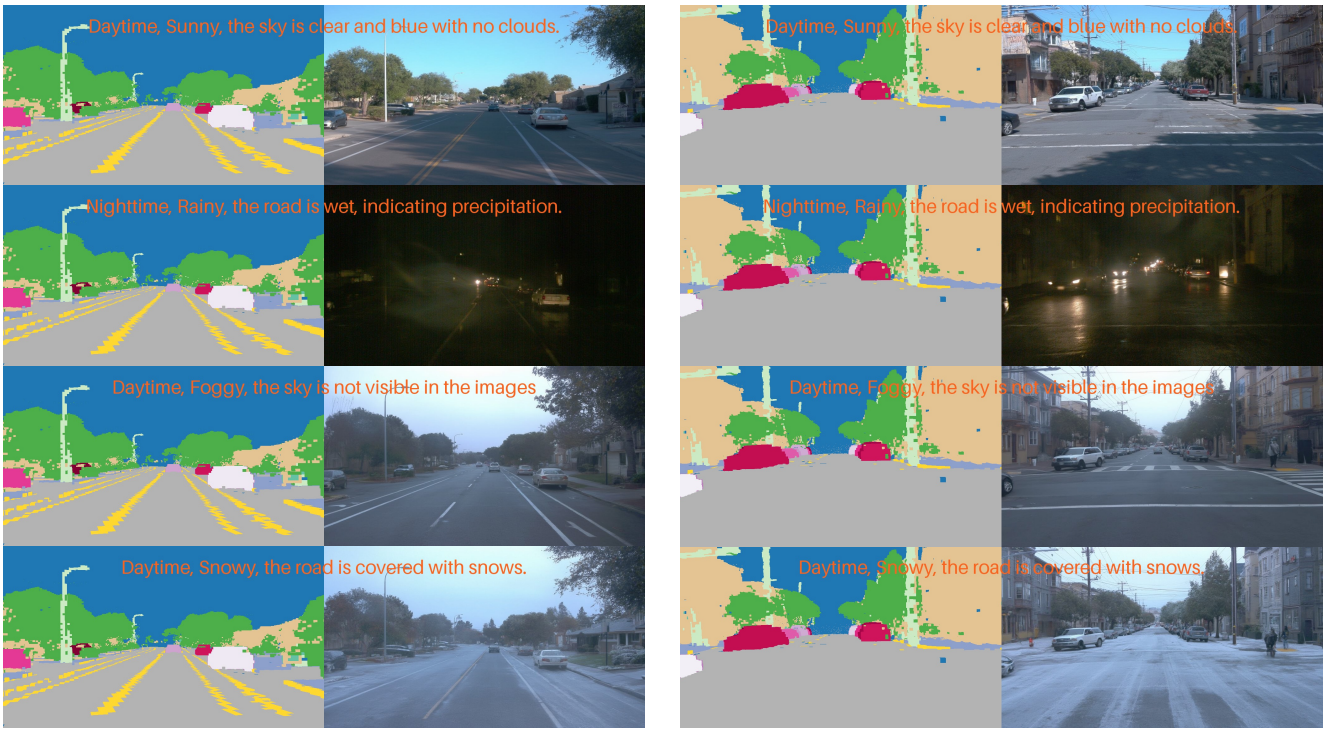

Figure S14. **First Frame Generation with Different Text Prompts.** We show additional results on generating the initial frame using semantic buffers with ControlNet [73] based on FLUX [1].

opacity (1) and relative position (3). The absolute 3D center of the 3D Gaussian is converted from the relative position using the same convention in SCube [43].

Network Architecture. The 3D UNet [43] has a base channel 64 and a channel multiplier of $[1, 2, 4]$ for each resolution stage. Within each resolution stage, we use 3D convolutional blocks with kernel size 3 for the feature encoding.

C.2. Pixel Branch

We take original RGB images $\mathbf{I} \in \mathbb{R}^{h \times w \times 3}$, randomly masked voxel depths $\tilde{\mathbf{Z}} \in \mathbb{R}^{h \times w \times 1}$ and intermediate features $\mathbf{F}_{\text{DAV2}} \in \mathbb{R}^{h \times w \times 32}$ from Depth Anything V2 [66] as the input of our 2D UNet model, and predict pixel-aligned 3D Gaussians in this branch. For the randomly masked voxel depth $\tilde{\mathbf{Z}}$, we zero out each non-overlapping 16×16 image patch with a probability of 0.5 in the training stage. Note that we use the full voxel depth $\mathbf{Z}$ in the inference stage (but they still do not cover the mid-ground region). For the Depth Anything V2 feature, we extract the fused feature from the Depth-Anything-V2-`Large` before the depth prediction head. We then apply several convolutional layers and upsampling layers to resize this fused feature to the image resolution while reducing the number of feature channels to 32. The total input channel for our 2D UNet is $3 + 1 + 32 = 36$. The final output channel of the UNet model is 24 for 2 Gaussians per pixel, where each 3D Gaussian uses 12 channels for RGB (3), rotation (4), scale (3), opacity (1) and depth (1).

