Video2Game: Real-time, Interactive, Realistic and Browser-Compatible Environment from a Single Video

Supplementary Material

A. Additional Results and Analysis

More qualitative results. We provide more qualitative comparison results among baselines [8, 12, 16, 20, 23] and our proposed method. For comparisons between Instant-NGP [16], Nerfacto [20], 3D Gaussian Splatting [12] and our base NeRF in KITTI-360 dataset [14] and Garden scene in Mipnerf-360 Dataset [7], see Fig. 1 and Fig. 2. We observe that our method renders less noisy geometries while maintaining a superior or comparable visual quality. Especially, 3D Gaussian Splatting [12] fails to learn correct 3D orientations of Gaussians in sparse settings like KITTI-360 [14], leading to weird color renderings in novel views and noisy geometry rendering. As for mesh rendering qualitative comparison between [8, 23] and ours, see Fig. 3. Our mesh rendering has similar and comparable rendering results in Garden scene [7]. However, in KITTI-360 dataset [14] which is extremely large-scale and open, the performance of MobileNeRF [8] drops dramatically and BakedSDF [23] generates slightly blurry in road-aside car rendering, while our mesh rendering is not only superior in KITTI-360 dataset [14], but it also maintains stable performance across different datasets.

B. Dataset Details

B.1. KITTI-360 Dataset

We build "KITTI-Loop game" based on KITTI-360 Dataset [14]. We use frames from sequence 0. The loop we build in our game utilizes frames 4240-4364, 6354-6577, 7606-7800, and 10919-11050. We compose those four snippets into a closed loop in our game. For baseline comparison and ablation study, we perform experiments on two blocks containing frames 7606-7665 and 10919-11000. We split the validation set every 10 frames (frames 7610, 7620, 7630, 7640, 7650, and 7660 for the first block; frames 10930, 10940, 10950, 10960, 10970, 10980, 10990 for the second block). We report the average metrics of two blocks.

B.2. Mipnerf-360 Dataset

We build the "Gardenvase game" based on the Garden scene of Mipnerf-360 Dataset [7]. We split the validation set every 20 frames.

B.3. VRNeRF Dataset

We build our robot simulation environment based on the "table" scene of VRNeRF Dataset [22].

C. Video2Game Implementation Details

C.1. Base NeRF Training Details

Network architecture and hyper-parameters Our network consists of two hash grid encoding [16] components $\texttt{It}_d$ and $\texttt{It}_c$ and MLP headers $\texttt{MLP}^d_{\theta_d}$, $\texttt{MLP}^c_{\theta_c}$, $\texttt{MLP}^s_{\theta_s}$, and $\texttt{MLP}^n_{\theta_n}$, each with two 128 neurons layers inside. Taking 3D position input $\mathbf{x}$, density σ is calculated following $\sigma = \texttt{MLP}^d_{\theta_d}(\texttt{It}_d(\texttt{Ct}(\mathbf{x}), \Phi_d))$. Color feature $f = \texttt{It}_c(\texttt{Ct}(\mathbf{x}), \Phi_c)$. Then we calculate $\mathbf{c}, s, \mathbf{n}$ from feature f and direction $\mathbf{d}$ through $\mathbf{c} = \texttt{MLP}^c_{\theta_c}(f, \mathbf{d})$, $s = \texttt{MLP}^s_{\theta_s}(f)$ and $\mathbf{n} = \texttt{MLP}^n_{\theta_n}(f)$ respectively. All parameters involved in training our base NeRF can be represented as NGP voxel features $\Phi = \{\Phi_d, \Phi_c\}$ and MLP parameters $\theta = \{\theta_d, \theta_c, \theta_s, \theta_n\}$. To sum up, we get $\mathbf{c}, \sigma, s, \mathbf{n} = F_\theta(\mathbf{x}, \mathbf{d}; \Phi) = \texttt{MLP}_\theta(\texttt{It}(\texttt{Ct}(\mathbf{x}), \Phi), \mathbf{d})$. The detailed diagram of our NeRF can be found in Fig. 4.

Our hash grid encoding [16] is implemented by tiny-cuda-nn [15], and we set the number of levels to 16, the dimensionality of the feature vector to 8 and Base-2 logarithm of the number of elements in each backing hashtable is 19 for $\texttt{It}_d$ and 21 for $\texttt{It}_c$. As for activation functions, we use ReLU [17] inside all MLPs, Softplus for density σ output, Sigmoid for color $\mathbf{c}$ output, Softmax for semantic s output and no activation function for normal $\mathbf{n}$ output (We directly normalize it instead).

