ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bubeníček**	Jméno: **Tomáš**	Osobní číslo: **458050**	

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Konfigurovatelný nástroj pro tvorbu syntetických dat

Název diplomové práce anglicky:

Configurable tool for synthetic data creation

Pokyny pro vypracování:

Zmapujte existující volně dostupné sady syntetických dat určených pro strojové učení a nástroje pro jejich vytváření.
Navrhněte a implementujte aplikaci umožňující vytváření syntetických dat pro různé aplikační scénaře s využitím různorodých 3D modelů a jejich textur. Aplikace umožní exportovat rozšířená obrazová data včetně normál, paměti hloubky, sémantické segmentace, optického toku a pohybové masky. Podporujte také export neobrazových metadat jako například parametry kamery a obálky objektů. Vytvořte základní uživatelskou příručku k vytvořené aplikaci srozumitelnou uživatelům nepocházejícím z komunity počítačové grafiky.
S použitím vytvořeného nástroje vygenerujte nejméně tři sady syntetických dat přímo použitelných pro různé metody strojového učení. Vytvořte dokumentaci k vygenerovaným datům a srovnejte je s jinými existujícími datovými sadami.

Seznam doporučené literatury:

[1] Jason Gregory. Game Engine Architecture (3rd edition). CRC Press, 2018.
[2] Tomas Akenine-Moller et al. Real-Time Rendering (4th edition). CRC Press, 2018.
[3] Dosovitskiy, A.; Fischer, P.; Ilg, E.; Häusser, P.; Hazırbas, C.; Golkov, V.; van der Smagt, P.; Cremers, D. & Brox, T. FlowNet: Learning Optical Flow with Convolutional Networks .
[4] Mayer, N.; Ilg, E.; Hausser, P.; Fischer, P.; Cremers, D.; Dosovitskiy, A. & Brox, T. A Large: Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation
[5] Butler, D. J. and Wulff, J. and Stanley, G. B. and Black, M. J.: A naturalistic open source movie for optical flow evaluation, ECCV 2012.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **11.02.2020** Termín odevzdání diplomové práce: ____________

Platnost zadání diplomové práce: **30.09.2021**

	[illegible]	[illegible]
doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání	Podpis studenta

Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Configurable Utility for Synthetic Dataset Creation

Konfigurovatelný nástroj pro tvorbu syntetických dat

Tomáš Bubeníček

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Field of study: Computer Graphics
August 2020

Acknowledgements

I would like to thank to my family for support, the Toyota Research Lab members for aid during the development of the practical part of this project, and also my supervisor, Jiří Bittner, for his guidance not only during development but during writing this thesis.

Declaration

I declare that I have completed the work on my own and that I cited all used literature and sources.

Prague, August 13, 2020.

—

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 13. srpna 2020.

Abstract

When evaluating existing computer vision algorithms or training new machine learning algorithms, large datasets of various images with ground truth, the ideal known solution to the currently solved problem, need to be acquired. We review existing real-life datasets containing ground truth, which are used in computer vision, and explore how they were acquired. We then recount different synthetic datasets, and survey the different ways such data can be calculated. We propose a tool to simplify generation of such data, and implement such tool as an extension of the Unity editor. Our implementation is able to use textured 3D models to generate image sequences with additional labeling, such as surface normals, depth map, object segmentation, optical flow, motion segmentation among others. We use the tool to create a set of three example datasets.

Keywords: Synthetic dataset generation, Ground truth computation, Optical flow, Game engines in Machine learning

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Karlovo Namesti 13,
121 35 Praha 2,
Czech Republic

Abstrakt

Při vyhodnocování funkcionality algoritmů z oboru počítačového vidění či při trénování nových algoritmů za pomocí metody strojového učení, velké množství algoritmů obsahujících dodatečné ground truth výstupy, které reprezentují ideální výsledek daných algoritmů. V této práci jsme analyzovali existující datatové sady určené pro počítačové vidění. Zkoumali jsme, jak jsou taková data získávána jak ve skutečném světě, tak pomocí simulací. Navrhli jsme nástroj na zjednodušení tvorby syntetických dat tohoto typu a naimplementovali jsme ho jako rozšíření editoru Unity. Naše implementace je schopná využít texturované 3D modely a na jejich základě generovat mimo jiné informaci o povrchových normálách, hloubkových mapách, sémantické segmentace, optického toku a pohybových maskách. S využitím našeho nástroje jsme vygenerovali tři ukázkové datové sady.

Klíčová slova: Generování syntetických dat, Výpočet ground truth, Optický tok, Herní enginy ve strojovém učení

Překlad názvu: Konfigurovatelný nástroj pro tvorbu syntetických dat

Contents

Chapter 1

Introduction

One of the open problems of computer vision and image processing is generating or approximating augmented image data based on images acquired by a regular RGB camera. Augmented image data such as semantic segmentation help machines separate parts in factory lines, using optical flow data for video compression reduces redundancy, and depth and normals data are useful for approximating a 3D scene topology. Acquiring these augmented image data is often very difficult, cost-prohibitive, or sometimes even impossible. Many different algorithms exist with different levels of success. The modern state of the art research currently focuses on using machine learning and neural networks for generating the data from camera images. Figure 1.1 shows the result of one such algorithm, SegNet[1], which is able to augment the image by segmenting it into different sections based on different categories of objects.

Both for evaluating a given algorithm and training supervised machine learning algorithms, ground truth data are necessary. When evaluating, we can compare the output of the algorithm with the expected ground truth output based on the gold standard tests. When training supervised machine learning algorithms, the algorithm is tweaked by using training example pairs of possible inputs and expected outputs. For both of these uses of ground truth data we require large datasets, which are often hard to obtain.

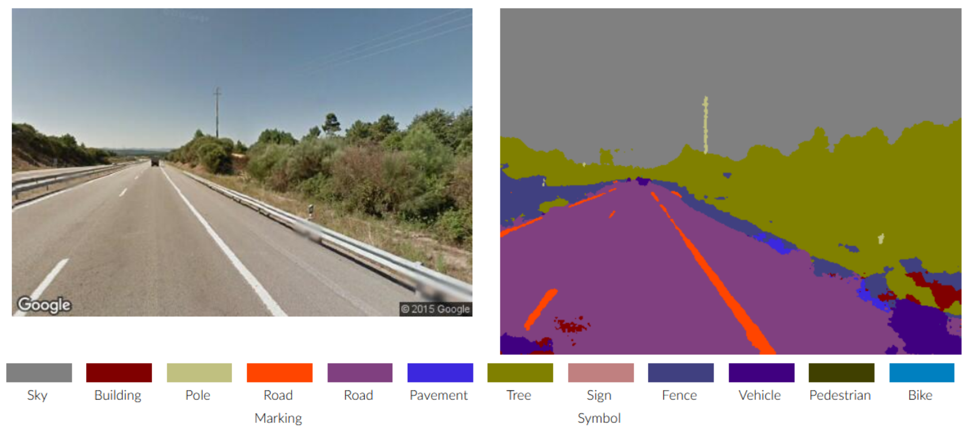


Figure 1.1: An example of the SegNet[1] algorithm, which generates semantic segmentation maps from a single RGB camera image.

For some uses, such as object categorization, broad, often human-labeled datasets are already publicly available. However, for some augmented image data (such as optical flow), the real measured ground truth is often sparse or not measurable by conventional sensors in general. For this very reason, synthetic datasets which are acquired purely from a simulated scene are also used.

1.1 Goals

We have several goals in this project. We wish to explore the currently available datasets for machine learning. Then we want to identify and describe the ground truth data for computer vision which the datasets contain. After we have full grasp on which type of data are necessary for scene understanding, our goal is to describe how we can generate such data using methods based on computer graphics. Our biggest goal is then to design and implement a tool which simplifies the generation of datasets for computer vision, as we believe such tool could be of use to the general vision research community. Our final goal is to generate a set of datasets which show the functionality of the tool itself.

1.2 Thesis Structure

In chapter 2, we discuss different already existing datasets used in the computer vision field. We mention both real-life and synthetic datasets, talk about tools which can be used to generate synthetic datasets, and explain the different ways how specific ground truth outputs can be represented.

The next chapte 3 explains how the data contained in synthetic datasets are generated. We discuss simulating the camera and the scene. We talk about how ground truth data for such synthetic scene are calculated.

In chapter 4, we talk about the broader design choices which were made when designing the tool to generate datasets containing ground truth. We select on which framework the tool is built and the structure of the generator itself.

Chapter 5 presents the implementation details of the tool. We talk about how each ground truth output is calculated in code and touch on framework-specific changes which were made in the calculations discussed in chapter 3.

Chapter 6 describes the usage of the tool, some of the datasets we generated and the tool's performance during generation. We also talk about the issues with the current implementation of the tool itself.

We conclude the thesis with chapter 7, where we compare the completed tasks with the assignment. We also talk about the future work which can be done on the described implementation and in the field.

Chapter 2

Related Work

This chapter presents already existing datasets and utilities used to create such datasets. In later section, we also talk about the possible representation of ground truth data for machine learning. We understand ground truth to be different information provided by direct observation of real-life or a simulation and the ideal expected result of algorithms for computer vision.

2.1 Datasets for Machine Learning

Data used for object segmentation are probably the biggest and most common datasets currently available. For example, the *COCO (Common Objects in Context)* [23] dataset contains over 200 thousand human-labeled real-life images and is useful for training networks to recognize objects located in photos. An example of a labeled image from such dataset can be seen in figure 2.1.

Real-life datasets containing depth are less common, but still readily obtainable, and can be useful for scene reconstruction. A combination of LIDAR and camera mounted on a car is usually the source of these datasets. This

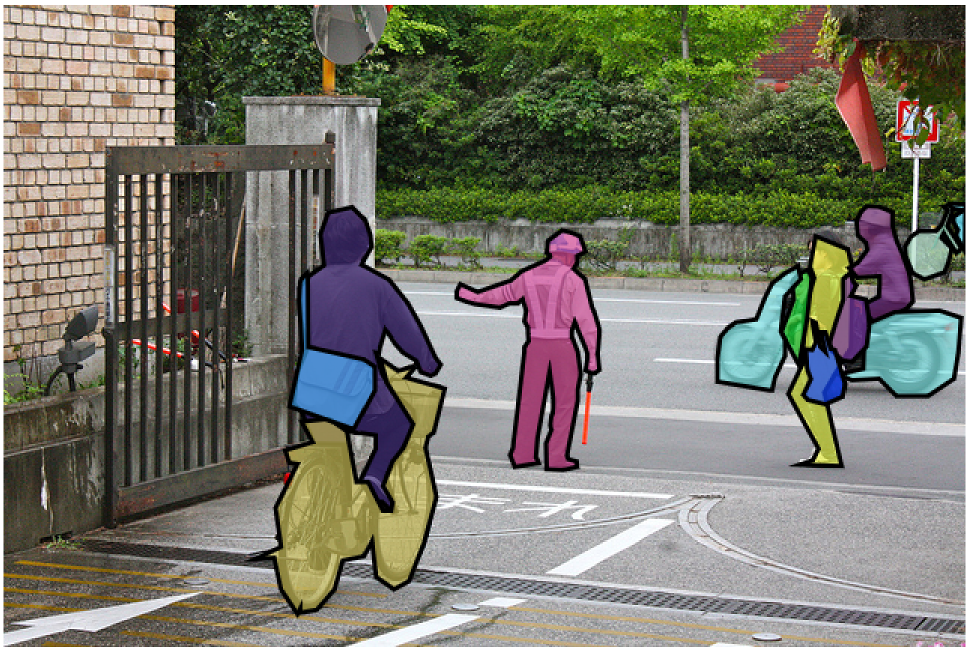
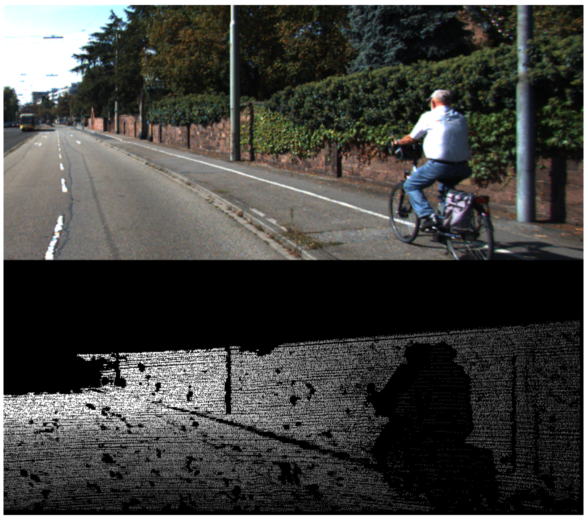
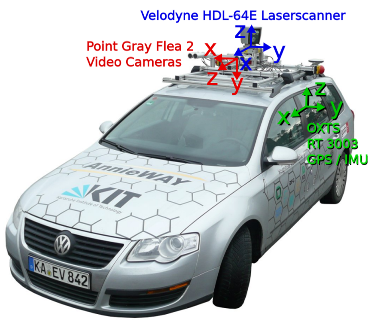


Figure 2.1: An example from the COCO dataset [23].



(a) : The top image shows the camera view and the bottom image contains the depth information acquired using LIDAR. Note how the depth information is sparse in comparision to the camera image.



(b) : The car, equipped with cameras, an inertial measurement unit and a LIDAR scanner, was used to capture the dataset.

Figure 2.2: An image from the KITTI datasets [15][26] and the car used to capture it.

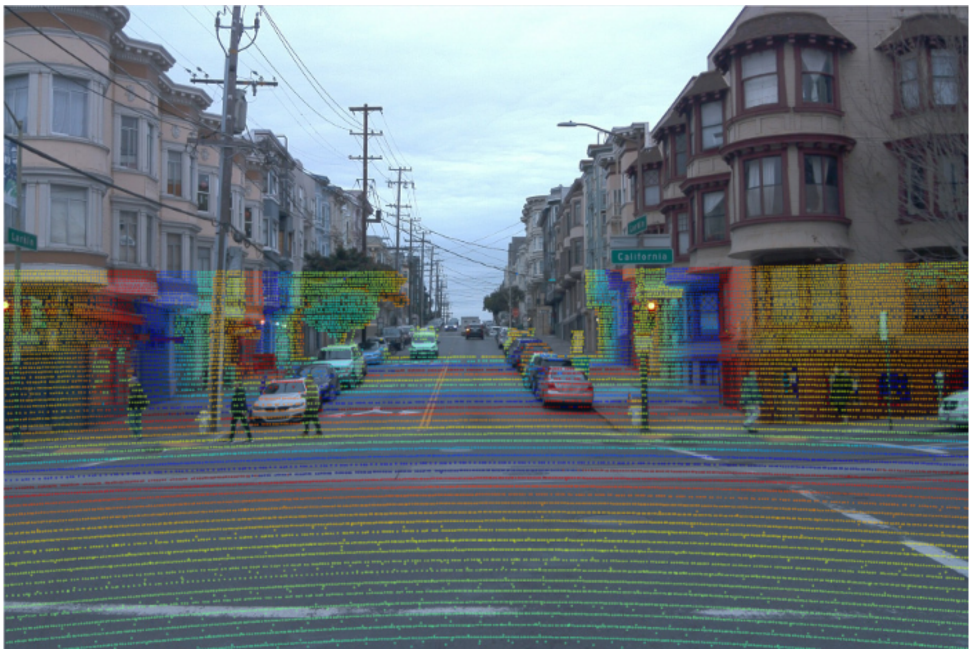


Figure 2.3: An example from the Waymo dataset [33] with the LIDAR data overlayed on top of the image.

type of measurement is the case for the *KITTI* datasets [15][26] created by the Karlsruhe Institute of Technology (seen in figure 2.2), and the *Waymo* dataset [33], created by the Google sister company Waymo for autonomous driving car development (seen in figure 2.3). *ScanNet* [9], a different dataset with depth information, sources such data differently, using off the shelf components such as a tablet and a 3D scanning sensor and provides the complete reconstructed 3D scenes together with the depth information as seen in figure 2.4. A common issue in these datasets is that due to the use of a LIDAR sensor mounted on a different position than the camera itself, the depth information is often sparse and doesn't contain information for all pixels of the camera view. The framerate of the LIDAR sensor is usually also not synchronised with the camera framerate.