We use a similar parameterization for the 3D Gaussians as GS-LRM [72]. Details for the parameterization can be found in the Appendix of GS-LRM [72]. The only difference is that we predict depth instead of distance for each Gaussian. This necessitates an additional step to convert the predicted depth into distance to determine the center of the 3D Gaussians in 3D space, utilizing the camera's origin and the ray direction. For a Gaussian i, the 3D position is obtained as:

$$\begin{aligned} \omega^i &= \sigma(\mathbf{G}^i_{\text{depth}}), \\ z^i &= (1 - \omega^i) \cdot z_{\text{near}} + \omega^i \cdot z_{\text{far}}, \\ t^i &= z^i / \cos(\text{ray}^i_d, \text{ray}^{\text{look-at}}_d), \\ \text{xyz}^i &= \text{ray}^i_o + t^i * \text{ray}^i_d, \end{aligned} \tag{S.3}$$

where the $\mathbf{G}^i_{\text{depth}}$ is the model's raw depth output for Gaussian i's prediction, and σ is the sigmoid function normalizing the raw output to a weight scalar w^i. Here z^i is the depth and t^i is the corresponding distance; we set $z_{\text{near}} = 0.5$ and $z_{\text{far}} = 300$ in our cases.

Network Architecture. The 2D UNet [45] has a base channel 32 and a channel multiplier of $[1, 2, 4, 8]$ for each resolution stage. Within each resolution stage, we use 2 ResNet blocks with kernel size 3 for the feature encoding, and an extra Conv/Transposed Conv for the downsampling and upsampling.

C.3. Sky Modeling

We use a lightweight transformer encoder to compress the sky features into a latent feature vector $\mathbf{c} \in \mathbb{R}^{192}$. We prepare a learnable query token $\mathbf{c}_{query}$, similar to the `[CLS]` token in ViT [11] for high-level sky feature learning, to interact with all the patches belonging to the sky area (we patchify the image with an 8×8 patch size and only keep those patches in the sky region). The appearance of the sky will be encoded in $\mathbf{c}$ as follows:

$$\mathbf{c}, \tilde{\mathbf{p}}^i_{i \in \{1,2,\ldots,m\}} = \text{TransformerEncoder}(\mathbf{c}_{query}, \mathbf{p}^i_{i \in \{1,2,\ldots,m\}}), \tag{S.4}$$

where $\mathbf{p}^i$ and $\tilde{\mathbf{p}}^i$ are the sky patches before and after the transformer encoder, m is the number of sky patches. A learning-based positional embedding is applied to the camera ray direction of each patch.

Then we use AdaGN [27] to modulate an MLP to take a viewing direction $\mathbf{d}$ and output the corresponding RGB value given the sky vector $\mathbf{c}$. Specifically, we first use learnable embedding to transform the view direction vector to a high-frequency representation $\gamma(\mathbf{d}) \in \mathbb{R}^{192}$, then normalize the high-frequency view vector $\gamma(\mathbf{d})$ by LayerNorm without the affine term (we denote the normalized one $\mathbf{x}$). To condition on the sky vector $\mathbf{c}$, we use a linear layer to predict the $\mathbf{scale}$ and $\mathbf{shift}$ from $\mathbf{c}$ for the modulation, i.e., $\mathbf{x} = \mathbf{x} \cdot (1 + \mathbf{scale}) + \mathbf{shift}$, and finally decode into a 3-dimensional RGB color with another linear layer.

D. Additional Details of Large-scale Scene Generation

We can reuse the ego trajectory from the Waymo Open Dataset [49] to build the voxel world and generate the guidance buffers. For some parts of the scenes without any coverage of the ego-car trajectories, we use a customized strategy to

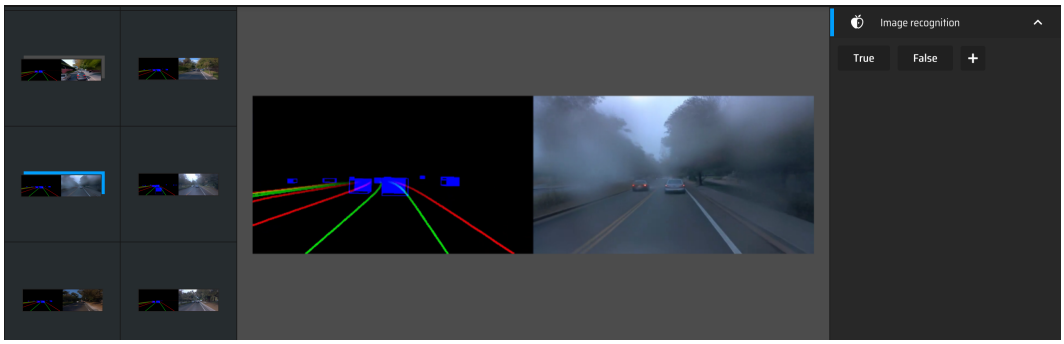

Figure S15. **User Study Interface Provided by MakeSense AI**. Users are required to judge if the HD map projection (left) aligns with the RGB image (right). Users are told that the red line is the road boundary, the green line is the lane line, and the blue bounding box is the vehicle.

generate the trajectories for the guidance buffers. While this can be realized by utilizing existing planning modules given HD maps as input, we further implement a real-time viewer based on Viser [40] for the user to drive in the voxel world. It will record the trajectory and render the voxel scene into guidance buffers.

E. Additional Details of User Study

We project the HD map and 3D bounding boxes onto the image plane as a reference for the user to judge if the generated video (image) aligns with the HD map condition. We generated 63 videos for ours and the baseline methods with different HD map layouts and text prompts, and extracted the 40^{th}, 80^{th}, and 120^{th} frames from the generated video and ask users to evaluate their alignment. We collected 180 samples for each frame index and calculated their positive rate. The user study is conducted with the platform MakeSense AI; see the user interface in Fig. S15.