KITTI-Loop additional training details In KITTI-Loop which uses KITTI-360 Dataset [14], we also leverage stereo depth generated from DeepPruner [10]. Here we calculate the actual depth from disparity and the distance between binocular cameras and adopt L1 loss to regress. We **haven't** used any LiDAR information to train our base NeRF in KITTI-360 Dataset [14].

C.2. Mesh Extraction and Post-processing Details

Mesh Post-processing details In mesh post-processing, we first utilize all training camera views to prune the vertices and faces that can't be seen. Next, we delete those unconnected mesh components that have a small number of faces below a threshold so as to delete those floaters in the mesh. Finally, we merge close vertices in the mesh, then perform re-meshing using PyMesh [3] package, which iteratively splits long edges over a threshold, merges short edges below a threshold and removes obtuse triangles. Remeshing helps us get better UV mapping results since it makes the mesh "slimmer" (less number of vertices and

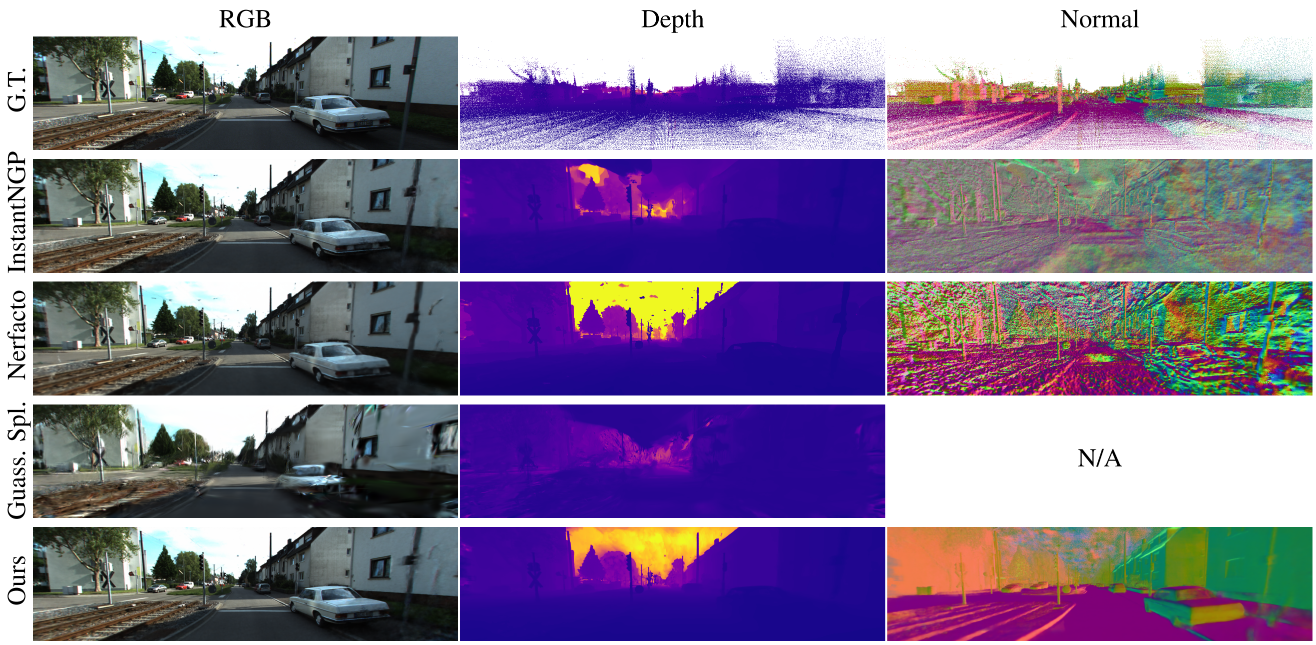

Figure 1. **Qualitative comparisons among NeRF models [16, 20] and 3D Gaussian Splatting [12] in KITTI-360 Dataset [14].** We provide NeRF rendering depths and normals for comparison as well. For 3D Gaussian Splatting, only rendering depth is provided. Here we consider depths measured by LiDAR point cloud in KITTI-360 and compute normals based on it as our ground truth.

faces) and has similar lengths of edges. After the post-processing, we get meshes with a relatively small number of vertices and faces while still effectively representing the scene.

