NViSII: A Scriptable Tool for Photorealistic Image Generation

Nathan Morrical^{1,2}, Jonathan Tremblay¹, Yunzhi Lin^{1,3}, Stephen Tyree¹, Stan Birchfield¹, Valerio Pascucci², Ingo Wald¹ ¹NVIDIA, ²University of Utah, ³Georgia Institute of Technology



Figure 1: Our scriptable tool leverages hardware-accelerated path tracing to generate photorealistic images. Annotations are shown along the bottom row, from left to right: image with no motion blur, depth, surface normals, segmentation labels, texture coordinates, and optical flow.

ABSTRACT

We present a Python-based renderer built on NVIDIA's OptiX ray tracing engine and the OptiX AI denoiser, designed to generate high-quality synthetic images for research in computer vision and deep learning. Our tool enables the description and manipulation of complex dynamic 3D scenes containing object meshes, materials, textures, lighting, volumetric data (*e.g.*, smoke), and backgrounds. Metadata, such as 2D/3D bounding boxes, segmentation masks, depth maps, normal maps, material properties, and optical flow vectors, can also be generated. In this work, we discuss design goals, architecture, and performance. We demonstrate the use of data generated by path tracing for training an object detector and pose estimator, showing improved performance in sim-to-real transfer in situations that are difficult for traditional raster-based renderers. We offer this tool as an easyto-use, performant, high-quality renderer for advancing research in synthetic data generation and deep learning.

1 INTRODUCTION

For many computer vision tasks, it is challenging or even impossible to obtain labeled real-world images for use in training deep neural networks. For example, labeled ground truth of rare events like vehicle collisions, or of dense high-dimensional values like optical flow vectors, are not easy to obtain. To overcome these limitations, researchers have explored synthetic data for a variety of applications: object pose estimation (Tremblay et al., 2018b; Denninger et al., 2019), depth estimation of transparent objects (Sajjan et al., 2019), scene segmentation (Handa et al., 2015; Roberts & Paczan, 2020), optical flow (Dosovitskiy et al., 2015), autonomous vehicles (Ros et al., 2016;

Table 1: Related work compared to our proposed system. ' \checkmark ' refers to fully supported, ' \checkmark ' not supported, and '-' partially supported or limited feature.

	AI2-Thor	iGibson	NDDS	Unity3D	Sapien	BlenderProc	Ours
path tracing	×	×	X	1	 Image: A set of the set of the	 Image: A set of the set of the	 Image: A start of the start of
easy installation (pip)	1	1	X	_	-	_	1
cross platform	1	1	X	1	×	1	1
Python API	1	1	X	×	-	_	1
headless rendering	1	1	×	×	1	1	1

Prakash et al., 2019), robotic control (Tobin et al., 2017), path planning and reasoning in 3D scenes (Kolve et al., 2017; Xia et al., 2020), and so forth.

To generate such datasets, a variety of tools have been developed, including AI2-Thor (Kolve et al., 2017), iGibson (Xia et al., 2020), NDDS (To et al., 2018), Unity3D (Crespi et al., 2020), Sapien (Xiang et al., 2020), and BlenderProc (Denninger et al., 2019), as shown in Table 1. Although AI2-Thor and iGibson come with powerful Python APIs, both of them are based on classic raster scanning. On the other hand, the more recent tools capable of photorealistic imagery via path tracing (such as Unity3D, Sapien, and BlenderProc) come with limited scriptable interfaces, such as runtime scene editing, limited material and light expression, etc. To overcome this limitation, in this work we introduce NVISII: the NVIDIA Scene Imaging Interface, a scriptable tool for path-traced image generation. With our tool, users can construct and manipulate complex dynamic 3D scenes containing object meshes, materials, textures, lighting, volumetric data (e.g., smoke), and cameras—all potentially randomized—using only Python code. This design choice ensures that users have full control, allowing scenes to be permuted on-the-fly according to the needs of the problem being considered. By leveraging path tracing, photorealistic images are produced, including physically-based materials, lighting, and camera effects. All of this can be achieved while maintaining interactive frame rates via NVIDIA's OptiX library and hardware accelerated ray tracing. Our tool is easily accessible via the pip packaging system.¹