One segment where there are issues in obtaining real-life datasets is optical flow information. Optical flow data describe the change of position of the surface represented by a pixel in two successive frames. A few datasets contain real measured data, such as the *Middlebury* dataset [2], released in 2007. The camera shows small scenes covered with a fluorescent paint pattern captured under both visible and UV light. Since the fluorescent paint is evident under

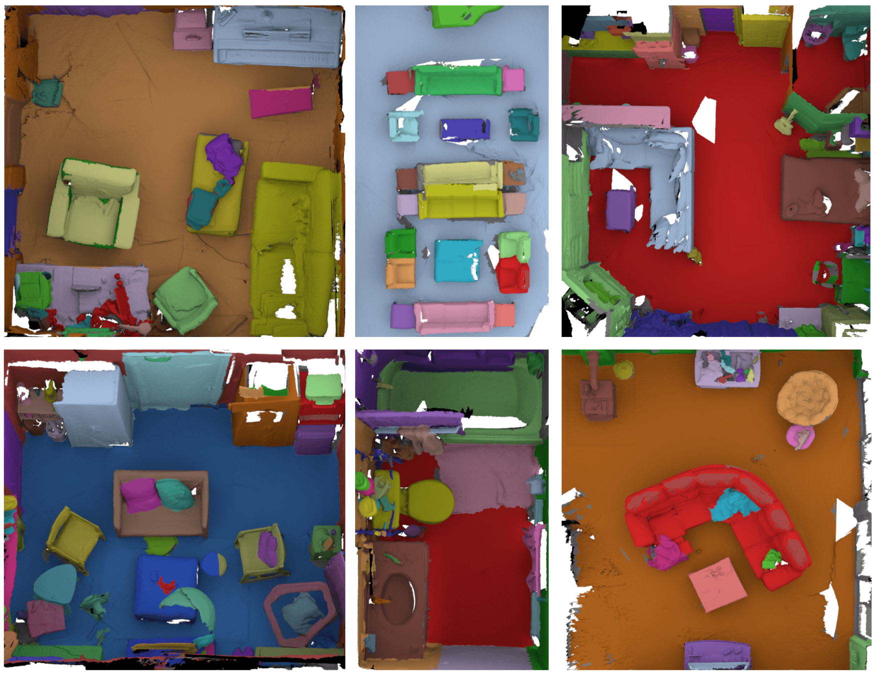


Figure 2.4: The ScanNet dataset [9] was created by using a depth sensor and contains hand annotated 3D semantic segmentation of indoor scenes.

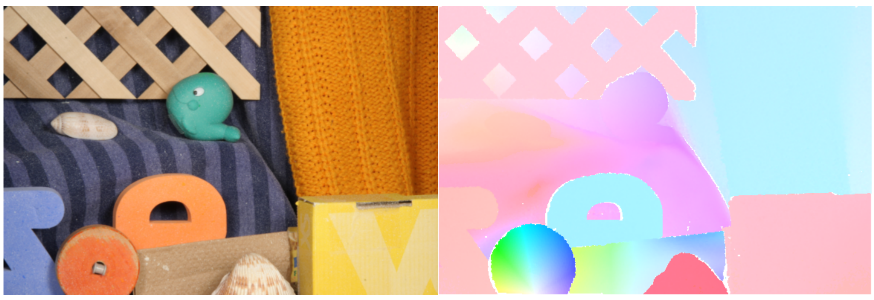


Figure 2.5: An example from the real-life section of the Middlebury dataset [2], acquired by using a special scene with fluorescent paint applied. Image on left represents the camera view and image on the right shows the optical flow field encoded as colors.

the UV lighting, the ground truth data was recoverable. As this method is complicated to reproduce, only eight short sequences using this method exist. A frame from this dataset is visible in figure 2.5. *KITTI* [15][26], also containing real-life optical flow data, calculated the data with the help of a LIDAR and egomotion of the car. Due to the way the calculation works, the framerate of the flow data is tenth the framerate of the camera itself and is only available for static parts of the scene.

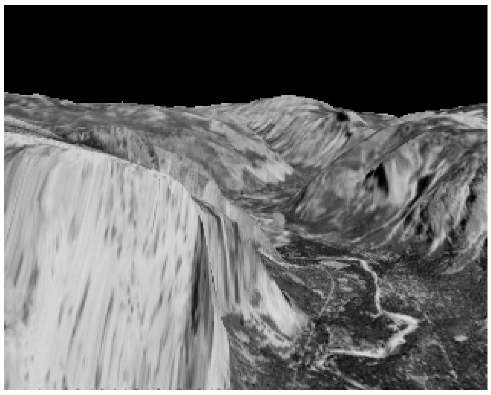


Figure 2.6: The Yosemite Flow Sequences [3] dataset is an early synthetic datasets used to evaluate optical flow estimation, released in 1994.

Capturing the optical flow in real-life scenes is a difficult task, so most other datasets build on synthetic generation. The first synthetic datasets used for evaluating optical flow estimation algorithms date back as early as 1994, where Barron, J. et al. used a *Yosemite Flow Sequences* dataset showing a 3D visualization of the Yosemite mountain range in [3] (seen in figure 2.6). In *Middlebury* [2], the eight remaining short scenes available are synthetic scenes rendered using the realistic MantaRay renderer. *FlyingChairs* [10] is another noteworthy synthetic dataset, later extended into *FlyingThings3D* [25] – simple objects (e.g. chairs) floating along random trajectories are rendered using a modified version of the open-source Blender renderer which allows the reading of optical flow data. Surprisingly, this abstract movement which has no relation to the real behavior of moving objects (the objects intersect each other) has been shown as an effective way of training neural networks. Synthetic datasets can also be built to emulate existing real-life datasets. Such is the case with the *Virtual KITTI* datasets [14][6] which contains scenes emulating the existing KITTI datasets.

Use of a modified Blender renderer also allows for datasets based on scenes from open-source animated shorts, *Sintel* [5] and *Monkaa* [25]. Although the use of the preexisting projects is excellent for more diverse outputs, it can also cause issues – camera behavior such as a change in focus may not be desirable for some usages. The last analyzed dataset that might be of interest is the *CrowdFlow* dataset [30] which shows aerial views of large crowds of people rendered in Unreal Engine as seen in figure 2.7. This dataset shows that for some uses, datasets specialized for a single task could be beneficial. In this

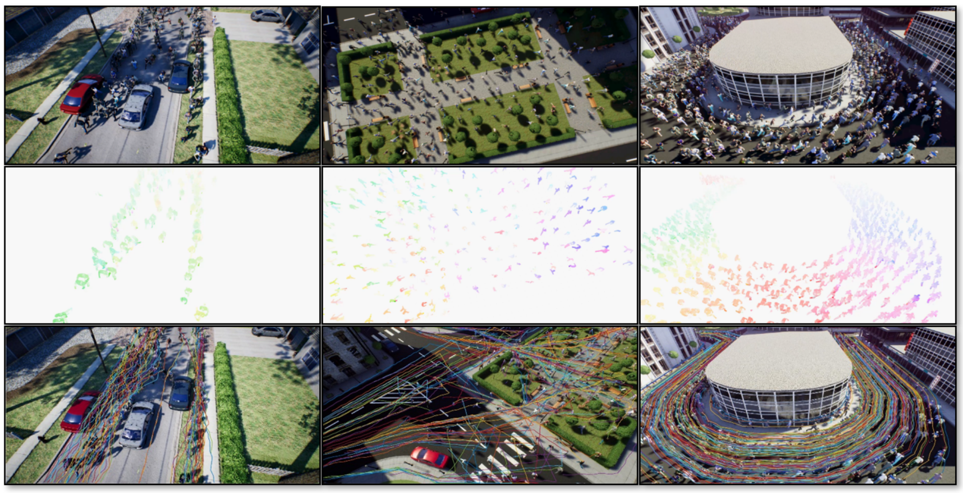


Figure 2.7: The CrowdFlow dataset [30] shows aerial views of outdoor scenes. It includes optical flow and trajectories of up to 1451 people.

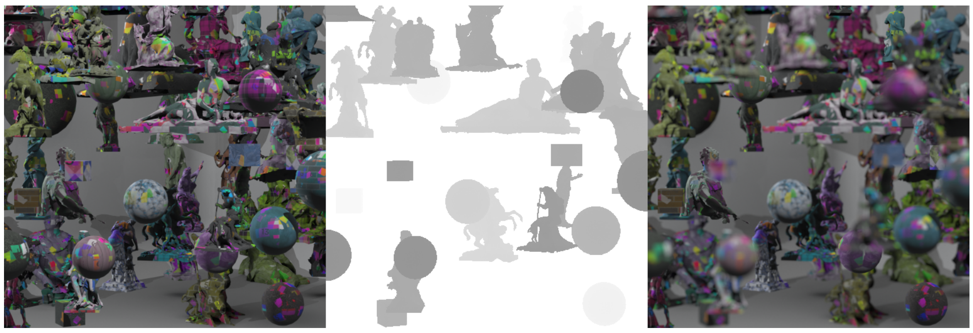
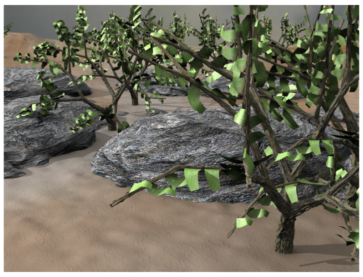


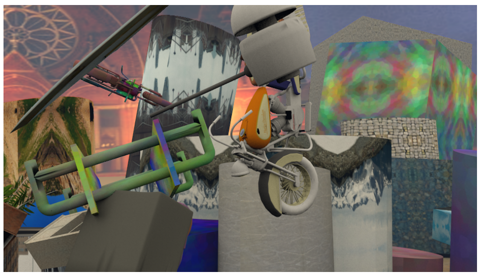
Figure 2.8: The DeepFocus dataset [36] contains images rendered using rasterization together with depth maps and corresponding images with accurately simulated defocus blur.

case, the dataset targets tracking behavior in large crowds. A selection of datasets containing optical flow ground truth information is seen in figure 2.9.

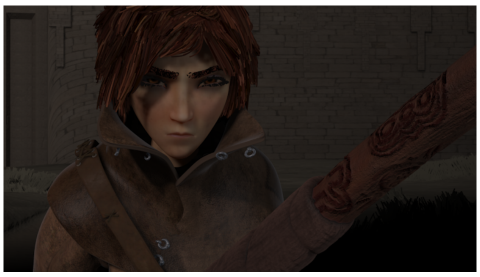
More recently, machine learning also begins to find more use not only in computer vision, but also in the field of computer graphics. For example, *DeepFocus* [36], a machine learning algorithm that emulates realistic depth of field defocus blur faster than other systems generating such blur using physically based methods. The algorithm was trained on a publicly available dataset, with ground truth depth of field blur generated in the Unity game engine using an accumulation buffer, and can be seen in 2.8. We include a comparison of a selection of publicly available datasets in table 2.1.



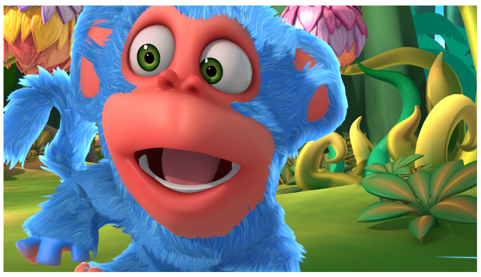
(a) : Middlebury dataset [2]



(b) : FlyingThings3D dataset [25]



(c) : Sintel dataset [5]



(d) : Monkaa dataset [25]

Figure 2.9: Different synthetic datasets containing optical flow data. The datasets 2.9a and 2.9b use relatively simple scenes, while the datasets 2.9c and 2.9d are based on existing animated short films.

	Synthetic	Segmentation	Depth	Optical Flow	Occlusions	Stereo	3D Bounding Box	2D Bounding Box	Frame Count
COCO [23]		X							$> 300,000$
Middlebury [2]	½		X	X					52
KITTI [15][26]			X[1]	X[1]		X[1]	X	X	$\approx 15,000$
Waymo [33]			X			X	X	X	$\approx 200,000$
FlyingChairs [10]	X		X						$\approx 20,000$
FlyingThings3D [25]	X	X	X	X		X			$\approx 20,000$
Monkaa [25]	X	X	X	X		X			$\approx 8,000$
Sintel [5]	X	X	X	X	X	X			$\approx 8,000$
CrowdFlow [30]	X			X					$\approx 3,000$
DeepFocus [30]	X		X						$\approx 5,000$

[1] Sparse data for a limited frame subset only

Table 2.1: A table comparing a selection of available datasets.

Figure 2.10: A scene from the CARLA [11] autonomous car simulator.

2.2 Generators

Several utilities for simplified creation of computer vision datasets already exist. Some of them are a part of more massive simulators, such as *CARLA* [11], an autonomous car simulator (seen in figure 2.10), or *AirSim* [32], a simulator for autonomous drones and cars (seen in figure 2.11). Both of these utilities are built using Unreal Engine and provide both C++ and Python APIs to control vehicles in the scene. The APIs also allow retrieving of synthetic image data from virtual sensors attached to the vehicles. Their primary purpose is not the generation of new datasets but simulating entire road or sky scenes for virtual vehicles to move in, so the types of augmented image data are limited mostly to basic types such as depth or segmentation maps.

There are some preexisting plugins for game engines that enable the acquisition of augmented image data. One of which is *NVIDIA Deep learning Dataset Synthesizer (NDDS)* [34], which, built on Unreal Engine, provides blueprints to access depth and segmentation data, along with bounding box metadata and additional components for creation of randomized scenes. An example of

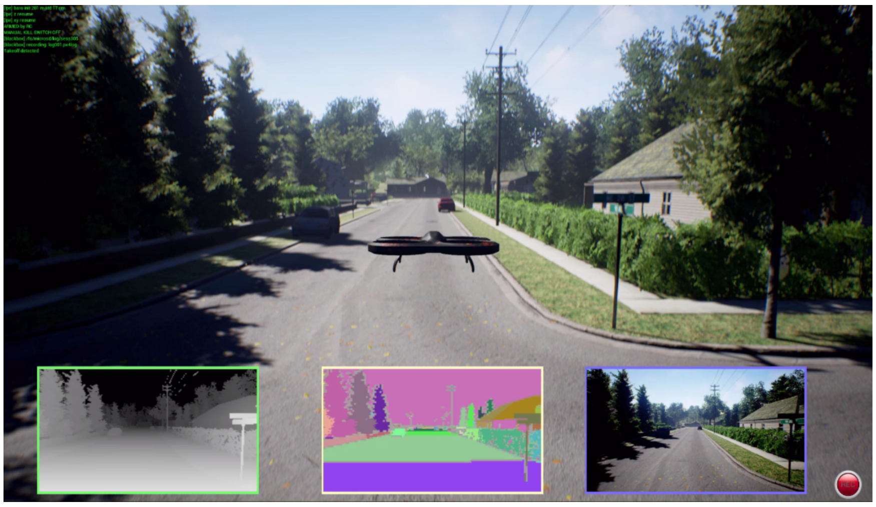


Figure 2.11: A snapshot from AirSim [32] autonomous drone and car simulator, showing different ground truth camera outputs.

a dataset generated with NDDS can be seen in figure 2.12. Another option built on top of Unreal Engine is *UnrealCV* [29], which, compared to NDDS, exposes Python API to capture the images programmatically and directly feed them to a neural network. The API allows interacting with objects in the scene, setting labeling colors for segmentation and retrieving of depth, normal, or segmentation image data. The system is virtually plug-and-play, where the plugin can be added to an existing Unreal Engine project or game and start generating augmented image data.