Special settings in KITTI-Loop In KITTI-Loop, we partition the whole loop into 14 overlapping blocks. Since we adopt pose normalization and contract space in each block when training, it needs alignments when we compose them together. For each block, we first generate its own mesh. We partition the whole contract space ($[-1, 1]^3$) into 3*3*3 regions, and perform marching cubes with the resolution of 256*256*256 in each region. We then transform those vertices back from contract space to the coordinates before contraction. We then perform mesh post-processing here. To compose each part of the mesh in KITTI-Loop together, we then transform the mesh to KITTI-Loop world coordinates. For those overlapping regions, we manually define the block boundary and split the mesh accordingly. Finally, we add a global sky dome over the KITTI-Loop.

C.3. NeRF Baking Details

For each extracted mesh, we bake the NeRF's color and specular components to it with nvdiffrast [13].

GLSL MLP settings We adopt a two-layer tiny MLP with 32 hidden neurons. We use ReLU [17] activation for the first layer and sigmoid for the second. We re-implement that MLP with GLSL code in Three.js renderer's shader.

Initialization of texture maps and MLP shader Training the textures $\mathbf{T} = [\mathbf{B}; \mathbf{S}]$ and MLP shader $\texttt{MLP}_\theta^{\text{shader}}$ all from scratch is slow. Instead, we adopt an initialization procedure. Inspired by [21, 23], we encode the 3D space by hash encoding [16] $\texttt{It}^M$ and an additional MLP $\texttt{MLP}_{\theta_0}^M$. Specifically, we first rasterize the mesh into screen space, obtain the corresponding 3D position x_i on the surface of the mesh within each pixel, transform it into contract space $\texttt{Ct}(x_i)$, and then feed it into $\texttt{It}^M$ and $\texttt{MLP}_{\theta_0}^M$ to get the base color $\mathbf{B}_i$ and specular feature $\mathbf{S}_i$, represented as $\mathbf{B}_i$, $\mathbf{S}_i = \texttt{MLP}_{\theta_0}^M(\texttt{It}^M(\texttt{Ct}(x_i), \Phi_0))$. Finally we computes the sum of the view-independent base color $\mathbf{B}_i$ and the view-dependent specular color following $\mathbf{C}_\text{R} = \mathbf{B}_i + \texttt{MLP}_\theta^{\text{shader}}(\mathbf{S}_i, \mathbf{d}_i)$. The parameters Φ_0, θ_0, θ are optimized by minimizing the color difference between the mesh model and the ground truth: $\mathcal{L}^{\text{render}}_{\text{initialize}\Phi_0,\theta_0,\theta} = \sum_\mathbf{r} \|\mathbf{C}_\text{R}(\mathbf{r}) - \mathbf{C}_\text{GT}(\mathbf{r})\|_2^2$. Anti-aliasing is also adopted in the initialization step by perturbing the optical center of the camera. With learned parameters, every corresponding 3D positions x_i in each pixel of 2D unwrapped texture maps $\mathbf{T} = [\mathbf{B}; \mathbf{S}]$ is initialized following $\mathbf{B}_i$, $\mathbf{S}_i = \texttt{MLP}_{\theta_0}^M(\texttt{It}^M(\texttt{Ct}(x_i), \Phi_0))$ and the parameters of $\texttt{MLP}_\theta^{\text{shader}}$ is directly copied from initialization stage.