We offer this tool to the community to enable researchers to procedurally manage arbitrarily complex scenes for photorealistic synthetic image generation. Our contributions are as follows: 1) An open source, Python-enabled ray tracer built on NVIDIA's OptiX, with a C++/CUDA backend, to advance sim-to-real and related research. 2) A demonstration of the tool's capabilities in generating synthetic images by training a DOPE pose estimator network (Tremblay et al., 2018c) and a 2D object detector (Zhou et al., 2019) for application to real images. 3) An investigation into how physically-based material definitions can increase a pose estimator's accuracy for objects containing specular materials.

2 PATH TRACER WITH PYTHON INTERFACE

We develop the tool with three goals in mind: 1) ease of installation, 2) speed of development, and 3) rendering capabilities. For ease of installation, we ensure that the solution is accessible, open source, and cross platform (Linux and Windows), with pre-compiled binaries that are distributed using a package manager (thus obviating the need to build the tool from source). For speed of development, the solution provides a comprehensive and interactive Python API for procedural scene generation and domain randomization. The tool does not require embedded interpreters, and it is well-documented with examples. Finally, we want a solution that supports advanced rendering capabilities, such as multi-GPU enabled ray tracing, physically-based materials, physically-based light controls, accurate camera models (including defocus blur and motion blur), native headless rendering, and exporting various metadata (e.g. segmentation, motion vectors, optical flow, depth, surface normals, and albedo). See Figure 2 for example renders.

2.1 TOOL ARCHITECTURE

Our rendering tool follows a data-driven entity component system (ECS) design. Such ECS designs are commonly used for game engines and 3D design suites, as they facilitate intuitive and flexible scene descriptions by avoiding complex multiple-inheritance hierarchies required by object-oriented

¹pip install nvisii



Figure 2: Example renders. TOP: bistro scene¹ (zoom for details), indoor scene², texture used as light. BOTTOM: smoke with 3D volume, reflective metal, domain randomization.



Figure 4: Different material configurations. From left to right: material with roughness = 1.0 (velvet), roughness = 0.0 (plastic), transmission = 1.0 (glass), metallic = 1.0 (metal).

designs. This flat design allows for simpler procedural generation of scenes when compared to object-oriented designs.

After initialization, the scene entities and components are created at runtime, as opposed to an offline scene description format like many prior solutions. See Figure 3 for a simple example. By enabling runtime scene edits, our solution more effectively leverages modern rendering performance capabilities during synthetic image generation. These components can be created using either the C++ or Python API, and as these components are created and connected together using entities, the out-of-date components are asynchronously uploaded to a collection of GPUs in a data-replicated pattern for interactive rendering.

Any given entity can be attached to any combination of the following component types: transform, mesh, material, light, camera, and volume. Transforms refer to the entity's SE(3) behaviour, *i.e.*, 3D translation and rotation, which causes motion blur and so forth. *Meshes* describe the 3D points and triangles to be rendered. *Materials* refer to the entity's physically-based rendering material definition. *Lights* define the energy emitted by the entity, which requires a mesh since point lights are physically impossible. *Cameras* refer to scene views, along with focal length and aperture. *Volumes* can be used in place of meshes to represent voxel data like smoke. In addition to the aforementioned components, textures can be used to drive any material properties, and when a texture is used in concert with a light, the texture RGB value defines the color of the light. Once connected to an entity, components immediately affect the appearance of the scene. These components can also be reused across entities as a basic form of instancing. Swapping out and modifying components additionally serves as an effective way to randomize the scene.

¹https://developer.nvidia.com/orca/amazon-lumberyard-bistro

²https://blendswap.com/blend/12584



Figure 5: Motion blur and defocus blur.



Figure 6: Still image (left) and motion-blurred image (right) of a robot reaching a specific goal (green sphere).



Figure 7: Examples of rendering 3D volumes.