By default, the Unreal Engine does not provide access to motion vector data, which represents backward optical flow from current to the previous frame. Nevertheless, since the source code is available (under a proprietary license), such functionality can be enabled by modifying the source code and recompiling the engine. *Unreal Optical Flow Demo* [19] presents a patch enabling the functionality used in Unreal based robot simulator *Pavilion* [20].

The last generator analyzed is a simple plugin for the Unity game engine. *ML-ImageSynthesis* [37] is a script providing augmented image data for object segmentation, categorization, depth and surface normal estimation. Compared to other previously mentioned plugins, it also provides backward optical flow data, which is obtained from Unity motion vectors. An example of the outputs generated by the plugin can be seen in figure 2.13.

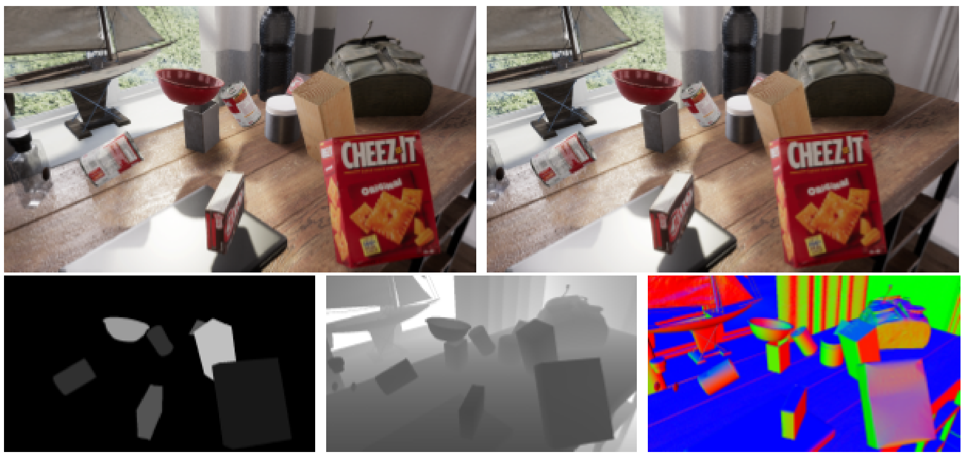


Figure 2.12: An example of a dataset generated using the NVIDIA Deep learning Dataset Synthesizer containing stereo views, per-pixel segmentation, depth and surface normal information [18].

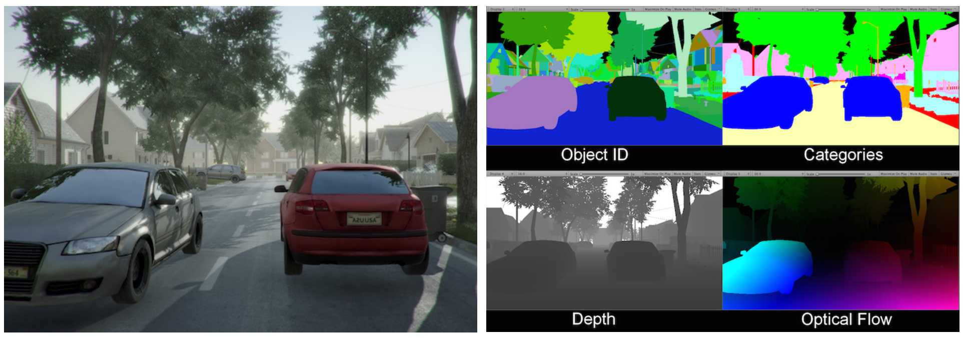


Figure 2.13: Example outputs from the ML-ImageSynthesis Unity plugin.

2.3 Representing Ground Truth

In the existing datasets, there are several different approaches how to represent the ground truth data. For object segmentation the COCO dataset [23] uses simple 2D polygons to mark down the outline of each object labeled in the image. Image segmentation can be also understood as assigning each individual pixel on screen a label which is the same for each pixel sharing a given characteristic. Most synthetic datasets (if not all) follow this principle and represent segmentation by a simple three channel raster image, where each color is understood to be a unique label.

The raster image representation has a downside in size, as it is many times bigger than representing just boundaries of the segmented image, but is considerably easier to synthesize, as the generator can simply use the same rendering pipeline that was used for the RGB camera image, just with modifications to output the same value for each object. This representation is also more desirable for the use in modern machine learning algorithms. For example the COCO dataset API provides functionality to convert their polygon representation into raster masks.

Both the polygon and raster representations of segmentation have issues with scenes where a pixel represents more than only one object. As all real scenes can contain transparencies, depth of field and motion blur, these scenes will often not be represented correctly using these approaches. In addition, raster representation suffers from aliasing on boundaries, as each label is represented as a unique color. For polygon representation, simply allowing the polygons to overlap would solve a part of the issue. In that case we would be able to tell whether a pixel represents two objects at once, but we still wouldn't be able to tell how much each object influences the pixel. For raster representation, another possibility is to use separate images as different depth layers of the segmentation. This representation is called Layered Depth Images (LDI) and was first described in [31]. Such raster representation can be for example generated with Z-buffer rendering using a depth peeling technique described in [12]. The image is rendered once for each layer, using the previous layers depth buffer to limit the closest rendered point. A side-view example of this rendering approach can be seen in figure 2.14. However, this approach still cannot easily represent motion or defocus blur, as Z-buffer rendering techniques only emulate such effects using post processing.

Similar needs are often found in film production, where precise anti-aliased masks are required for different tasks during postproduction (e.g. recoloring already rendered objects). Several specialized formats for saving such mask already exist, with the current industry-standard being Cryptomatte [13]. It is an open standard and has support in a large number of visual effects and compositing software (Blender, V-Ray, OctaneRender, RenderMan, Houdini, Nuke, Fusion. . .). As a base, the format uses OpenEXR multichannel images,

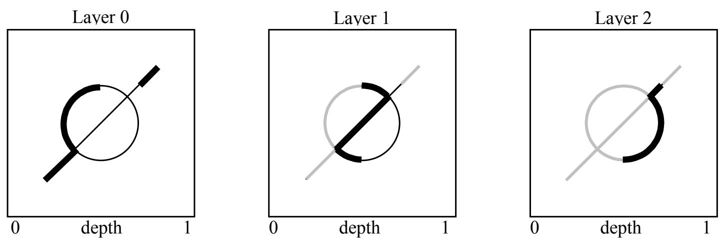


Figure 2.14: Depth peeling principle from [12] in each successive pass. The images show the view of a scene from side, with the camera looking in from the left. The currently drawn frontmost surface is a bold black line, hidden surfaces are thin black lines and "peeled away" surfaces are light grey lines.

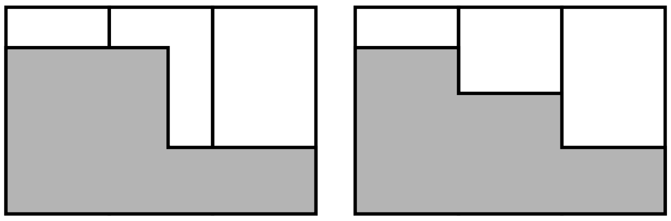


Figure 2.15: An example of error introduced by antialiasing depth values. Side view, columns represent the pixels. Right image represents the actual geometry, left image shows incorrect value introduced by antialiasing.

into which it encodes either object ID, namespace ID, or material ID matte. The format could also be repurposed with relative ease to represent segmentation masks for images, as tools for generating such file are widely available. Interpreting such data can cause some complications, because the format is relatively hard to understand.

For depth information, similar issues with regards to transparencies, depth of field or motion blur occur. When rendering with antialiasing, incorrect values appear on boundaries, as antialiasing would just average the two distinct (but correct) values with an incorrect value, as seen in figure 2.15. Just as with the previous situation, visual effects industry often works with depth information to insert new objects to a scene during post production. An open standard to represent such images exists. The OpenEXR file format [21] has support for "deep images", in which each pixel can store an unlimited number of samples, each associated with a distance from viewer, or depth. Deep image workflow is also possible in a number of visual effects and compositing software (V-Ray, OctaneRender, Houdini, Nuke. . .), therefore tools for generating OpenEXR deep images are widely available. However, interpreting deep images could pose a problem, as even though OpenEXR is a open-source format, libraries for reading the format are often limited and for example no existing Python libraries support deep images.

Chapter 3

Synthetic Data Generation

In this chapter, will describe different parts of the simulation. We talk about the steps necessary to simulate the realistic camera view, the world itself and finally, we discuss the different approaches to generate additional ground truth outputs for the camera view.

3.1 Simulating the Camera

When creating synthetic datasets, much care should be taken not only when generating high quality augmented image data such as segmentation but also with simulating the view of a real camera, for which we generate the ground truth images. We can understand simulating the camera as simulating the camera sensor, its optics and geometry and the movement of the camera.

First, we focus on describing the sensor itself. In a real camera, light, passing through the camera optics, is captured by a sensor. We can represent the light falling on the sensor as an continuous image function $f(x, y)$, where x and y are the coordinates on the surface of the sensor. The sensor samples from this function, usually in a regular raster, and outputs a quantized value

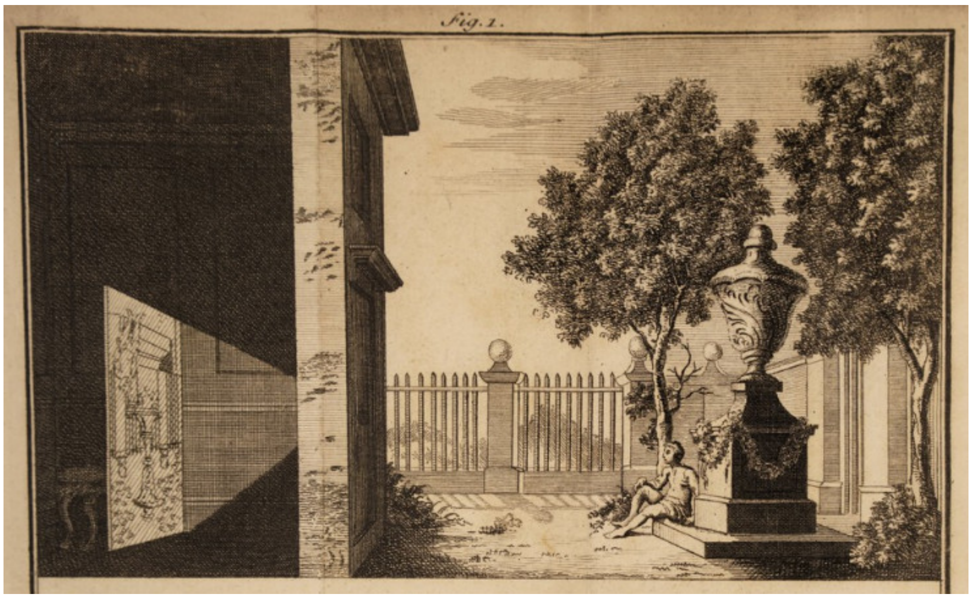


Figure 3.1: Illustration of the camera obscura pinhole camera from James Ayscough's A Short Account of the Eye and Nature of Vision (1755, fourth eidition).

of the sample as the individual image pixels. The sensor measures the values in a given time period – exposure. Longer exposure means often a brighter, less noisy image, but moving objects appear blurred.

When looking at the optics and geometry of a camera, we first must understand how to simulate an ideal camera with no distortion of the view. The pinhole camera model describes such camera geometry. The model is based on a camera obscura (figure 3.1), an optical phenomenon where if a box contains a small hole on one side, the light coming through the hole projects the outside view on the back side of the box. With the geometric model, we imagine the hole being represented by a single point through which all the light rays projecting the image must pass through. The projected image through an ideal pinhole, as used in the geometric model, is uniformly sharp and contains no distortion, and is often used to reasonably well approximate the behavior of actual cameras. It is relatively easy to project on the image plane, as each point in space in front of the image plane is projected only to a single point on the plane.

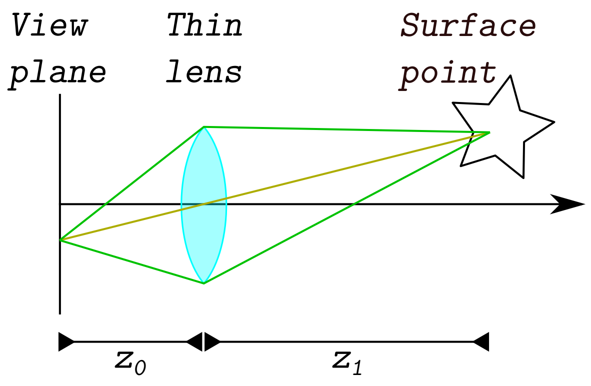


Figure 3.2: The thin lens geometry model focuses all light from a single point in the focal distance to a single point in the view plane.

Real camera sensors usually require more light than what can usually pass through the small pinhole of the camera obscura, so bigger lenses focusing light on the sensor are used. This allows sensors to operate properly, but causes unfocused objects to appear blurry. In real cameras, the system of lenses focusing light on the sensor can be very complex and is often simplified by using a thin lens model. The simplified model works with a lens which has zero thickness and focuses all light hitting the lens from a single point on the focus plane to a single point on the image plane.

Most camera systems can be relatively well simulated by projecting on an image plane by using either of these methods, but in case of camera optic systems which include strong distortion this is not possible. The pinhole model and the thin lens model are both unable to properly simulate wide-angle lenses such as the fisheye lens, since they are unable to project points on the plane going through the camera center which is parallel to the image plane. If we want to simulate such projections, it's often possible to simulate 6 thin lens cameras aligned in such a way that their image planes form a cube, and then distort the cube based on the lens we want to simulate (since the cube contains an accurate 360° view of the scene).

Generating realistic and semi-realistic images is a well-researched field. Modern approaches, such as Physically Based Rendering [28] seek accurate modeling of the flow of light to achieve highly accurate and realistic images. As we want our datasets to be able to train algorithms that apply to real-life situations, we should aim to generate the reference camera images as realistic as possible. This means including effects such as depth of field, motion blur, caustics, or even issues in the camera mechanism such as optical aberrations or noise.

Systems that generate realistic images these days most often use rasterization methods for real-time interactive rendering and ray tracing based methods for more physically accurate but slower (often offline) rendering. Rasterization is based on projecting objects on the image plane using the pinhole camera model, and usually simulates defocus blur during post-processing. When generating datasets, real-time rendering or interactivity of the scene is not a priority, as we want to save the dataset on disk for later use. Using ray-tracing based algorithms such as path tracing to render the realistic camera view is ideal for creating the simulated view of a real camera.

3.2 Simulating the World

The camera gives us access to images from the simulated world, and if we wish to generate as realistic images as possible, we must take close care that the world looks and behaves realistically as well. The world simulation then can include physics simulation, which makes sure objects in the scene do not intersect each other, weather simulation, simulating the behavior of rain and other atmospheric phenomena or agent simulation, controlling the behavior of vehicles and other actors in the scene.

The person creating the dataset should be able to configure the behavior of the world based on the needs of the dataset. Some datasets require simulating crowds of people (such as in the *CrowdFlow* dataset [30]), some datasets require simulating vehicle traffic (similar to the *KITTI* datasets [15][26]), while some datasets should visualise highly abstract scenes (similar to the *FlyingThings3D* dataset [25]).

As the structure of the world and the behavior of the simulation is heavily dependent on the target dataset, we do not describe the process in detail. The scene can be defined inside a 3D modeling software or a scene editor of a game engine, similarly to the world behavior, which can be baked as an animation in modeling software or run real-time inside a game engine.

3.3 Simulating Ground Truth Measurements

Methods for generating ground truth outputs can be considerably more straightforward than the systems used to generate the camera view itself, as the ground truth is only a subset of all the information included in the realistic camera view. Realistic effects such as antialiasing or depth of field would often be detrimental to the ground truth data when stored in a simple 2D image. For example, such effects break categorization labeling in segmentation masks (although representing such data with more complex data structures is possible, as discussed in section 2.3). Therefore, we can utilize more straightforward methods to generate ground truth data based on rasterization. In the next sections, we describe how different ground truth data are calculated.