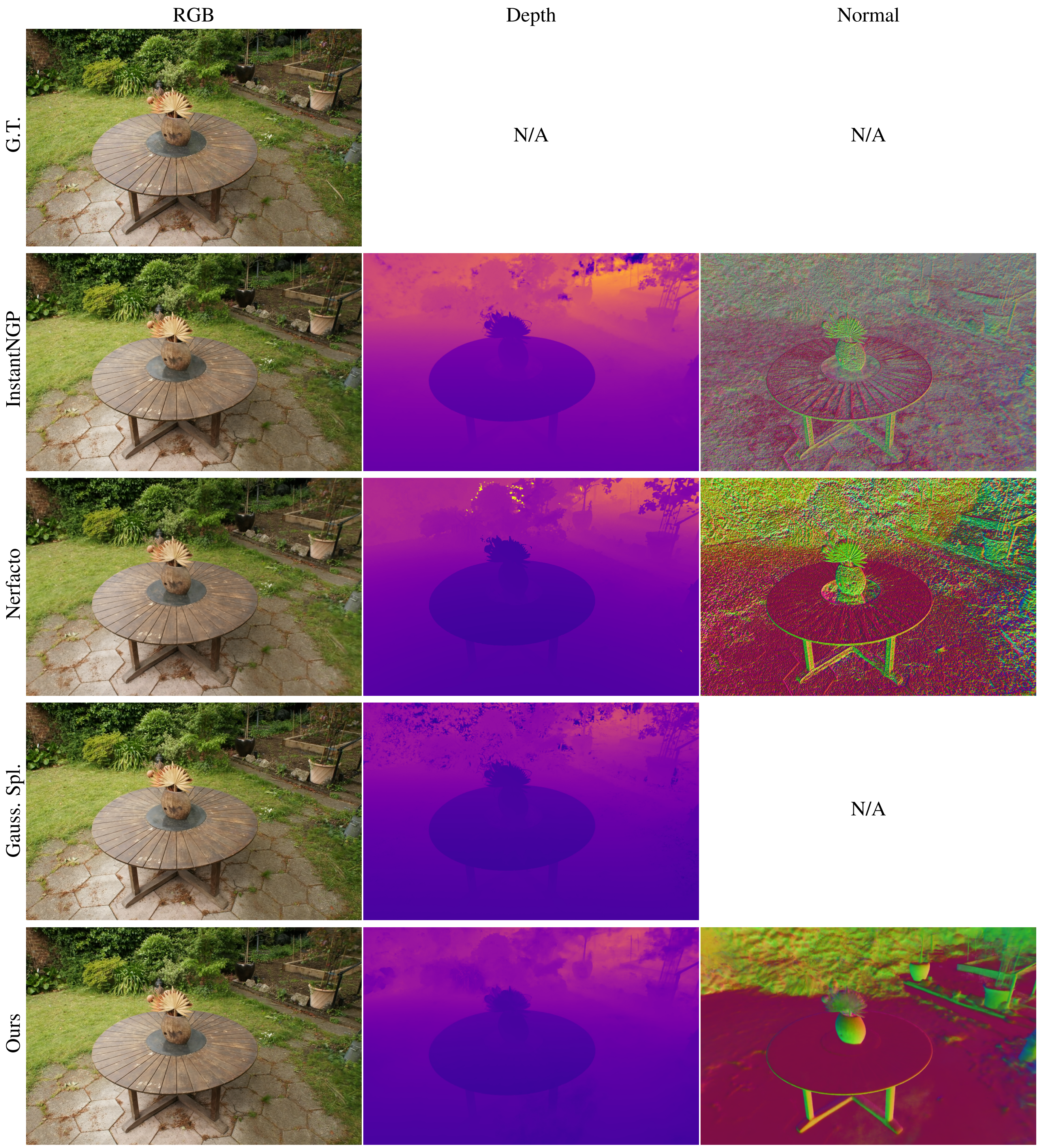

Figure 2. **Qualitative comparisons among NeRF models [16, 20] and 3D Gaussian Splatting [12] in Garden scene [7].** We provide NeRF rendering depths and normals for comparison as well. For 3D Gaussian Splatting, only rendering depth is provided.

C.4. Physical Module Details

Physical dynamics It is important to note that our approach to generating collision geometries is characterized by meticulous design. In the case of box collider generation, we seamlessly repurpose the collider used in scene decomposition. When it comes to triangle mesh colliders,