2.2 RENDERING CAPABILITIES

The material definition follows the Principled BSDF (Burley, 2015). This model consists of sixteen "principled" parameters, where all combinations result in physically-plausible, or at least well-behaved, results. Among these are base color, metallic, transmission, and roughness parameters. All parameters can be driven using either constant scalar values or texture components. With these parameters, scenes can represent smooth and rough plastics, metals, and dielectric materials (*e.g.*, glass), see Figure 4. In addition, material components also accept an optional "normal map" texture to allow for more details on surface geometry.

Mesh components can be constructed either using a supported mesh format or alternatively by passing a list of vertices and corresponding pervertex values. When meshes are created, we asynchronously construct a bottom level acceleration structure (BLAS) for later ray traversal. Instead of a mesh, you can also use volumetric



Figure 3: A minimal Python script example.

data, such as *nanovdb* files or a 3D numpy array of density values, see Figure 7. When attaching a camera and a transform component together via an entity, users can select that entity to use as their view into the scene. When attaching a mesh, a material, and a transform component together via an entity, our renderer allocates an instance of the corresponding BLAS into a common instance level acceleration structure (IAS), at which point an object will appear in the scene. When in use, camera components can be used to induce defocus blur, and transform components can be used to induce motion blur, both of which are demonstrated in Figure 5 and the latter in Figure 6.

Direct and indirect lighting is computed using NVIDIA's OptiX ray tracing framework (Parker et al., 2010) through a standard path tracer with next event estimation. Multi-GPU support is enabled using the OptiX Wrapper Library (OWL) developed by Wald et al. (2020). These frameworks enable real-world effects like reflections, refractions, and global illumination, while simultaneously benefiting from hardware-accelerated ray tracing. Finally, the OptiX denoiser is used to more quickly obtain clean images for use in training.

2.3 DEVELOPMENT

An early part of development went into minimizing technical debt by automating several recurring maintenance processes like Python binding and cross-platform binary generation. To filter out potential OS specific compilation errors, we use continuous integration (CI) through Github actions, which at the time of writing are free for open source projects. We also use these CI machines as a source for binary artifacts that can be installed directly by an end user. For use in C++ projects, end users must still link the final binaries into their projects; however, for Python users we are able to expose the entirety of the C++ API during runtime through Python bindings that are automatically generated using SWIG. We also use SWIG to translate C++ Doxygen comments into Python Doc-Strings. Binary artifacts and Python bindings built by the CI service are then uploaded to the pip package manager on tagged releases for improved accessibility. Although these design decisions are not necessarily novel, we believe these decisions make for a library that is more accessible to potential end users within the robotics and computer vision communities.

2.4 EXTERNAL PACKAGING

Many components in our renderer expose APIs to make integration with external packages easier. Wherever possible, the API supports types native to the language. For example, mesh and texture components can be constructed either from a file on disk, or alternatively from raw Python lists / NumPy arrays of vertices and pixels generated at runtime by external packages. We have found

this to be quite useful for domain randomization, where textures can be constructed randomly using third-party pattern generator packages. For physics systems like PyBullet, this is also useful for constructing colliders from the vertices and indices of our mesh component.

The API leverages existing language features for improved code legibility (for example by including support for keyword arguments in Python). Camera components return either intrinsic projection matrices or affine 4×4 projection matrices, as one might be more convenient than the other depending on the domain expert using the tool. Finally, to help new users get started, we include comprehensive documentation through C++ Doxygen that is automatically translated to Python DocStrings during binding generation. Sphinx is then used to expose this documentation in an accessible website format. In the repository, we also include an extensive list of examples for cross-referencing.

3 EVALUATION

In this section we explore the use of our tool to generate synthetic data for training neural networks. We first explore the problem of 2D detections and then explore how different material propreties can be used to enhance pose estimation of metallic objects.

3.1 HOPE OBJECT DETECTION

We trained CenterNet (Duan et al., 2019) to perform 2D detection of known objects from the HOPE dataset (Tyree et al., 2019) consisting of 28 toy grocery items with associated 3D models. Since the dataset does not contain any training images, it is an ideal candidate for sim-to-real transfer. The test set contains 238 unique images with 914 unique object annotations.

Using NViSII we generated 3 different dataset types: DOME, MESH, and FAT.