3.3.1 Depth Output

One of the more straightforward outputs to generate is depth information for each pixel in the image. With such output, one can calculate the camera-relative position of each point in the image by using the depth information in conjunction with the screen space position of the point and the knowledge of the image's field of view angle. For raytracing, the exact ray intersection point in world space is directly available. For rasterization, the Z-buffer non-linearly encodes the distance of each pixel from the camera plane.

There is a significant distinction between the distance from the camera plane and the distance from the camera center. One might think the depth

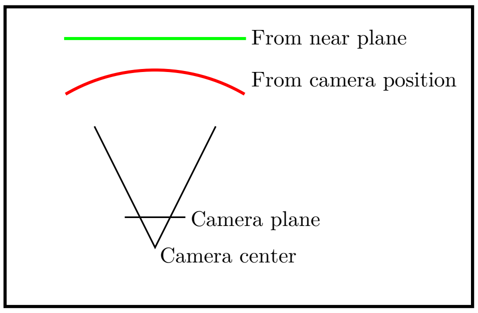


Figure 3.3: The difference between measuring the distance from the camera plane and the camera position. The green line shows points with unit distance from the camera near plane. The red curve shows the unit distance from the camera itself.

in the image is the same as its distance from the camera (in this case, the camera position), but in fact, the depth information describes the distance from the camera plane instead. The figure 3.3 shows this distinction. The green line on the top shows points with unit distance from the camera near plane, while the red curve shows the unit distance from the camera itself.

Converting between these two representations of distance is relatively easy when the camera parameters are known, but the distinction is still relatively important. In most cases, the output we want is the distance between the camera plane and the point, which when using rasterization with the perspective projection is directly encoded in the Z-buffer by this formula:

$$z_{Linear} = \frac{2.0 * z_{Near} * z_{Far}}{z_{Far} + z_{Near} - z_{NonLinear} * (z_{Far} - z_{Near})}$$

where $z_{NonLinear}$ is the value from the Z-buffer in range $[0, 1]$, and z_{Near} and z_{Far} are the near and far plane distances of the projection matrix.

3.3.2 Normals Output

The information about the normals of all visible surfaces can help with new understanding of the scene. We could calculate an approximation of the normals as a gradient of the depth output, but this approach could cause issues with scenes with high-frequency changes in depth (such as when viewing a chainlink fence). Normals are also directly accessible during rendering, as they are relied upon when shading the image. Calculating the normals separately from the depth output allows us to represent them accurately when normal information is separate from the geometry itself. This happens for instance when we interpolate normals between vertices using smooth shading or when we use normal mapping.

There are multiple ways we can represent the normals inside the dataset. One way is to return the normal vectors in relation to the camera rotation only (in view space), and the other is to display the normals modified by the perspective transform (in screen space). The difference between the two outputs can be seen in figure 3.4. When displaying normals in relation to the view (displayed on the left), flat surfaces share the same value in the output. On the other hand, when displaying normals modified by the perspective transform, the value is perceptionally correct and as such changes over flat surfaces. The post-perspective transform representation of normals can be used to directly visualise the normals on the image, while the view-space representation is more useful for segmenting the image, as flat surfaces have the same value.

Converting between the different representations of normals can be done by using the camera relative position of the point in space which can be calculated by using the depth output.

3.3.3 Bounding Box Outputs

One of the goals of computer vision is to detect where objects are located in an image and in the scene itself. For that, selected objects or object categories should have both camera relative 3D bounding boxes and screen space 2D bounding boxes made available.

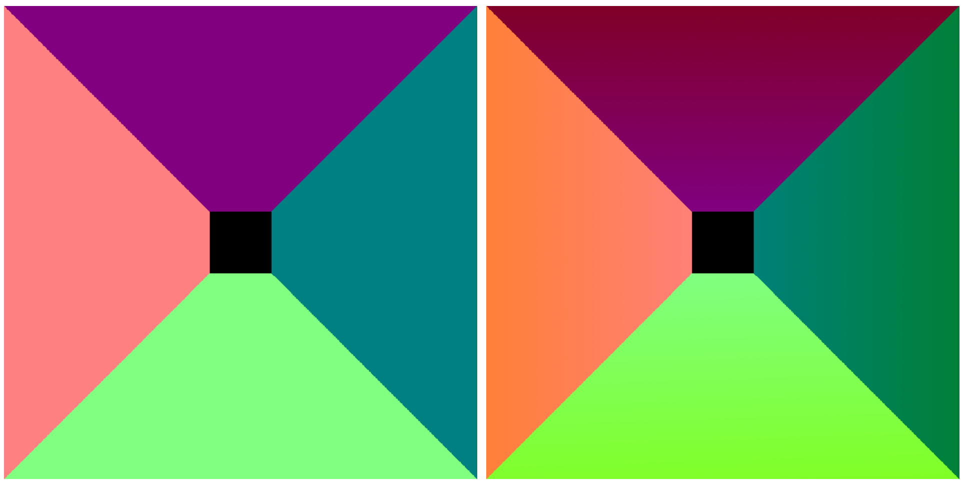


Figure 3.4: The difference between view space oriented normals (left) and screen space oriented normals (right). The images show a view through a long hallway, with the normals mapped as RGB colors. With view space oriented normals, the normals of the flat walls stay the same color, while with screen space oriented normals the color isn't constant, indicating their direction changes.

We build 2D screen-space bounding boxes from minimum and maximum screen coordinates of the transformed vertices during rasterization. As such, they are relatively straightforward to acquire when the entire rendering pipeline is under user control. Often, though, this is not the case. For example, with the OpenGL rasterization pipeline, the vertex transformation, which happens inside the vertex shader, is directly connected to other parts of the pipeline. Its outputs (the transformed vertices) are not available to be read directly. It is often necessary to reimplement the transformation elsewhere and calculate it separately from rendering, either on CPU or as a GPU compute shader.

When implementing the transformation for rigid objects, using the objects convex hull can also speed up the computation. The bounding box of the convex hull and of the object itself is identical, and the convex hull will often contain considerably fewer vertices. Convex hull computation is a relatively costly computation (often $O(n\ log\ n)$) and is not suitable for non-rigid objects. It can also return incorrect values due to floating-point precision issues. This optimization, although often useful, should not be set by default for all objects because of these issues.

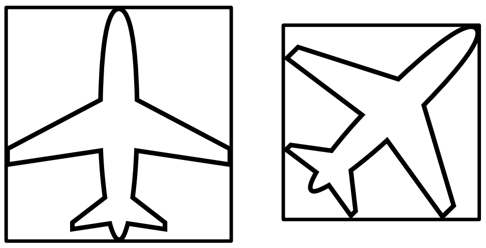


Figure 3.5: A comparison between an object aligned bounding box (left) and a tight bounding box (right). The airplane icon provided under public domain by Burak Kucukparmaksiz.

For 3D bounding boxes, the calculation is considerably more straightforward, as rigid objects do not change size in time. Therefore, we set their size per object before the dataset creation. As tight 3D bounding boxes are used most often in current datasets, one might assume that we should limit the tool only to output tight bounding boxes as well, but this is not the case. Bounding boxes are currently most often used to represent cars, for which tight bounding boxes make the most sense – the front plane of the box represents the front of the vehicle, and we want to align the bottom plane of the box with the ground. However, for some other vehicles, we would like to create object detectors for, this would not apply for tight bounding boxes. The figure 3.5 shows a situation where a tight bounding box does not correctly represent the vehicle. Therefore, 3D bounding box orientation should be provided together with the model.

Some care should be taken when saving bounding box information on objects in the scene. The scene can contain many objects, for most of which we do not need to generate bounding boxes. Therefore, we should be able to generate bounding objects per object or object category. Objects are also often nested in a scene graph hierarchy, and we must make sure that, when creating bounding boxes of non-leaf nodes, we include all its child meshes.

3.3.4 Segmentation Outputs

Image segmentation is the process of dividing the image into different segments that contain pixels that share a particular feature. Instance segmentation separates the image into segments in such a way that each unique object in the image belongs to its unique segment. Similarly, we can use any other semantic feature to segment an image. For example, an image of a street can contain several segments based on predefined categories such as road surface, footpath, or buildings.

As discussed in section 2.3, we can also understand segmentation as assigning each image pixel a unique label. When preparing the scene, we can assign a specific color label to an object, and when drawing the object, draw only the assigned color instead of shading the object. With some segmentation outputs, we can rely on the computer to automatically label objects. Object segmentation can be achieved by automatically giving each object a unique label. Motion segmentation can label objects that are moving in the scene, and segmenting unique materials or meshes is also possible. Segmenting the scene in different human-understandable categories requires the objects to be categorized manually before the dataset creation.

3.3.5 Amodal Segmentation Masks

Instance segmentation map contains masks for all visible objects in one image. When an object is partially occluded, the masks are occluded as well. When viewing partially occluded objects, one might assume what shape the object has in the occluded region. In the visual perception field, the phenomenon that humans are capable of estimating occluded shapes is well observed and called *amodal completion* [24]. As we want to create algorithms that are as good (or even better) at understanding the scene, we should be able to generate ground truth data for partially occluded parts of the scene. There is a relative lack of amodal instance segmentation datasets, and such a dataset could prove useful [22].

Rendering an amodal mask of a single object is a relatively simple task, as it the same as rendering the scene with only the given object. Issues may arise when the amount of objects rendered in the scene is high, as we must create a separate mask image for each object. If we want to limit the number of different images we have to generate, multiple approaches are possible.

First, it is possible to render multiple objects into the same image under the condition that they do not overlap. We calculate which objects do not overlap by comparing their separately calculated 2D bounding boxes. Second, we can render onto the same buffer multiple times, each time into either a different channel or, using the additive blending mode in the rasterization pipeline, using different bits inside a single channel. This way, with an RGBA image with 8 bits per channel, we can get up to 32 different layers on which we can draw amodal masks of different objects (or different sets of non-overlapping masks).

3.3.6 Optical Flow

Understanding movement in a sequence of images is an essential task in understanding the scene itself. Optical flow describes the distribution of apparent velocities of movement of visible patterns in a sequence of images. It is often used to estimate object motion, as it represents the relative movement of objects and the viewer [17]. Discontinuities in such optical flow also help when segmenting images into regions of corresponding objects.

Using this definition optical flow doesn't directly carry information about the movement of objects in the scene itself, but only about the movement of "visible patterns", and therefore finds its uses also during video compression. For example when using the MPEG video compression, a process called motion-compensation is used. Each frame is split into 16×16 macroblocks, and each macroblock contains a motion vector describing movement of the block, together with information on how the block differs from a previous (or future) frame. Optical flow can then be used as a basis on which the macroblock motion vectors are calculated [7].

The relationship of optical flow and object motion is not fully specified by this definition. For example, a rotating uniformly colored lambertian sphere does not contain any visible patterns. Its optical flow is static and does not represent the actual object motion in the scene. If we target our ground truth output of this category at scene movement estimation, our output should not just describe the apparent velocities of movement of visible patterns, but actual velocities of the projection of visible surfaces (pixels) onto the image plane. The vector field describing the motion of surfaces is sometimes called motion field [35], but in almost every dataset or benchmark (such as in Sintel [5]), the motion field output is labeled as optical flow. As we wish to keep in line with existing research, we label our output as optical flow as well.

Such per-pixel optical flow is relatively easy to compute and is often used in game engines to approximate motion blur during post-processing [27]. We transform both using the transformation matrix of the current frame and the previous frame during the rasterization process. The difference between the two transformed screen-space positions is the resulting optical flow. Let M_{Cur} and M_{Prev} be the current and the previous transformation matrices from world space to screen space, v be the transformed vertex, and $X(v, M)$ be the transformation operator. From [7] for each vertex in the scene, the optical flow when going from M_{Cur} to M_{Prev} is

$$\Delta v_{screen} = X(v, M_{Cur}) - X(v, M_{Prev}).$$

As the surfaces between vertices are traditionally flat triangles, we interpolate the optical flow of any point on the surface of a triangle from the optical flow of the vertices of the triangle. This computation results in backward flow information for the current frame. If we want to compute forward flow, we can use M_{Next} instead of M_{Prev}, representing the transformation matrices of the objects in next frame.

This calculation applies similarly for stereo disparity, where instead of working with two transformation matrices representing view from the same camera during different points in time, the two matrices represent the view from two separate cameras.

Another related term to optical flow is scene flow. Whereas we understand optical flow as a field of 2D vectors representing the movement of points on the screen we are projecting the scene on, scene flow is a 3D field representing the actual movement of points in space, including the movement in the camera view aligned Z axis.

3.3.7 Occlusions

When viewing a scene from two distinct points in time or space, it is often needed to find corresponding points between the images. Between different moments in time, optical flow vectors connect such points, and disparity vectors connect them between different camera views. Not all points have such correspondences, as they may not be visible in the other images. Such points then cannot be, for example, used to calculate distance using binocular disparity. A way to mask out such occluded points is then necessary. Occlusion output then highlights which points visible in one image are occluded in an image from a different point in time or space.

Such a situation is relatively easy to detect using a modified raytracing renderer: Send out rays from the camera, when the rays hit a solid surface, send out a ray from the hit point towards the camera we are checking for occlusions. When checking for an occlusion between multiple points in time, we transform the point according to the the movement of the object the point belongs to before sending the ray towards the camera.

With rasterization, it is slightly more difficult to check such information in a single pass precisely. There are multiple ways of approximating such outputs. For stereo disparity, we can use a system similar to direct lighting calculation to get a precise mask: When rendering, replace the camera we are checking the occlusions against with a light source. We then see the visible points from the other camera in the lit part of the image, and occluded points are in shadows. When using accurate shadowing techniques such as shadow volumes the results are precise [8]. However, they are limited to occlusion from polygon shapes (shadow volumes do not support objects using an alpha cutout textures such as a tree leaf). Shadow volumes emulate shadows by manually calculating a 3d mesh representation of the unlit volume, which we then use to decide whether a surface is lit or in shadow.

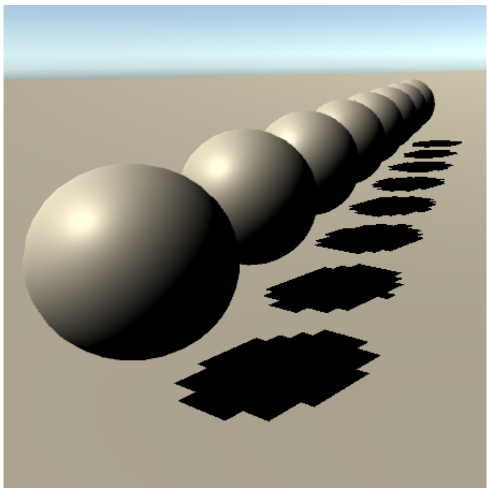


Figure 3.6: A perspective aliasing issue as seen in Unity - shadows closer to the camera show an error

Other less accurate methods for casting shadows can be used, such as shadow mapping. Shadow mapping works by rendering the scene's Z-buffer from the light source's point of view and mapping it on the surface of the scene. When rendering, we then compare the distance from the light point in the shadow map texture with the distance from the light point and the currently rendered point. If the distance is higher, we consider the currently rendered point to be in shadow. In shadow mapping, situations can arise where we output an error greater than one pixel in size at the border of the shadow. For example, we see a perspective alias when looking at the shadow mapped texture near the camera (as seen in figure 3.6), or a projection alias when we cast a shadow on a plane almost parallel to the light direction (as seen in figure 3.7).

These methods get slightly more complicated when calculating temporal occlusions between two moments in time. However, we can use the shadow volumes or the shadow map from the previous frame with both approaches instead of that from the current one. The issue with no support for cutout textures with shadow volumes and accuracy issues with shadow mapping can still occur.