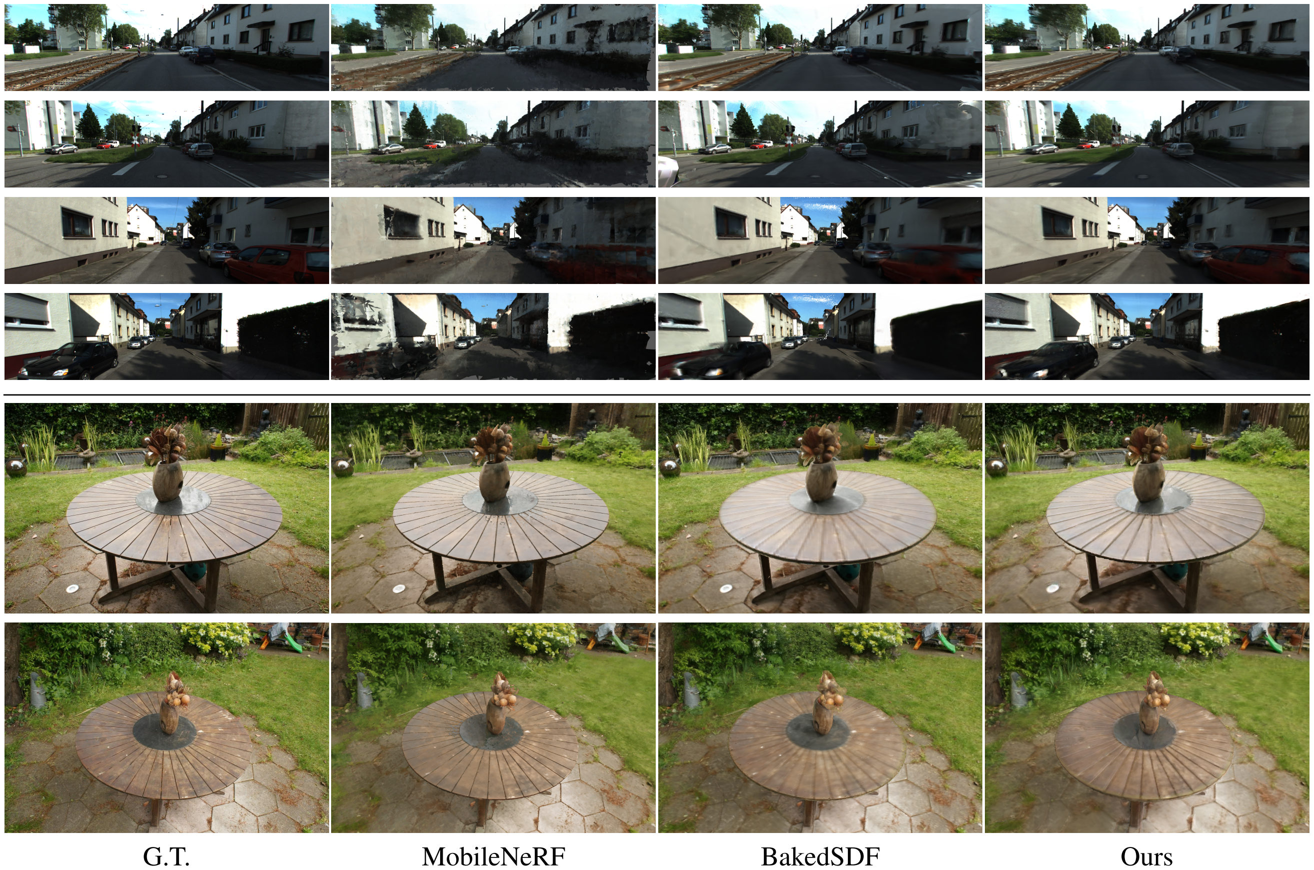

Figure 3. **Qualitative comparisons in mesh rendering.** We compare our proposed mesh rendering method to MobileNeRF [8] and BakedSDF [23] in KITTI-360 Dataset [14] and Garden scene [7].

we prioritize collision detection efficiency by simplifying the original mesh. Additionally, for convex polygon colliders, we leverage V-HACD [6] to execute a precise convex decomposition of the meshes.

Physical parameters assignments. Physical parameters for static objects, such as the ground, were set to default values. For interactive instances like cars and vases, we could query GPT-4 with box highlights and the prompts as shown on the left. Note that we reason about mass and friction using the same prompt. The output is a range, and we find that selecting a value within this range provides reasonable results. See Fig. 5 for an example. Unit conversion from the metric system to each engine's specific system is needed.

C.5. Robot Simulation Details

Data preparation We demonstrate the potential of leveraging Video2Game for robot simulation using the VRNeRF [22] dataset. We reconstruct the scene and segment simulatable rigid-body objects (*e.g.*, the fruit bowl on the table). Then collision models are generated for those physical entities for subsequent physical simulations.

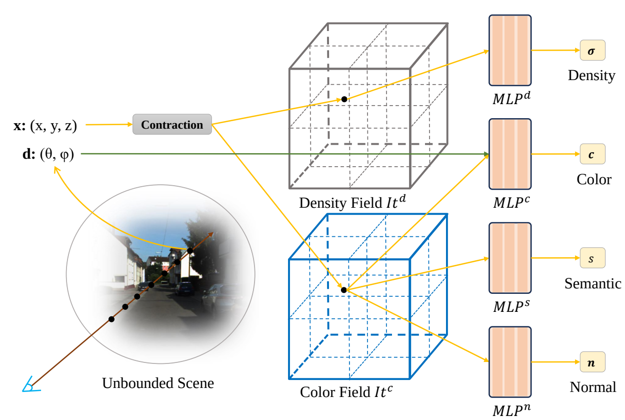

Figure 4. **Video2Game NeRF Module:** The diagram of our designed NeRF.