DOME. Tobin et al. (2017) demonstrated that *domain randomization* can be used to train a model fully on synthetic data to be used on real data. Tremblay et al. (2018a) extended the framework for 2D car detection. In this work we generate similar images, see Figure 8 first row for examples. In our tool, we leverage its capacity to illuminate the scene using an HDR dome texture (as in Figure 2 top-right, with the texture spread across the upper hemisphere), which offers a natural looking light that mimics interreflections. Similar to Tremblay et al. (2018a) we randomize the object poses within a volume and add flying distractors. See Figure 8 first row.

MESH. Hinterstoisser et al. (2019) introduced the concept that using other 3D meshes as background could potentially lead to better sim-to-real. As such, we use our tool capacity to generate random 3D meshes and applied random material to these, we used around a 1000 moving meshes as background as seen in Figure 8 second row. For illumination we used 2 to 6 random lights (random color and intensity) placed behind the camera to generate random light context. See Figure 8 second row.

FAT. Tremblay et al. (2018b) introduced a dataset where objects were allowed to freely fall into a complex 3D scene. Following on that work Tremblay et al. (2018c) proposed to mix falling dataset and domain randomization dataset to solve the sim-to-real problem. As such we integrated our tool with PyBullet (Coumans & Bai, 2016–2019) to let objects fall onto a simple plane, see Figure 8 third row. We simplify the prior work were we use a dome texture to create photorealistic backgrounds and create natural lights, and we then simply apply a random floor texture onto the plane to simulate the surface. See Figure 8 third row. These renders are also similar to images generated by BlenderProc (Denninger et al., 2019).

We use a collection of 6k rendered images to train the object detector, and we compare using a single type of dataset with a mixture of datasets, while keeping the training size constant. We trained CenterNet² (Duan et al., 2019) from scratch, using SGD optimizer with learning rate of 0.02 reduced by a factor of 10 at 81k and 108k steps, and a batch size of 128. The network was trained for 126k iterations on eight NVIDIA V100s for about 12 hours. Table 2 shows the results, specifically that the DOME dataset performs better than MESH. These results also confirm the observation from Tremblay et al. (2018c) that mixing different dataset generation methods outperfoms using a single

²We used the repo at github.com/FateScript/CenterNet-better



DOME (Tremblay et al., 2018a)



MESH (Hinterstoisser et al., 2019)



FAT (Tremblay et al., 2018b)

Figure 8: Training images for pose estimation and object detection. For each row, images were generated using a procedure similar to that of the reference shown.



Figure 9: Detections on real HOPE images (Tyree et al., 2019) using CenterNet (Duan et al., 2019) trained on synthetic data generated by NViSII. On these images, 68.2% of objects were detected, with no false positives.

Methods	AP	AP50	AP75
DOME + MESH + FAT	47.8	70.0	47.7
DOME + MESH	46.2	70.1	47.2
DOME	44.2	66.0	45.4
MESH	36.1	59.4	36.2
FAT	33.7	49.2	34.9

Table 2: Sim-to-real 2D bounding box detection experiment on HOPE dataset.

one. Detections from the better model trained on using all the presented datasets can be seen on Figure 9.

3.2 METALLIC MATERIAL OBJECT POSE ESTIMATOR.

In the work by Tremblay et al. (2018c), the authors hypothesized that the somewhat disappointing pose estimation accuracy of the YCB potted meat model was caused by an inaccurate synthetic representation of the object's real-world material properties. This model is metallic on the top and bottom, and is wrapped with a plastic-like label. Additionally, the original 3D model provided by YCB (Calli et al., 2015) has lighting conditions (highlights) baked into the model's textures. This results in an unrealistic appearance, especially under different lighting conditions. To test this hypothesis, we modified the original base color texture of the model to remove all baked highlights. Next, we manually segmented the different material properties of the object (specifically the metallic vs. non-metallic regions) to create a more physically-accurate material description. Figure 10 compares the physically-correct material vs. the original material with baked lighting. These images demonstrate how the metallic texture causes a highlight from the lights along the reflection direction under certain view angles, whereas the original texture is flat and contains unnatural highlights that do not match the surrounding synthetic scene. Using the approach proposed by Hinterstoisser et al. (2019) we generated 60k domain randomized synthetic images using path raytracer for training with random meshes and random material as background. Two to six lights were placed randomly behind the camera with randomized position, temperature, and intensity. We compare our results with the DOPE weights available online on the YCB-video (Xiang et al., 2017) and we report the area under the ADD threshold curve. The original method scores 0.314 whereas our proposed solution gets 0.462. We see an overall performance improvement in our trained pose estimator over the same network trained with the original NDDS images.