Another option is to calculate occlusions in a post-processing step using already existing outputs. For example, if we have both scene flow and depth

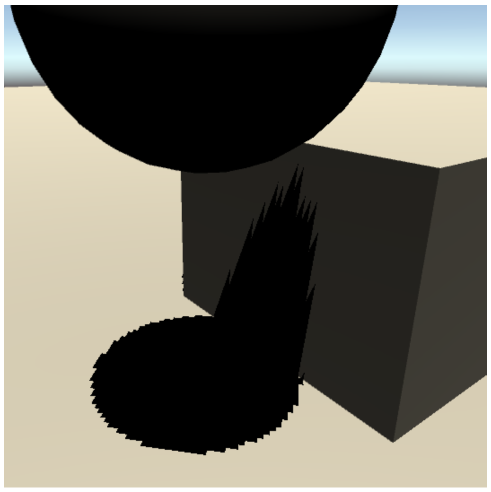


Figure 3.7: A projection aliasing issue as seen in Unity - shadows on surfaces parallel to light direction show error

output available, calculating the occlusions is done trivially by comparing the depth of each point with the depth of the given point when shifted by the scene flow vector of the given point. This relies on the scene flow vectors being three dimensional, but if only 2D optical flow information is available, is not applicable.

If scene flow is not available, it is still possible to use 2D optical flow to reach an approximate level of certainty on whether or not is the point visible in the next frame. For example, when the object segmentation of the point and the point its optical flow vector points to in the other image are not the same, we can definitely say it has been occluded by another object. If we want to check whether the object is not self-occluding the point, we can use a separate buffer, on which we render for each pixel the local 3d coordinates of the object and then compare those.

The post-processing approach is not without issues, however. We are working with regularly sampled images, and the optical flow or scene flow vectors are pointing at precise points in the image, which almost never align with the samples. Therefore, it is impossible to properly decide whether a point is visible in the other image. When calculating object occlusion, a situation as seen in figure 3.8 can occur. As interpolating would break the labeling of the samples, we can check the four closest samples instead and

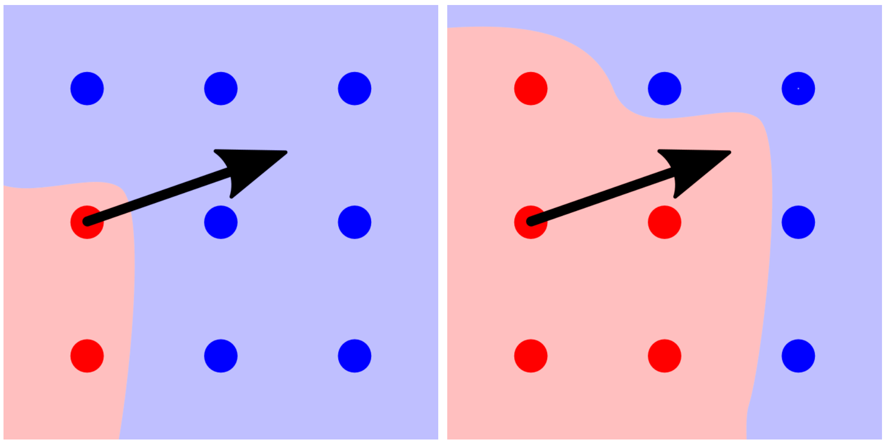


Figure 3.8: An issue occurs when using post-processing to calculate the occlusions when the optical flow vector does not point directly at a measured sample. In this case, looking at the corner sample of the red shape, even though the point does belong to the red area, we can only have a ¼ confidence about it's situation when following its flow vector (represented by an arrow in the image) to the next frame, since three of the four nearby samples belong to a different area

give a "confidence rating" whether the point is occluded. When working with depth map or the local 3D coordinates, linear interpolation can be used, and similarly a confidence rating can be returned instead of binary visible/occluded value.

3.3.8 Motion Segmentation

Motion segmentation is the task of identifying independently moving objects and separating them from the background [4]. Deciding whether a rigid object is in motion can be done by comparing the position and rotation of the object between two frames.

Question comes on how we should handle non-rigid objects. There are multiple different approaches on what to label as a moving object. When a part of a non-rigid object is moving, should we label only the moving parts, the entire object, or only label the object when its position or rotation changes? In our implementation, we decide to label the entire object only when its position and rotation changes, but it could be also possible to label each pixel independently, either by directly calculating the difference similarly to calculating optical flow, or (when access to rasterization pipeline modification is limited) by using the existing optical flow output and subtracting flow induced by the camera movement.

3.3.9 Camera Calibration

With all these relatively complex outputs, additional information about the camera should be also provided. First and foremost, both intrinsic and extrinsic camera parameters should be available for each camera view. In computer vision, intrinsic camera parameters are represented by a 3×4 calibration matrix K in this form:

$$K = \begin{bmatrix} \alpha_x & \gamma & u_0 & 0 \\ 0 & \alpha_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

For rendering using rasterization, the matrix representing the internal parameters of the camera is a 4×4 projection matrix P. When using the OpenGL framework, the matrix is written in this form:

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

At first, the two matrices may seem very different, but they represent the same process, and in fact, are equivalent, just with the third row of the projection matrix removed, as the calibration matrix only projects onto a plane and is not used for Z-buffer rendering. The parameters α_x and α_y represent the focal length scaled by the final projection space, u_0 and v_0

represent the center of the image and γ represents the skew factor of the image, which in case of the OpenGL projection matrix is 0. The z direction is flipped in OpenGL, and so the last row contains -1 instead of 1.

As the projection matrix represents more information about the projection used, sharing the matrix directly instead of converting it to the calibration matrix form is preferred. The parameters used to construct the matrix should be provided as well, because the matrix is often not user defined by using the t, b, l, and r terms, but those terms are computed from the screen shape and the desired vertical field of view.

Extrinsic camera parameters are represented by an identical 4×4 matrix both for rendering in computer graphics and calibration in computer vision. The matrix is in this form:

$$\begin{bmatrix} R_{3\times3} & T_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix}_{4\times4}$$

Where R and T are the extrinsic camera parameters, defining the rotation and position of the world with respect to the camera (the matrix is the inverse of the camera's transformation matrix). Therefore, we provide both the matrix and in addition separate information on the rotation and translation of the camera in world space.

Chapter 4

Design of the Data Generator

In this section, we discuss the broader design choices that were made when developing the ground truth generator and the scenes used for machine learning.

4.1 Platform

When considering the design of the utility, we considered three different platforms:

- Blender
- Unity
- Unreal Engine

All of these platforms are capable of rendering realistic images, which is one of the main requirements of this project. Blender has an integrated unbiased PBR path-tracer Cycles, and Unity and Unreal Engine use a high-quality integrated PBR rasterizing pipeline, with the possible use of external path tracing plugins such as OctaneRender. All three platforms were already used for the creation of datasets for machine learning. A part of the requirements is the ability to generate optical flow data, so systems directly allowing access to motion vector data are preferred.

Blender path-tracer allows for direct access to motion vector output via a vector pass in the settings and has a support of a limited scripting API for plugins, but for access to most data, direct changes in the source code would be required. As it is an open-source application, these changes are easily possible and have been previously made for the creation of specific datasets, such as the FlyingChairs3D dataset. It is an application purely targeted at 3D rendering and modeling. It does not contain a game engine, which means it can only render prebaked animations, and the scene cannot change interactively. The UI is also targeted for 3D editing and isn't that user-friendly to newcomers without prior experience and would be relatively challenging to accommodate for purposes of dataset generation.

Unreal Engine 4 is often used as a base platform for different simulators such as CARLA or AirSim. It provides a way of writing applications, either using a modified version of C++ or the Blueprints Visual Scripting system (combining both is possible, but can pose challenges), in addition to having direct access to the engine source code, which can be modified. Without modification, the engine does not allow the reading of motion vectors. Custom shader programs are only possible to be created by the use of a visual shader graph programming language, but custom nodes for the shader graph can be created with HLSL programming language. As it is a game engine, it allows exporting the completed utility as a separate executable without the need to install the editor itself. Use of the path-traced renderer OctaneRender is possible, though limited, since rendering is only allowed inside the editor while the gameplay simulation is not running.

Unity is a proprietary game engine, which is, like the Unreal Engine, also used for machine learning simulation. The scripting language used for creating applications for Unity is C# and currently has no integrated visual scripting options outside of proprietary plugins. The engine also allows writing shader programs using a variant of the HLSL language, which also provides access to motion vector data. Direct access to the source code is limited, and no modifications are allowed by the license. OctaneRender can be used for path-tracing of scenes inside the Unity editor and can be used to record gameplay for later rendering in a standalone application.

	Pros	Cons
Blender	Established - FlyingThings3D	Not targeted for application creation
	Fully open source	No proper motion vector access
	Raytracing support	Large modifications required
Unreal Engine 4	Established - CARLA, AirSim	No proper motion vector access
	Blueprint and C++ scripting	Limited documentation
	Source available	Difficult for newcomers
	Raytracing using OctaneRender	Limited shader programming
Unity	Scripting using C#	Source code not available
	Good documentation	
	Raytracing using OctaneRender	
	Shader programming using HLSL	

Table 4.1: A Comparison of different considered platforms to base the generator on

We have selected Unity as the platform to develop the application on, mainly because of more straightforward access to motion vectors and better integration with third party path tracing renderer OctaneRender.

4.2 Software Design

The project consists of two separate parts: A Unity plugin designed to generate ground truth data and a set of scenes useful for training neural networks. The figure 4.1 shows which parts of the system are responsible for each output.

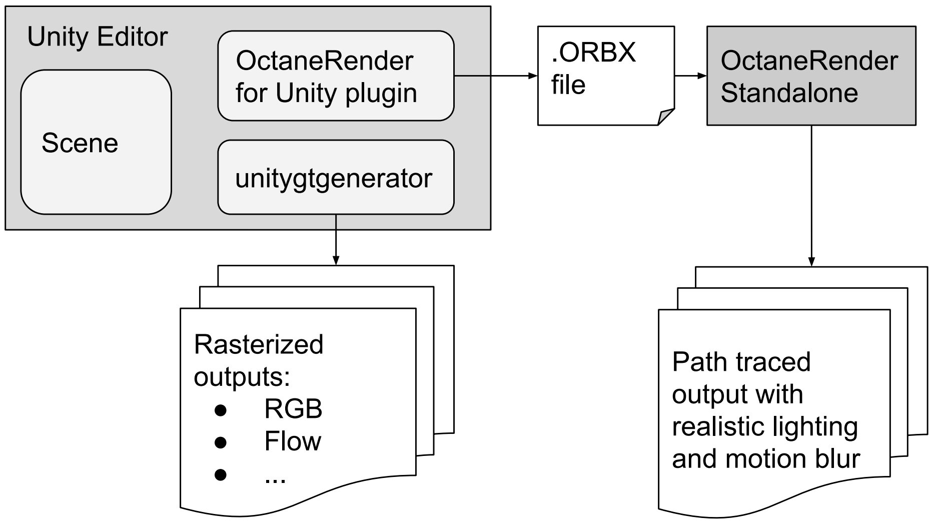


Figure 4.1: Diagram of data output sources.

4.2.1 Ground Truth Generation System

Our primary focus for the project is creating a system for generating realistic images augmented with additional metadata. We propose a Unity plugin called *unitygtgenerator*, which provides such capabilities. The plugin itself is composed of several C# scripts. A scene set up for ground truth generation contains a single instance of a *GT Manager* component and each camera set up for generating ground truth data has an instance of the *Unity GT Gen* component attached.

The *GT Manager* component contains the code managing the entire dataset generation. It manages scene-wide settings, such as the frequency and time when during the simulation we generate ground truth data, and configures objects in the scene for easier tracking. The saving of images themselves is triggered for all
*Unity GT Gen*s at the same time by the *GT Manager* and the dataset always contains the ground truth data from all cameras for each frame.

The *Unity GT Gen* component manages which type of ground truth data the system should generate. It contains a list of different outputs the camera will generate and their respective settings. Some cameras can have a different set of ground truth outputs prepared for different use cases (e.g. a camera can be set up to preview the current state of the scene, but as its view isn't planned to be used while training, only a limited number of ground truth outputs for it can be generated). Each different type of outputs is then implemented as a separate class. The type of generated data the camera attached component outputs is configurable via a human-readable file outside the scene description, so even people with no knowledge of how to modify the Unity scene should be able to configure their desired outputs.

The classes generating individual outputs all build on a standardized interface, which allows a simpler definition of new output types. Before implementing different possible ground truth outputs, we should have a strong grasp on which augmented image data are currently being extracted from images using state-of-the-art algorithms and how they are represented. We target generation of all these systems:

- **RGB camera output**, containing the simulated view of a camera,
- **Segmentation outputs (object, category)**, assigning each pixel into a category based on the object that occupies the given pixel,
- **Motion segmentation mask**, labeling each pixel that belongs to a moving object in scene,
- **Optical flow (backward and forward)**, labeling for each pixel where the point represented by the pixel is located in the previous or next frame of the dataset,
- **Occlusions**, labeling for each pixel whether the point represented by the pixel is visible in the previous or next frame of the dataset
- **Depth and normal map outputs**, describing the distance of each point from camera and its normal,
- **Specularity and transparency map**, possibly with other information about the materials of points visible in the scene,
- **Camera parameters**, describing the camera calibration and position in the scene..

The interface includes a standardized way to call the constructor, with additional settings given using a JSON object, and a way to expose default settings for the given output. It also exposes a save function, which the main *Unity GT Gen* script triggers at the end of each frame, which will save the data about the current frame.

An additional output is a path-traced output using the third-party OctaneRender for Unity plugin, with which the plugin interacts. This way we will be able to generate ground truth rasterized outputs relatively quickly, and will be able to provide high quality realistic camera view, which includes effects that are hard to emulate using Unity's rasterizer, such as motion blur or depth of field. The internal structure of the plugin can be seen in figure 4.2.

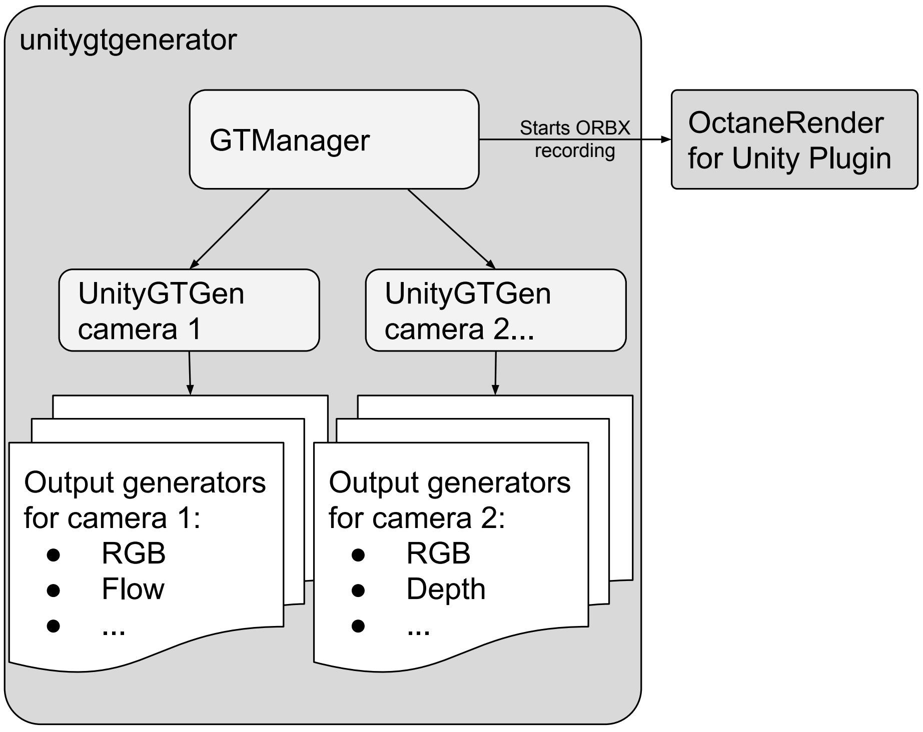


Figure 4.2: The generator consists of one GTManager and one UnityGTGen instance for each camera. Each UnityGTGen then manages its own list of classes responsible for generating ground truth outputs.