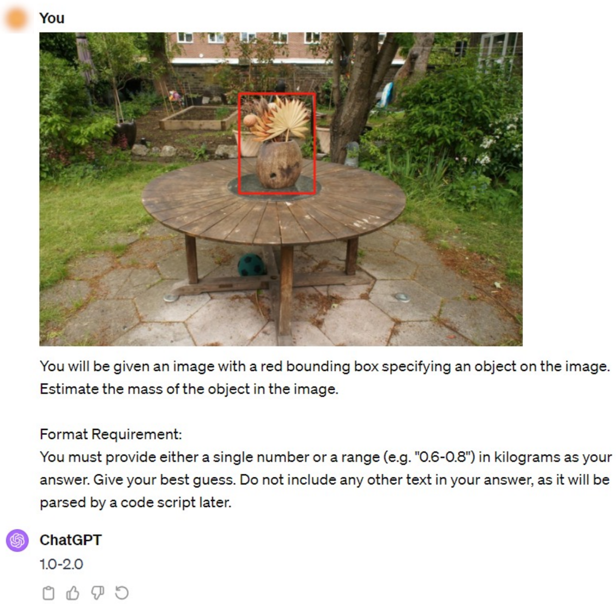

Figure 5. **Example of physical property reasoning by GPT-4.**

Physical simulation To simulate the interactions between robots and physical entities in a dynamic environment, we employ PyBullet [9], a Python module designed for physics simulations in the realms of games, robotics, and machine learning. Given the intricate dynamics of articulated robots, PyBullet serves as a powerful tool for conducting physics calculations within the context of robot simulation. Our approach involves loading all generated collision models and URDF [1] files for both the Stretch Robot [4] and Fetch Robot [1]. Utilizing PyBullet's integrated robotic inverse kinematics, we can effectively control the mechanical arms of the robots to interact with surrounding objects. Specifically, for the Stretch Robot, we define a predefined path for its arm, enabling it to exert a direct force to displace the central bowl off the table. On the other hand, for the Fetch Robot, we leverage the collision boxes specified in its URDF file. Our manipulation involves grasping the corresponding collision model of the central bowl on the table, eschewing the use of the magnetic gripper for object control. Subsequently, the robot lifts the bowl and relocates it to a different position. Following the simulations in PyBullet, we extract physics calculation results, including joint values and the position of the robots' base link. These results are then exported and integrated into the rendering engine of Three.js for further visualization and analysis.

Rendering in robot simulation We import the URDF files of our robots into our engine using the urdf-loader [5] in Three.js, a library that facilitates the rendering and configuration of joint values for the robots. Leveraging pre-computed physics simulations in PyBullet, which are based on our collision models, we seamlessly integrate these simulations into the Three.js environment. This integration allows us to generate and render realistic robot simulation videos corresponding to the simulated physics interactions.

[1] http://wiki.ros.org/urdf

C.6. Training time

For base NeRF training, it takes 8 hours for training 150k iterations on an A6000. For the NeRF baking procedure, the initialization and training take 4 hours on an A5000.

D. Baseline Details

D.1. Instant-NGP

We adopt the re-implementation of Instant-NGP [16] in [2]. We choose the best hyper-parameters for comparison. For normal rendering, we calculate by the derivative of density value.

D.2. Nerfacto

Nerfacto is proposed in Nerfstudio [20], an integrated system of simplified end-to-end process of creating, training, and testing NeRFs. We choose their recommended and default settings for the Nerfacto method.

D.3. 3D Gaussian Splatting

For 3D Gaussian Splatting [12] training in Garden scene [7], we follow all their default settings. In the KITTI-360 Dataset, there are no existing SfM [18] points. We choose to attain those 3D points by LoFTR [19] 2D image matching and triangulation in Kornia [11] using existing camera projection matrixs and matching results. We choose their best validation result throughout the training stage by testing every 1000 training iterations.