4 RELATED WORK

Given the high demand for large annotated data sets for use in deep learning, we have seen the rise in both synthetic data sets (Tremblay et al., 2018; Ros et al., 2016; Mayer et al., 2016; Handa et al., 2015; Zhang et al., 2016; Richter et al., 2016; Gaidon et al., 2016) as well as a rise in tooling for generating these data sets (Kolve et al., 2017; To et al., 2018; Crespi et al., 2020; Denninger et al., 2019; Xiang et al., 2020). However, through our investigations, we have found that each of these tools come with their own set of tradeoffs when it comes to installation, rendering capabilities, and ease of development for personal solutions.

In the field of robotics, physical simulators like PyBullet (Coumans & Bai, 2016–2019), MuJoCo (Todorov et al., 2012), and V-REP (Rohmer et al., 2013) all have rudimentary renderers that have been used to generate synthetic data for robotics and computer vision applications. For example, Tobin *et al.* (Tobin et al., 2017) introduced domain randomization, the technique of unrealistically randomizing 3D content to bridge the reality gap, and used Mujoco's renderer to generate synthetic images. Similarly, domain randomization has been used to train a robot to pick up a cube by James *et al.* (James et al., 2017), who used V-REP for synthetic image generation. PyBullet has also been used by Xie et al. (2020) to pre-train an RGBD based object segmentation method. These physics simulators typically come with renderers that are accessible and flexible, but they lack physically based light transport simulation, material definitions, and camera effects, making them quite limited when it comes to photorealistic rendering capabilities (*e.g.*, no support for dielectric materials like glass, no specular materials like metals and mirrors, and no advanced camera effects like depth of field or motion blur, etc).



Figure 10: TOP-LEFT: Original YCB texture with baked-in highlights. TOP-RIGHT: Corrected flat texture and properly associated metallic material used by our tool. BOTTOM-LEFT: DR image rendered by our tool. BOTTOM-RIGHT: Pose prediction from our trained model. (Note that the model trained on the original YCB texture does not detect the meat can in this image.)

In order to access more advanced rendering capabilities, researchers have also explored using video game based renderers, such as Unreal Engine (UE4) or Unity 3D. For example, NDDS was introduced by To *et al.* as an effort to simplify the domain randomization capabilities in UE4 (To et al., 2018). Similarly, Qiu and Yuille added a plugin to UE4 to export meta data (Qiu & Yuille, 2016). Recently, Unity 3D introduced an official computer vision package to export metadata and create simple scenes for computer vision (Crespi et al., 2020). Through these game engines, researchers have been able to explore a variety of training scenarios (Long & Yao, 2020; Kolve et al., 2017; Juliani et al., 2018). Although game engines offer state-of-the-art in raster based rendering, they prioritize frame rate over image quality, and offer limited capabilities when it comes to photo-realistic material properties and light transport simulation compared to ray tracing based solutions. Addition-ally, since these applications are primarily designed for game development, they can be daunting to install, learn, modify, and script for use in synthetic data generation.