4.2.2 Scenes for machine learning

The *Unity GT Gen* component is a plug and play component, and should be be compatible with any camera in any already existing Unity scene. However, for some networks, a specialized scene might prove more useful – for example, the CrowdFlow dataset targets on visual crowd analysis. We propose several simple scenes.

CTUFlyingThings is a simple scene that is very similar to the already existing FlyingThings and FlyingChair3D datasets. It contains a random assortment of models flying around in space. The difference from the datasets mentioned earlier is that the models are not intersecting each other but bounce off each other. Because the models can be user-selected, it should be easily possible to modify the scene to generate datasets benefitial for specific uses (e.g., a dataset for tracking of purely flat objects).

Together with Toyota Research Lab at the Czech Technical University in Prague, we propose several other more complex and naturalistic scenes in the *CTUDriving* project. The scenes are targeted at computer vision for autonomous vehicles, and contain small road networks with randomized buildings, vehicle paths, and densely populated parking lots.

Chapter 5

Implementation

In this chapter, we describe the implementation of the ground truth data generator and its different outputs. We talk separately about the system responsible for dataset generation and the graphical user interface.

5.1 Backend

In this section, we describe the implementation of the system generating the ground truth data. We talk about the individual classes behind the generation and show short code snippets explaining the generation and encoding of ground truth information.

5.1.1 UnityGTGen

As a base for the implementation, we used *ML-ImageSynthesis* as an inspiration. Since Unity has released it under the MIT license, we were able to reuse some parts of the code, such as helper functions for optical flow visualization and shader code to create already supported data types. The author of the tool included all the different behavior in one uber-shader, which contained branches for separate outputs. As we wanted the output types to be easily extendable, we separated the shader code into several distinct shader files.

The main component handling the ground truth data creation is the *UnityGTGen* class. The component, attached to a camera, reads a human-readable JSON config file and creates instances of the data generating classes. Having the configuration stored outside the object itself (and outside the scene description) allows multiple cameras to share the same settings and sharing the configuration between different projects. The script itself has no hardcoded list of available outputs since the name of the classes representing outputs is listed in the config file. Instead, we use a feature of C# called Reflection, where the system searches for implementations of the outputs as classes in the `GTGenOutputs` namespace and instantiates them based on which outputs are mentioned in the config file.

As we want to be able to generate data from multiple cameras at once, there can be multiple instances of the script in the scene.

Output Types

All different data outputs inherit from the base class `OutputInterface`. This class describes all public methods and variables, which the UnityGTGen script uses when generating data.

All output types are required to have the following public members:

- A constructor which takes the current camera and config as parameters,
- `defaultConfig` JSON string, which describes possible config values and their defaults,
- `requires` property, which returns a list of outputs and their configuration, which the current output requires,
- `jsonconfig` JSONObject property, which describes the current config of the output,

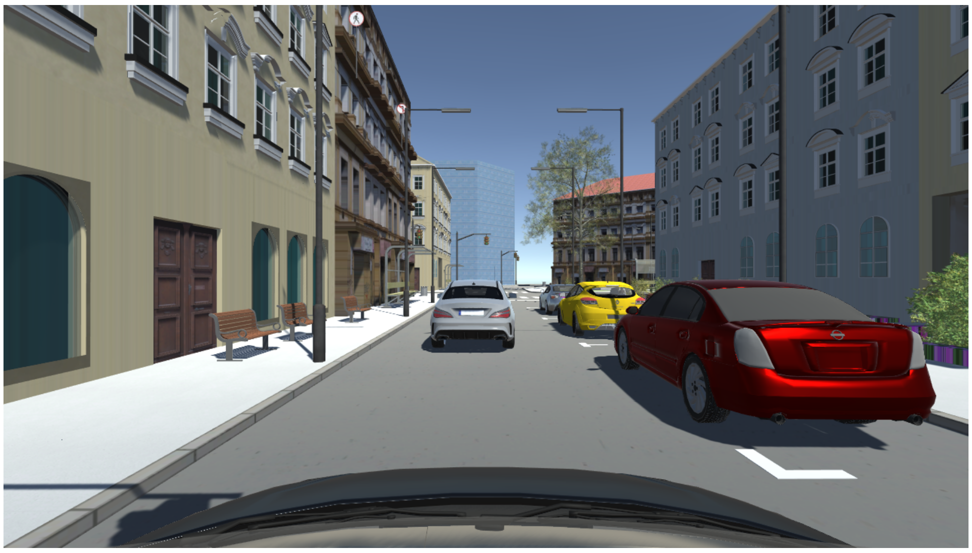


Figure 5.1: An example of the RGBImage output, rendered using the Unity rasterizer.

- `Save()` method, called every frame when an output should be generated,
- `SceneChange(objectList)` method, called whenever the scene changes.

RGBImage output. The primary output type saves the camera view using all shaders set in Unity, thus generating a lit, shaded output. As an additional configuration, the user can disable post-processing or shadow mapping for this output. The default unity shader are based on Physically Based Rendering, and so the rasterized output can be often used as an approximation of the realistic view of the camera, although of lower quality than path-traced images. An example of a view from the RGBImage output can be seen in figure 5.1.

Simple Outputs Using Replacement Shaders

Unity Camera objects have an option enabling the replacement of shaders using the `Camera.SetReplacementShader()` method. The specified shader then takes care of rendering the scene. Additionally, separate subshaders for different render types can be defined to treat opaque, transparent, or cutout materials differently. For these outputs, a class `SimpleShaderOutput` was created, from which outputs based on replacement shaders can inherit base functionality. The shaders themselves are relatively simple. Similarly to the implementation in *ML-ImageSynthesis*, we base them on an already existing Unity internal shader, `Internal-DepthNormalsTexture.shader`, with only differences in the behavior of the `Output()` function. Sadly, the meta-language Unity uses to describe shader program behavior, ShaderLab, does not easily support code reuse, and therefore most of the different shader implementations share large amounts of code outside the `Output()` function.

In this section, different outputs based on the principle of replacement shaders are described.

Depth output. This output generates an image with depth encoded and shares some code from the existing *ML-ImageSynthesis* shader. The shader reads the z-buffer and decodes the linear distance from camera by using the `COMPUTE_DEPTH_01` macro, which is included in Unity. Additionally, custom near and far plane distances can be set, differing from the planes' distances in the primary camera. The depth is either encoded directly (thus only spanning values of 0-255, seen in figure 5.2), or multichannel using the given conversion:

```
    float lowBits = frac(depth01 * 256);
    float highBits = depth01 - lowBits / 256;
    return float4(lowBits, highBits, depth01, 1);
```

Normal output. This output displays normal vectors in view space of all visible points and can be seen in figure 5.3. As normal vectors span values from $[-1; 1]$, conversion to the $[0; 1]$ range must happen:

```
float3 c = normal * 0.5 + 0.5;
return float4(c, 1);
```

Object, Mesh and Layer segmentation masks. Simple segmentation outputs can be rendered merely by adding a unique material property to each object in the scene. As some objects share materials, we add them using

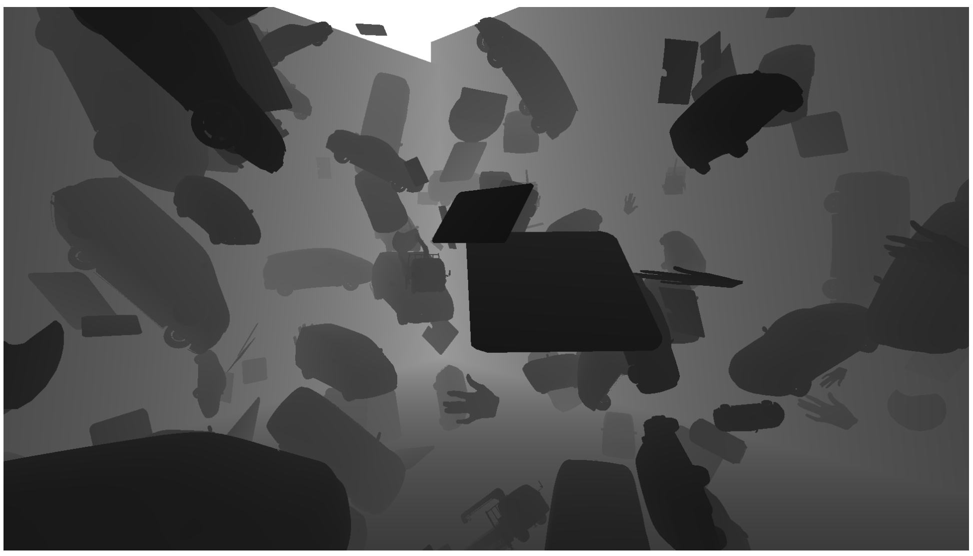


Figure 5.2: An example of the depth output, closer objects are drawn darker.

`MaterialPropertyBlock` feature of Unity materials, allowing multiple objects to share the same material with slightly different properties. We encode each unique segmentation label as a color property. We directly draw in a fragment shader and write the label color to a separate text file together with the label name.

This system is relatively simple and allows not only to be used to create object instance segmentation, but also to segment images based on different criteria. We can segment based on the mesh they are using (so two unique instances of the same shape share labeling), or even label the image based on user-defined criteria, such as the Unity layer in which the object is contained.

Motion Segmentation. For motion segmentation to be possible, we need to track whether each object moves during the simulation. To track information about each object in the scene, we attach a `SceneObject` component to every object in the scene, which then tracks whether or not the position or rotation of a given object changes between frames. This information is then during rendering sent to the shader, which then labels moving objects white and non-moving objects black. An image of this output is seen in figure 5.5.

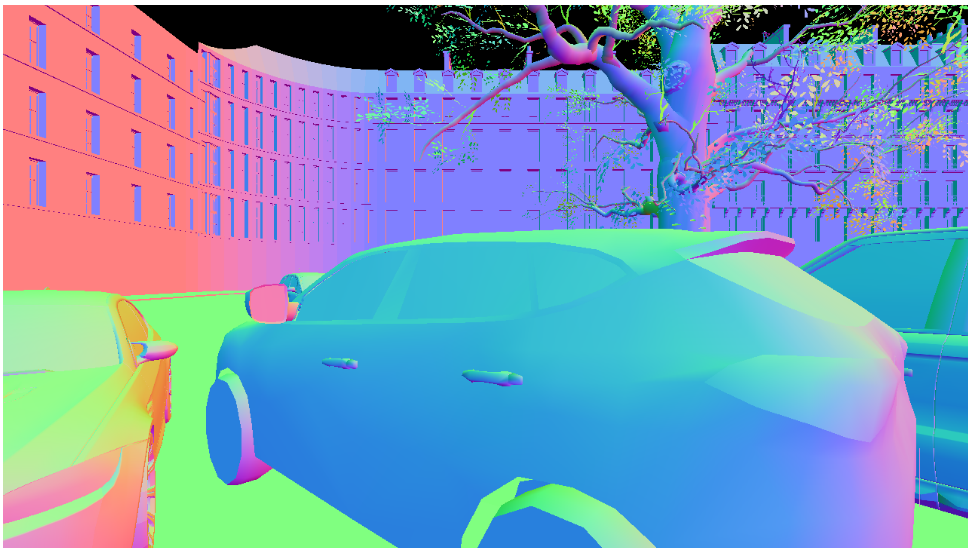


Figure 5.3: An example of the normals output. The XYZ vectors in view space are remapped to the RGB color space such that pure blue is a surface perpendicular to the camera view axis.

The current implementation works well with rigid objects but can cause issues for non-rigid objects, as we can consider deformation to be motion as well. As the term motion segmentation is relatively vague when talking about non-rigid objects, we have decided to only label objects as moving when their position or rotation changes.

Local and skinned coordinates. Information on 3D coordinates of a point in the local space of a rigid object is useful not only when calculating occlusions (as explained in section 3.3.7) and when tracking the movement of points in a more extended sequence of images. With object segmentation, tracking the position of an entire object in the image is easy. By encoding local coordinates, tracking the points on the surface of the object is also possible. Each point is then uniquely described by the object it belongs to and the 3D coordinates in the local space of the object.

The rendering pipeline uses the local coordinates of each point (defined in model space) during rendering, and therefore, it is easy to expose them. The only issue that can arise is from the limited way the fragment shader can output values: The shader program is only allowed to output a four-element vector, which the GPU then converts to a color image, such that the clamped

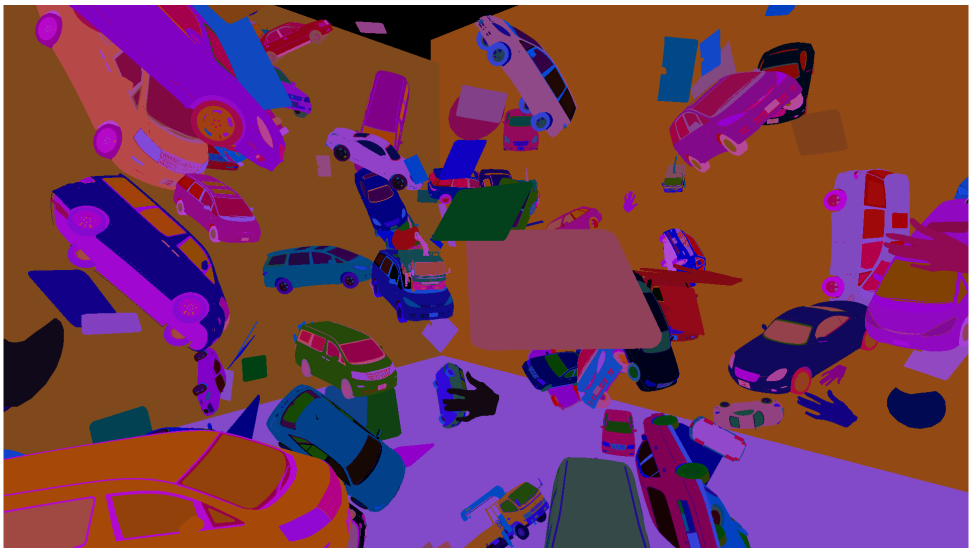


Figure 5.4: An example of an object segmentation map output.

output values of $[0, 1]$ are mapped to the minimal and maximal intensity of the represented color[1]. We calculate the axis-aligned bounding box of the mesh and map the coordinates so that the output values lie in the limited output range of fragment shaders when encoding as an image. The code below shows such mapping:

```
float4 mapColor(float3 input) {
    float3 retvec;
    retvec = (input - BoundsLower) / (BoundsUpper - BoundsLower)
    ;
    return retvec;
}
```

This approach works well to track points on objects that are rigid and do not change shape during the simulation. Points which are on objects that deform their shape between frames (such as on skinned meshes in Unity) do not have static local coordinates, so a different approach is needed. Instead of using the local model space coordinates after the deformation, we output the coordinates of each point *before* the deformation happened. That way, the value of the point is still unique for each point. If the deformation happens directly in a vertex shader, we can use the local input coordinates

[1]As we will discuss in the section on computing optical flow, it is possible to read the unclamped values outside this range, but with some performance issues.