D.4. MobileNeRF

In Garden scene [7], we directly follow the default settings of MobileNeRF [8]. For training in KITTI-360 Dataset [14], we adopt their "*unbounded* 360 scenes" setting for the configurations of polygonal meshes, which is aligned with KITTI-360 Dataset.

D.5. BakedSDF

We adopt the training codes of BakedSDF [23] in SDFStudio [24], from which we can attain the exported meshes with the resolution of 1024x1024x1024 by marching cubes. For the baking stage, we adopt three Spherical Gaussians for every vertices and the same hyper-parameters of NGP [16] mentioned in [23]. We follow the notation BakedSDF [23] used in its paper, where "offline" means volume rendering results.

E. Limitation

Although Video2Game framework could learn view-dependent visual appearance through its NeRF module and mesh module, it doesn't learn necessary material properties for physics-informed relighting, such as the metallic property of textures. Creating an unbounded, *relightable* scene from a single video, while extremely challenging, can further enhance realism. We leave this for future work.

References

[1] Fetch Mobile Manipulator. https://fetchrobotics.borealtech.com/robotics-platforms/fetch-mobile-manipulator/?lang=en. 5

[2] ngp-pl. https://github.com/kwea123/ngp_pl. 5

[3] PyMesh. https://github.com/PyMesh/PyMesh. 1

[4] Stretch® Research Edition. https://hello-robot.com/product. 5

[5] urdf-loaders. https://github.com/gkjohnson/urdf-loaders. 5

[6] V-HACD. https://github.com/kmammou/v-hacd. 4

[7] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *CVPR*, 2022. 1, 3, 4, 5

[8] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. *CVPR*, 2023. 1, 4, 5

[9] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021. 5

[10] Shivam Duggal, Shenlong Wang, Wei-Chiu Ma, Rui Hu, and Raquel Urtasun. Deeppruner: Learning efficient stereo matching via differentiable patchmatch. In *ICCV*, 2019. 1

[11] D. Ponsa E. Rublee E. Riba, D. Mishkin and G. Bradski. Kornia: an open source differentiable computer vision library for pytorch. In *Winter Conference on Applications of Computer Vision*, 2020. 5

[12] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics (ToG)*, 42(4):1–14, 2023. 1, 2, 3, 5

[13] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics*, 39(6), 2020. 2

[14] Yiyi Liao, Jun Xie, and Andreas Geiger. Kitti-360: A novel dataset and benchmarks for urban scene understanding in 2d and 3d. *IEEE TPAMI*, 2022. 1, 2, 4, 5

[15] Thomas Müller. tiny-cuda-nn, 2021. 1

[16] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM TOG*, 2022. 1, 2, 3, 5

[17] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. 1, 2

[18] Johannes L Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. In *CVPR*, 2016. 5

[19] Jiaming Sun, Zehong Shen, Yuang Wang, Hujun Bao, and Xiaowei Zhou. Loftr: Detector-free local feature matching with transformers, 2021. 5

[20] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Justin Kerr, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, David McAllister, and Angjoo Kanazawa. Nerfstudio: A modular framework for neural radiance field development. *arXiv*, 2023. 1, 2, 3, 5

[21] Jiaxiang Tang, Hang Zhou, Xiaokang Chen, Tianshu Hu, Errui Ding, Jingdong Wang, and Gang Zeng. Delicate textured mesh recovery from nerf via adaptive surface refinement. *arXiv preprint arXiv:2303.02091*, 2023. 2

[22] Linning Xu, Vasu Agrawal, William Laney, Tony Garcia, Aayush Bansal, Changil Kim, Samuel Rota Bulò, Lorenzo Porzi, Peter Kontschieder, Aljaž Božič, Dahua Lin, Michael Zollhöfer, and Christian Richardt. VR-NeRF: High-fidelity virtualized walkable spaces. In *SIGGRAPH Asia Conference Proceedings*, 2023. 1, 4

[23] Lior Yariv, Peter Hedman, Christian Reiser, Dor Verbin, Pratul P Srinivasan, Richard Szeliski, Jonathan T Barron, and Ben Mildenhall. Bakedsdf: Meshing neural sdfs for real-time view synthesis. *Siggraph*, 2023. 1, 2, 4, 5

[24] Zehao Yu, Anpei Chen, Bozidar Antic, Songyou Peng, Apratim Bhattacharyya, Michael Niemeyer, Siyu Tang, Torsten Sattler, and Andreas Geiger. Sdfstudio: A unified framework for surface reconstruction, 2022. 5