Ray tracing based synthetic data generation has also been explored, as ray-tracing techniques more closely model the behavior of light, and can produce photorealistic images. This exploration has mainly been done through the use of the Cycles renderer included in Blender, as Blender is an open source and easy to install solution for 3D modeling and animation. For example, Iraci (2013) used Cycles to render transparent objects in order to train a computer vision network to detect the point cloud (Sajjan et al., 2019). Blender and Cycles have also been used to generate data sets of head poses (Gu et al., 2017), eye poses (Wood et al., 2015), different kind of objects (Ron & Elbaz, 2020), robot poses (Heindl et al., 2019), and so on. In an attempt to more easily generate synthetic images, Denninger et al. (2019) introduced an extension to Blender that render falling objects onto a plane with randomized camera poses. These tailored solutions tend to be inflexible when users want to explore different scene configurations. Although Blender has proven to be a useful tool for synthetic image generation, as Blender's Python API requires the use of an included, embedded interpreter, it cannot easily be scripted externally. As a result, solutions which use Blender for synthetic data generation tend to be constrained, and do not allow the user to easily edit and randomize the currently loaded scene.

Other tools, like SAPIEN (Xiang et al., 2020), extend the Python interpreter for improved flexibility. SAPIEN optionally leverages NVIDIA's OptiX for high-quality ray tracing of articulated objects and robotics scenes; however, this functionality requires building SAPIEN from source, making their framework difficult to install. SAPIEN's material model also cannot easily be randomized and cannot easily be controlled to produce non-Lambertian materials like glass and metallics.

5 CONCLUSION

We have presented an open-source Python-enabled ray tracer built on NVIDIA's OptiX with a C++/CUDA backend to advance sim-to-real and related research. The tool's design philosophy is easy install, accessible hardware requirements, enable scenes to be created through scripting, and rendering of photorealistic images. We release NViSII in the hope that it will be helpful to the community.

REFERENCES

- Brent Burley. Extending the Disney BRDF to a BSDF with integrated subsurface scattering. *Physically Based Shading in Theory and Practice SIGGRAPH Course*, 2015.
- B. Calli, A. Walsman, A. Singh, S. Srinivasa, P. Abbeel, and A. M. Dollar. The YCB object and model set: Towards common benchmarks for manipulation research. In *Intl. Conf. on Advanced Robotics (ICAR)*, 2015.
- Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2019.
- Adam Crespi, Cesar Romero, Srinivas Annambhotla, Jonathan Hogins, and Alex Thaman. Unity perception, 2020. https://blogs.unity3d.com/2020/06/10/.
- Maximilian Denninger, Martin Sundermeyer, Dominik Winkelbauer, Youssef Zidan, Dmitry Olefir, Mohamad Elbadrawy, Ahsan Lodhi, and Harinandan Katam. Blenderproc. *arXiv preprint arXiv:1911.01911*, 2019.
- A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazırbaş, V. Golkov, P. Smagt, D. Cremers, and T. Brox. FlowNet: Learning optical flow with convolutional networks. In *ICCV*, 2015.
- Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *Proceedings of the IEEE/CVF International Conference* on Computer Vision, pp. 6569–6578, 2019.
- Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual worlds as proxy for multiobject tracking analysis. In *CVPR*, 2016.
- Jinwei Gu, Xiaodong Yang, Shalini De Mello, and Jan Kautz. Dynamic facial analysis: From Bayesian filtering to recurrent neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.
- A. Handa, V. Pătrăucean, V. Badrinarayanan, S. Stent, and R. Cipolla. SceneNet: Understanding real world indoor scenes with synthetic data. In *arXiv* 1511.07041, 2015.
- Christoph Heindl, Sebastian Zambal, and Josef Scharinger. Learning to predict robot keypoints using artificially generated images. In 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1536–1539, 2019.
- Stefan Hinterstoisser, Olivier Pauly, Hauke Heibel, Martina Marek, and Martin Bokeloh. An annotation saved is an annotation earned: Using fully synthetic training for object instance detection. *arXiv preprint arXiv:1902.09967*, 2019.
- Bernardo Iraci. Blender cycles: lighting and rendering cookbook. Packt Publishing Ltd, 2013.
- Stephen James, Andrew J Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task. In *arXiv:1707.02267*, 2017.

- Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627, 2018.
- Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An interactive 3D environment for visual AI. *arXiv preprint arXiv:1712.05474*, 2017.
- Shangbang Long and Cong Yao. Unrealtext: Synthesizing realistic scene text images from the unreal world. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 5488–5497, 2020.
- Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *CVPR*, 2016.
- Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, and Austin Robison. OptiX: A General Purpose Ray Tracing Engine. ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH), 2010.
- Aayush Prakash, Shaad Boochoon, Mark Brophy, David Acuna, Eric Cameracci, Gavriel State, Omer Shapira, and Stan Birchfield. Structured domain randomization: Bridging the reality gap by context-aware synthetic data. In *ICRA*, 2019.
- W. Qiu and A. Yuille. UnrealCV: Connecting computer vision to unreal engine. In *arXiv* 1609.01326, 2016.
- Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision (ECCV)*, pp. 102–118, 2016.
- Mike Roberts and Nathan Paczan. Hypersim: A photorealistic synthetic dataset for holistic indoor scene understanding, 2020. URL https://arxiv.org/pdf/2011.02523.pdf.
- Eric Rohmer, Surya PN Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1321–1326, 2013.
- Roey Ron and Gil Elbaz. EXPO-HD: Exact object perception using high distraction synthetic data. In *arXiv:2007.14354*, 2020.
- G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. Lopez. The SYNTHIA dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *CVPR*, 2016.
- Shreeyak S Sajjan, Matthew Moore, Mike Pan, Ganesh Nagaraja, Johnny Lee, Andy Zeng, and Shuran Song. Cleargrasp: 3d shape estimation of transparent objects for manipulation. *arXiv* preprint arXiv:1910.02550, 2019.
- Thang To, Jonathan Tremblay, Duncan McKay, Yukie Yamaguchi, Kirby Leung, Adrian Balanon, Jia Cheng, and Stan Birchfield. NDDS: NVIDIA deep learning dataset synthesizer, 2018. https://github.com/NVIDIA/Dataset_Synthesizer.
- Joshua Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IROS*, 2017.
- Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IROS*, pp. 5026–5033, 2012.
- Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *CVPR Workshop on Autonomous Driving (WAD)*, 2018a.

- Jonathan Tremblay, Thang To, and Stan Birchfield. Falling things: A synthetic dataset for 3D object detection and pose estimation. In CVPR Workshop on Real World Challenges and New Benchmarks for Deep Learning in Robotic Vision, 2018b.
- Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. In *CoRL*, 2018c.
- Stephen Tyree, Jonathan Tremblay, Thang To, Jia Cheng, Terry Mossier, and Stan Birchfield. 6-DoF pose estimation of household objects for robotic manipulation: an accessible dataset and benchmark. In *ICCV Workshop on Recovering 6D Object Pose*, 2019.
- I. Wald, N. Morrical, and E. Haines. OWL The OptiX 7 Wrapper Library, 2020. URL https: //github.com/owl-project/owl.
- Erroll Wood, Tadas Baltrusaitis, Xucong Zhang, Yusuke Sugano, Peter Robinson, and Andreas Bulling. Rendering of eyes for eye-shape registration and gaze estimation. In *Proc. of the IEEE International Conference on Computer Vision (ICCV)*, 2015.
- Fei Xia, William B Shen, Chengshu Li, Priya Kasimbeg, Micael Edmond Tchapmi, Alexander Toshev, Roberto Martín-Martín, and Silvio Savarese. Interactive gibson benchmark: A benchmark for interactive navigation in cluttered environments. *IEEE Robotics and Automation Letters*, 5(2): 713–720, 2020.
- Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. SAPIEN: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11097–11107, 2020.
- Y. Xiang, T. Schmidt, V. Narayanan, and D. Fox. PoseCNN: A convolutional neural network for 6D object pose estimation in cluttered scenes. In *arXiv* 1711.00199, 2017.
- Christopher Xie, Yu Xiang, Arsalan Mousavian, and Dieter Fox. The best of both modes: Separately leveraging RGB and depth for unseen object instance segmentation. In *Conference on Robot Learning (CoRL)*, pp. 1369–1378, 2020.
- Y. Zhang, W. Qiu, Q. Chen, X. Hu, and A. Yuille. UnrealStereo: A synthetic dataset for analyzing stereo vision. In *arXiv:1612.04647*, 2016.
- Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. In *arXiv preprint* arXiv:1904.07850, 2019.