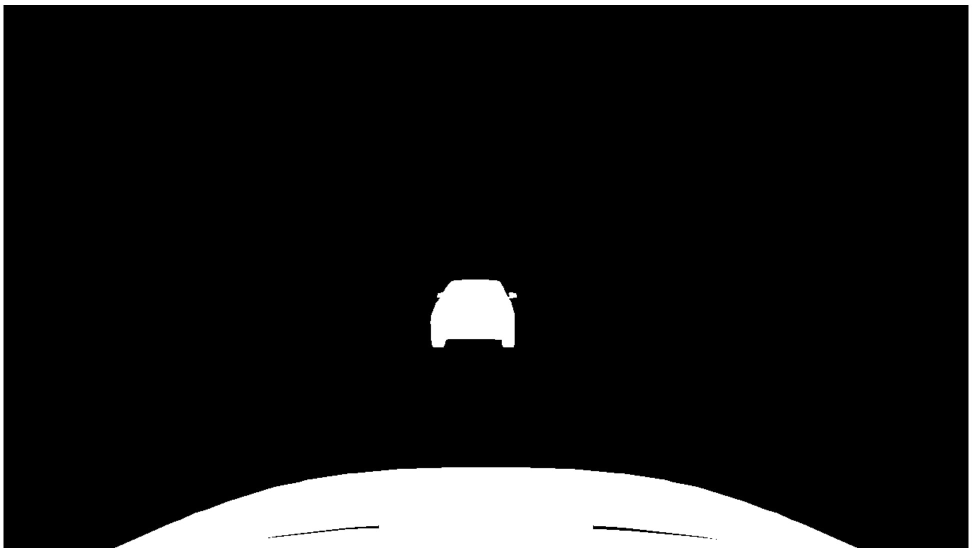


Figure 5.5: An example of the motion segmentation output. The scene is the same as in the RGBImage output example in figure 5.1

(mapped to a $[0, 1]$ interval) and output it unchanged to the fragment shader for displaying.

Unity's skinned mesh deformation happens outside the programmable vertex shader, and as such, our implementation works differently. When initializing, the mesh is modified, so that the vertices encode their model space coordinates both in the vertex position and vertex color information. This solution could cause issues if the material in the unmodified camera output uses the vertex color information. However, all the standard Unity shaders do not support vertex color by default, so this issue should not come up often. The output from the local coordinates output can be seen in figure 5.6.

Material information. Unity standard material uses multiple input textures to simulate the surface's realistic appearance. Therefore, some of the input textures themselves can be useful as ground truth information for different algorithms. For example, drawing the albedo texture directly on the mesh instead of shaded can help train delighting algorithms.

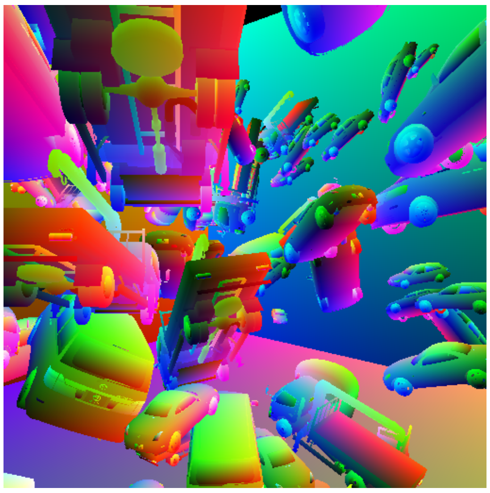


Figure 5.6: An example of the local coordinates output. Each point on an objects surface encodes the local coordinates of that point in object space.

As our primary goal is to generate ground truth data for computer vision and scene understanding, we provide a transparency map for objects in the scene, which can the training systems use as an ignore mask for places where a single pixel samples multiple different points in the scene. Such a mask is easy to generate, as we can directly use the alpha channel of either the texture or the color of the object in a shader.

```
fixed4 texcol = tex2D(MainTex, frag.uv);
float val = texcol.a * _Color.a;
return float4(val, val, val, 1);
```

Optical Flow and Motion Vectors

When implementing optical flow outputs, we implemented three approaches using different features of the Unity engine. The resulting approaches support different features and have different bit depths.

Unity motion vectors. Most game engines have an option to calculate optical flow from the current frame to the previous when calculating per-

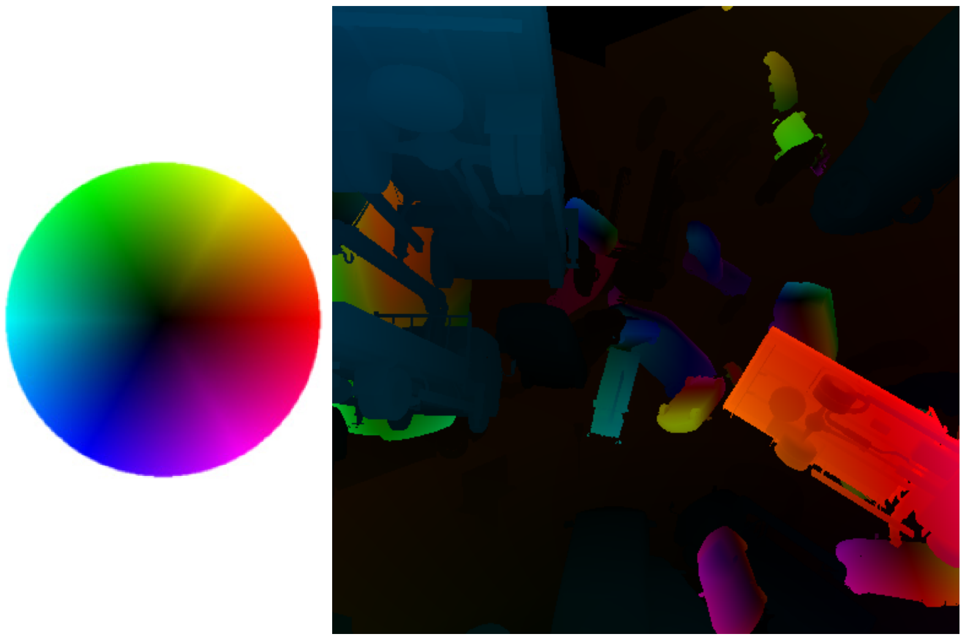


Figure 5.7: A color encoded output from the Unity motion vector based optical flow output. Color corresponds with the vector direction while the intensity represents the vector magnitude, based on encoding originally in [16].

object motion blur under the title *"motion vectors"*. In Unity documentation, *"motion vectors track the per-pixel object velocity from one frame to the next"*.

An implementation generating optical flow using this Unity feature already exists in [37], where the output is encoded using HSV encoding as described in [16]. The system encodes the directional vector using polar coordinates and encodes the direction in the hue and the vector magnitude in the color intensity. We reimplement the generation (seen in figure 5.7), where our system can also output pixel-precise movement in the Middlebury `.flo` binary file format [2]. To get the precise pixel values, as well as values outside the clamped $[0, 1]$ range, we copy the GPU texture to a CPU buffer and iterate over it in a script, directly writing to the binary file (instead of saving the texture to an image using a Unity builtin function). The creators of the `.flo` file format did not include any compression, so saving of the output is relatively slow. Because the computation works with relative screen coordinates in the range $[-1, 1]$ and `.flo` uses absolute pixel distance, we also remap the output to the screen resolution range.

The implementation using the default Unity motion vectors has a relatively significant issue of only describing one direction of the optical flow. If we want the flow in both directions, we need to implement the motion vector calculation on our own. Based on the description of motion vectors from Unity's documentation, one might also assume the output represents forward flow, but it does not. Unity calculates the vectors pointing from the current frame to the previous and flips their value. This limitation is not an issue when using motion vectors only to simulate motion blur, but for dataset training, the data would be incorrect when treated as forward flow.

Custom optical flow calculation. If we want to have the actual forward optical flow, one issue arises: when rendering the current scene frame, we do not know where the objects will be in a future frame. Instead of rendering forward flow for the current frame, we remember the state of the previous frame and render that frame's forward optical flow, pointing from the previous frame to the current one.

We already save the position and rotation of each object in the previous frame because of the motion segmentation output. Our next implementation of optical flow calculation uses a vertex shader to transform the rendered object both using the current and previous transformation matrix and calculates the optical flow in the fragment shader. By switching the current and previous matrices in the vertex shader, we can switch whether the shader renders optical flow from current to the previous frame, or from previous to the current frame Because perspective division only happens on the rendered vertices during the fixed rasterization step and the differently transformed vertices are not rendered directly, we must manually divide it by the w component in the shader:

```
float3 curpos = (i.curpos.xyz / i.curpos.w);
float3 prevpos = (i.prevpos.xyz / i.prevpos.w);
float3 motion = (curpos - prevpos) * 0.5;
//magnitude of 1 is the size of the screen
```

This solution works well with rigid objects, but not with non-rigid skinned meshes. We can set the skinned meshes in Unity to be double buffered so that that the engine calculates the motion vectors correctly. However, this buffer with vertex data from the previous frame is not available in the vertex and fragment shader when rendering an image directly.

Custom motion vector shaders. Luckily, we can fix the issue with non-rigid skinned meshes by using a different type of shader. Unity shaders written in ShaderLab can have different tags attached to them, describing for what purpose each subshader code snippet (written in HLSL) is. One assignable tag is a `LightMode` Pass tag, which describes in which render pass should the engine use the shader. For example, the Unity standard shader contains several subshaders for forward rendering – One which renders the scene with main and ambient light, and one which renders the scene additively with per-pixel lights, one render pass per light.

It is possible to override the default shader for calculating the motion vectors by assigning a `LightMode = MotionVector` Pass tag. When used, fragment shader does not draw to a color buffer attached to the camera, but instead to a RGHalf Motion Vector buffer. More importantly, when rendering skinned meshes, the previous vertex positions and several other (undocumented) properties, such as the previous transformation matrices of the object, are provided to the vertex shader.

We reimplemented the previous version of the shader to render into the Motion Vector buffer instead of the color image buffer. Then we can display the buffer by using the same shader as for rendering the original Unity Motion Vectors directly. This approach solves the issue, as non-rigid objects now have calculated both backward and forward optical flow.

Using the Unity managed Motion Vector buffer to draw the optical flow does have limitations when rendering into an image directly. When rendering to an image, we can decide the format and bit depth of the texture we render to. The Motion Vector buffer is hardcoded always to use the RGHalf format, which only contains two 16-bit channels of floating-point numbers. The implementation that draws to the image buffer directly uses RGBFloat, and therefore is more precise, due to using 32-bits per channel. It could output scene flow as well, as we can use the third channel to encode the z direction.

With the custom Motion Vector calculation, we can calculate the scene flow for non-occluded parts of the scene by using optical flow and depth output in conjunction, but the occluded parts of the scene are missing such information.

Another disadvantage when relying on the integrated Unity motion vector buffer is the fact that the optical flow has to always represent two neighboring frames. The solution in section 5.1.1 can be expanded to calculate optical flow between arbitrary frames, since we use our own matrices for the previous frame for which we calculate the flow.

Occlusions. Unity does not have built-in support for shadow volumes, and, as described in section 3.3.7, calculating occlusions by the use of shadow mapping would cause an error with an inconsistent size at the border of the shadows. We have decided to use a post-processing approach in calculating occlusions. Implementing the shadow volume algorithm in Unity should be possible. However, as we want to support objects with an alpha cutout texture, shadow volumes would not be ideal for all scenes, although it would result in more precise occlusion maps.

We read the generated optical flow of one frame and object segmentation and local coordinate outputs of two neighboring frames to generate an occlusion map. First, we handle object-object occlusion by comparing each point with the point shifted by the optical flow vector in the neighboring frame. Because of the raster sampling of the neighboring frame, we check the four closest samples of the shifted point and return the confidence value as

```
float testFour(sampler2D target, float2 target_uv, float4
    knownValue)
{
    float2 pixelpos_target = target_uv * TexelSize.zw;
    float2 uv_target_11, uv_target_12, uv_target_21,
    uv_target_22;

    uv_target_11.x = uv_target_12.x =
      (floor(pixelpos_target.x) + 0.5) * TexelSize.x;

    uv_target_11.y = uv_target_21.y =
      (floor(pixelpos_target.y) + 0.5) * TexelSize.y;

    uv_target_21.x = uv_target_22.x =
      (ceil(pixelpos_target.x) + 0.5) * TexelSize.x;

    uv_target_12.y = uv_target_22.y =
      (ceil(pixelpos_target.y) + 0.5) * TexelSize.y;

    float ID_occlusion = 0.0;

    ID_occlusion +=
        tex2D(target, uv_target_11) != knownValue ? 0.25 : 0.0;
    ID_occlusion +=
        tex2D(target, uv_target_12) != knownValue ? 0.25 : 0.0;
    ID_occlusion +=
        tex2D(target, uv_target_21) != knownValue ? 0.25 : 0.0;
    ID_occlusion +=
        tex2D(target, uv_target_22) != knownValue ? 0.25 : 0.0;

    return ID_occlusion;
}
```

Afterward, we (optionally) handle self-occluding objects by using the local coordinate output. Using this output, we do not need to use any custom sampling, as we can rely on GPU driven texture interpolation. As there might be some issues due to floating-point precision errors (both in the sampled local coordinate output textures and in the optical flow texture), we return a confidence value based on the distance between the two read local coordinates.

```
if (SelfOcclusion) {
    float d = distance(Loc_cur, Loc_prev);
    self_occlusion = clamp(pow(d,2) * 4, 0, 1);
}
```

When we have both the object-object and self-occlusion confidence values, we conservatively use the maximum value of both outputs as the final occlusion map value, as the mask will most likely be used to label set of pixels which we ignore during the learning process, and we do not wish to learn from any occluded pixels. An example of the occlusions output is visible in figure 5.8.

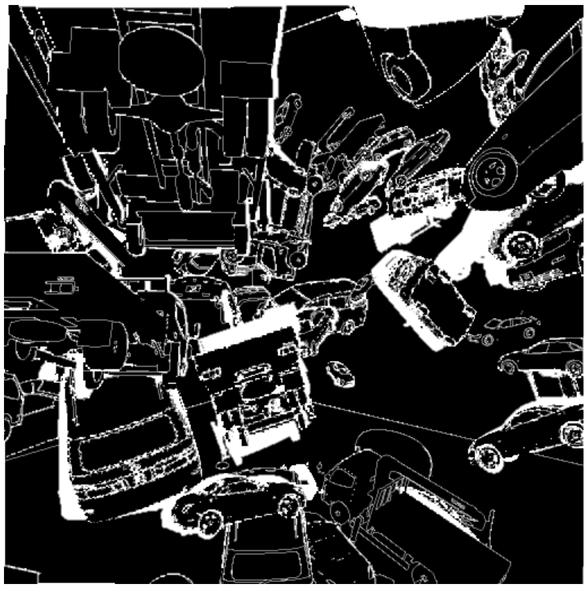


Figure 5.8: The occlusions output marks all pixels with a confidence rating whether or not the pixel is visible in the next or the previous frame.

Text Based Outputs

To have full scene understanding, we not only need per pixel image information, but additional per-object information can be of use as well. As such, we provide several non-image based outputs that help with understanding the scene itself.

Camera parameters. We expose the camera parameters of the camera associated with the `UnityGTGen` component in a JSON file generated for each frame. The file contains information about the image (such as the image resolution), intrinsic camera parameters (such as the projection matrix and field of view), and extrinsic parameters (such as the view matrix and world space position and rotation).

Tagged object info. When we want to expose information on scene objects, an issue can arise if there are many objects in the scene. Instead of outputting information on all objects in the scene, we should allow users to select which objects should output additional information. For this, we use a tags system and only information on objects with a specific tag assigned are outputted.

Unity has a built-in system for assigning tags to objects in the scene, but the system is limited to only allow one tag per object, so we expand the `SceneObject` component to contain a list of user-assignable tags. The Tagger Object Info output then generates a JSON file for each frame, which contains information about all objects of a given tag in the scene (using both the Unity tags system and our custom tags). By selecting a tagged object, we also select all of its children in the Unity scene hierarchy.

For each object of a given tag, we first check whether its renderer bounds are within the camera viewing frustum. If the renderer bounds are outside, Unity's rendering pipeline guarantees that the object itself is not visible in the image, and we do not provide additional info about it. If the object is within the viewing frustum, we provide information about the center of its bounds in screen space and the distance from the camera.

Afterward, we optionally provide information about the 2D and 3D bounding boxes. For both of these, a helper component `BoundingBoxHelper` is used. The component gets automatically assigned to all tracked objects. It calculates a 3D axis-aligned bounding box and a convex hull of the object, which we then use to calculate the 2D screen space bounding box at runtime. In some cases, we want to align the bounding box differently or disable the convex hull simplification (in case of floating-point precision errors). The automatic generation can be in those cases overridden by assigning the component to a model manually and changing its settings.

When outputting the 3D bounding box, no extra calculation is necessary, and we output the size of the bounding box directly. The position and rotation of the bounding box is transformed by the position and rotation of the object itself. For 2D screen space bounding boxes, we iterate over every vertex of the convex hull (or optionally the object itself, when hull simplification is disabled) and calculate the bounds as such:

```
var points = hull.mesh.vertices;
```

```
foreach (var point in points)
{
    var worldpoint = object.transform.TransformPoint(point);
    var screenpoint = m_Camera.WorldToScreenPoint(worldpoint);

    screenpoint.y = m_Camera.pixelHeight - screenpoint.y;
    xmin = Mathf.Min(screenpoint.x,xmin);
    ymin = Mathf.Min(screenpoint.y,ymin);
    xmax = Mathf.Max(screenpoint.x,xmax);
    ymax = Mathf.Max(screenpoint.y,ymax);
}
```

We also optionally include world space coordinates of the object and its local-to-world-space model matrix.

On Handling Transparency

As discussed in section 2.3, using a raster image to represent the ground truth information comes with a disadvantage of not being able to output ground truth on semi-transparent objects accurately. On pixels with transparent objects, ground truth would have to represent multiple objects at the same time. Our solution is to give each image output an alpha threshold setting. The user can set an alpha cutoff value, and the shader generating the output will then treat partially transparent objects with transparency below the threshold as fully transparent.

Realistic Path Traced Camera View

Additional to these outputs, we use the *OctaneRender for Unity* plugin to create a path-traced output with realistic lighting. The plugin includes a PBR Recorder utility, which can record a Unity scene running through the editor, either directly path-traced during gameplay as PNG images or saved in an intermediary ORBX file format for later rendering in the OctaneRender Standalone utility. When directly path-traced, the images lack

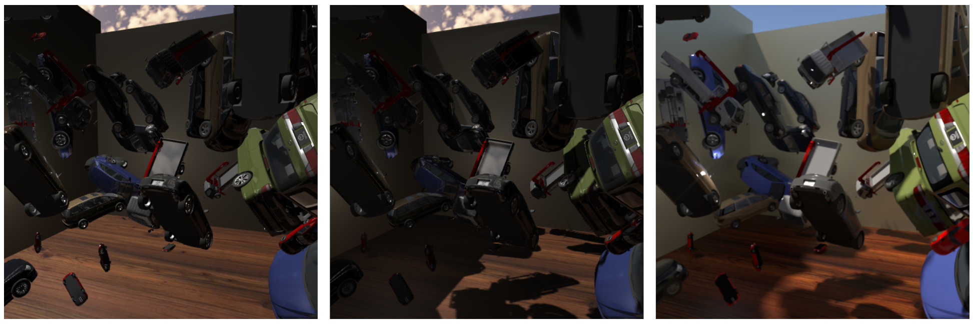


Figure 5.9: Image outputs with increasing levels of lighting realism. Left to right: Unity rasterizer with no shadows, Unity rasterizer with hard shadows created using shadow mapping, OctaneRender path-tracer. The path-traced image correctly simulates soft shadows and global illumination, which are missing in the rasterized images. It also includes an accurate motion and defocus blur.

motion blur. Therefore, we use the ORBX recorder mode to save the scene into a separate ORBX file which includes the animations and render the final results separately. Rendering to the separate scene description file also allows us to change certain parameters such as camera shutter speed or exposure after the simulation itself was completed. Comparision of visual differences with and without path-tracing can be found in figure 5.9.

When using the OctaneRender path tracer, we can also use more complex materials, for which we can more precisely define parameters such as transparency, subsurface scattering or index of refraction. Such materials, though, are not accessible in Unity and cause the transparency output to return incorrect values.

5.1.2 GT Manager

When using multiple cameras, some options are common between all cameras, for instance the time and frequency when the ground truth images should be generated. The `GTManager` class is a manager component, which takes care of configuring behavior which should be common between all outputs in the scene.

One such behavior is deciding when the outputs should be generated, and the duration of the output generation. We allow generation to be set to run constantly, in a single frame, or in a given frame count or time duration. The manager also defines common folder, in which all of the different camera outputs are saved.

The manager also takes care of additional behavior, such as warning the user that they're trying to generate a new dataset when the changes in the scene weren't commited to git. The user can start the ground truth generation inside the manager inspector GUI.

5.1.3 Interfacing with OctaneRender

Although the built-in Unity renderer can approximate real images relatively well, it still has some limitations, the main being the absence of realistic motion blur and depth of field simulation. To overcome such limitations and get a more realistic physically based image, we use an third party path-tracing renderer OctaneRender. The renderer has limited integration with the Unity game engine through the OctaneRender for Unity plugin, which is developed by OTOY, the company behind OctaneRender.

The plugin adds several new components inside Unity, one of which being the PBR Render Component, through which we can configure a path traced viewport to the scene. By default, there are two ways of getting path traced output from Octane, either by rendering directly in edit mode, or by rendering during the Unity play mode through OctaneRecorder component. As we want to render the scene during simulation, which happens inside the play mode, we can use the OctaneRecorder to set a duration in which the frames are rendered and render the images there.

Unity runs the simulation by stepping through discrete moments in time and when rendering directly inside the OctaneRender for Unity plugin, this approach makes it impossible to render motion blur. However, OctaneRecorder can, instead of rendering the scene directly, save the scene to an ORBX scene description file, which, when rendered separately, can include motion blur by interpolating the stored animations.

One of the limitations of the OctaneRender for Unity plugin is that the API lacks proper documentation and is only distributed as a single .dll file with no source code. Therefore, we used a C# decompiler inside the JetBrains Rider IDE to explore the publicly available methods of the plugin and directly call them inside GT Manager when rendering the ground truth outputs enhanced by OctaneRender.

Another limitation of the plugin is the fact that it only allows rendering from a single camera. We want to be able to generate the ground truth information for multiple cameras at the same time, and because of that, we need to replay the scene multiple times to record the OctaneRender managed cameras separately. Saving the motion of all the objects and recording the entire scene to separate ORBX files can create very large files with duplicate information. To avoid saving these large ORBX scene file multiple times, instead of replaying the entire scene, we only record the camera movement and record it in an empty scene, generating much smaller files. Then, we use the Lua scripting capability of OctaneRender Standalone to copy the camera movements from the small ORBX files into the main file containing all the scene information. This main file is then used for final rendering.

5.2 User Interface

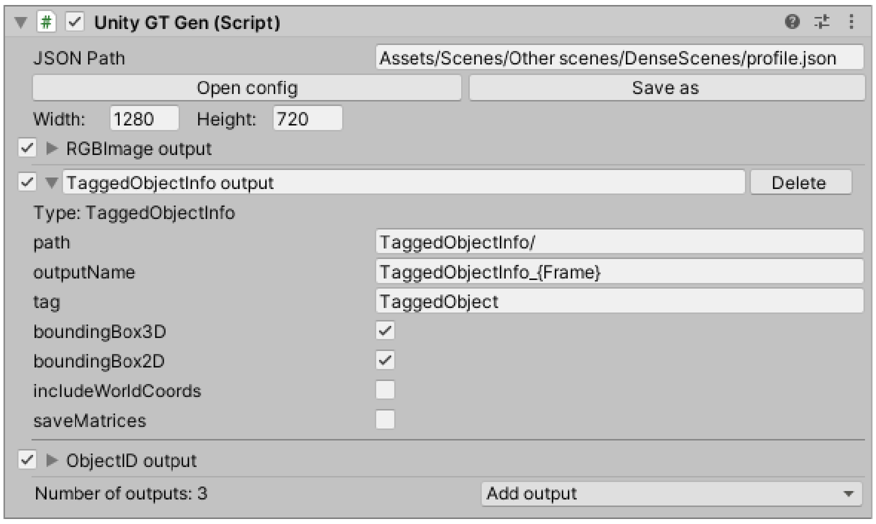
Workflow with the ground truth generator is based on workflow inside the Unity editor, which the tool extends. The Unity editor allows for creation of standalone apps, so we limit the generation only to the editor itself. The reason was that OctaneRender for Unity plugin, on which we rely when creating the path-traced realistic output, doesn't support rendering of the exported standalone, and so the ground truth generator would be severely limited when exporting the standalone app. Creating the tool as an editor extension instead of running standalone allows the user to always have full control over the scene for which they generate the ground truth.

The Unity editor allows the user to create a 3D scene with an object hierarchy and modify the object behavior by assigning components to them. The components represent C# scripts, and their public properties are by default visible in the user interface. We can override the default GUI by creating an `Editor` class with a `CustomEditor` attribute linking it with the script to which the editor belongs.

When creating the settings GUI, we first focused on the `UnityGTGen` component. The component, attached to a camera, uses a JSON based file to configure which outputs will the camera generate. The custom editor GUI then enables selecting which file will be loaded and manipulating with the file outside a text editor. The editor also allows editing the JSON information directly, such as adding which outputs the user will be generating, and allows configuring the outputs themselves. When adding new output types, the editor lists all implemented output types and displays them. When configuring a specific output, it also shows all available options by querying the implementation of the given output. The interface of the `UnityGTGen` component is visible in figure 5.10.

When a scene and its outputs are set up, the `GTManager` component is the main module the user interacts with. Global settings such as when the ground truth is generated can be set in the user interface, and the user can launch the generation itself by using a button in the component inspector. The component also takes care of recording the scene when using OctaneRender for Unity plugin, allowing users to generate path traced output with realistic lighting. The interface of the `GTManager` component is visible in figure 5.11.

Currently, there is no command line interface to run the tool, but as Unity supports executing C# scripts from the command line, such functionality could be added. Optionally, more feature rich API's could be also added, with functionality similar to UnrealCV [29]. Additionally, we have experimented with controlling the tool through the AutoHotkey automation system.


Unity GT Gen (Script)

JSON Path: Assets/Scenes/Other scenes/DenseScenes/profile.json

Open config | Save as

Width: 1280 Height: 720

✓ ▶ RGBImage output

✓ ▼ TaggedObjectInfo output | Delete

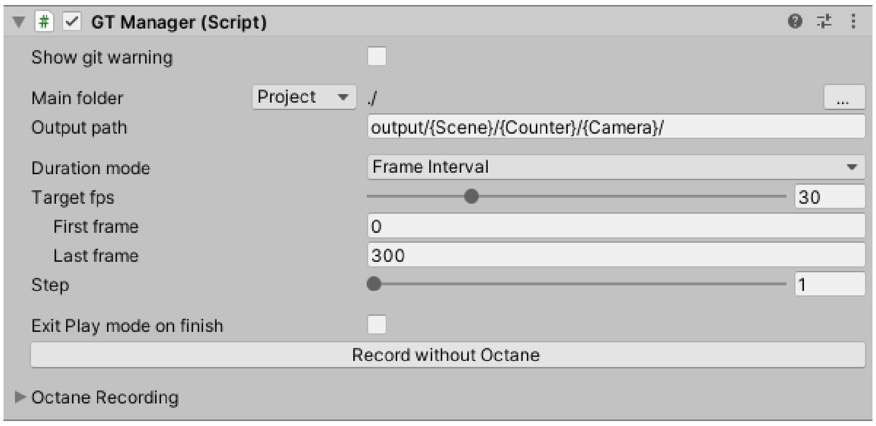
Type: TaggedObjectInfo

path	TaggedObjectInfo/
outputName	TaggedObjectInfo_{Frame}
tag	TaggedObject
boundingBox3D	✓
boundingBox2D	✓
includeWorldCoords	☐
saveMatrices	☐

✓ ▶ ObjectID output

Number of outputs: 3 | Add output

Figure 5.10: User interface of the UnityGTGen component allows the user to select which camera config file will be used and edit the file inside the editor.


GT Manager (Script)

Show git warning	☐
Main folder	Project ./
Output path	output/{Scene}/{Counter}/{Camera}/
Duration mode	Frame Interval
Target fps	30
First frame	0
Last frame	300
Step	1
Exit Play mode on finish	☐

Record without Octane

▶ Octane Recording

Figure 5.11: User interface of the GTManager component allows the user to set when the ground truth will be generated and start the generation. It also optionally allows launching the generation with the OctaneRender for Unity plugin.

Chapter 6

Results

In this chapter, we state the results of the project. We describe the functionality of the tool used for dataset generation from the users point of view, and finally, we present three example datasets, which we generated using the tool.

6.1 Using the utility

We have designed and implemented a utility to generate datasets which include camera view and various ground truth data as described by the implementation in chapter 5. Together with the tool, we created a user documentation in the form of a GitLab wiki, which we distribute both as a PDF and a separate HTML page together with the tool itself. In this section, we describe the usage of the tool to generate a dataset.

We distribute the utility as a plugin for the Unity game engine. When a user wants to create a dataset, they first need to make sure their scene works in the supported version of Unity 2019.3.5f1 and that the scene can run non-interactively.

6.1.1 Installing and Generating First Dataset

The plugin has two dependencies, the OctaneRender for Unity plugin, created by OTOY and available through the Unity Asset Store, and the Editor Coroutines package, distributed through the Unity Package Manager. After installing both these dependencies and setting up OctaneRender through its installer scene, we can set up the ground truth generation plugin itself.

Adding the plugin to a Unity project is straightforward. We add the folder containing the plugin to the Assets folder in the Unity project file structure, either by copying directly or by cloning from the git repository of the project.

After including the plugin folder, we add the GT Manager prefab from the plugin to the scene. The prefab, containing a game object with the `GT Manager` component attached to it, can be used to configure which frames from the scene run will be saved to the dataset.

To set up cameras in the scene so that they generate the ground truth, we attach the `Unity GT Gen` component to a camera object. Inside the inspector, we can set up the format of the generated ground truth. We set the resolution shared between all outputs from the camera and can add any of the implemented ground-truth outputs, which are described in section 5.1.1. For each output, we can then change its specific settings.

Finally, to generate the dataset itself without the use of OctaneRecorder, we open the `GT Manager` component in the Unity inspector and click on *"Record without Octane"*. If we want to include the path-traced camera view from OctaneRender, we need to set up the OctaneRender PBR Render Target to use the camera for which we generate the ground truth and use the OctaneRecorder window to set up the duration and recording mode of the rendering. Finally, we can use the `GT Manager` to record the scene together with OctaneRecorder by using the Octane foldout menu.

Setting Up OctaneRender

When setting up OctaneRender for Unity plugin, we add a PBR Render Target to the scene and in the Camera foldout select which camera in the scene will be rendered. Inside the PBR Render Target, there is a lot of different options, but some are important for the output to work as an extention of our dataset. Under *Film settings→Resolution*, the resolution (or at least its aspect ratio) should be the same as the resolution we set inside the ground truth generator. The render target settings can influence the speed of the rendering quite a bit, and lowering the maximum sample count and enabling denoising can create sufficiently realistic images for basic dataset creation (although higher sample counts are still useful).

6.2 Example Datasets

In this section, we show several small datasets created during the development of the tool. The datasets all target teaching and evaluating machine learning algorithms for computer vision, but are all intended for different uses. We do not treat them as final datasets for training algorithms (due to the low number of frames in each dataset), but more as examples of the types of data the tool can generate. A comparison of the ground truth data in our example datasets and a selection of currently available datasets is shown in table 6.1.

	RGB	Segmentation	Depth	Surface normals	Optical flow	Occlusions	Motion segmentation	Local coordinates	Stereo	3D bounding box	2D bounding box	Frame count
CTUFlyingThings	X	X	X	X	X	X	X	X		X	X	600
Dense	X	X	X	X						X	X	270
Traffic	X	X	X	X	X		X			X	X	700
FlyingThings3D [25]	X	X	X		X				X			$\approx 20,000$
KITTI [26]	X	X	X[1]		X[1]				X	X	X	$\approx 15,000$
COCO [23]	X	X[2]										$\approx 200,000$

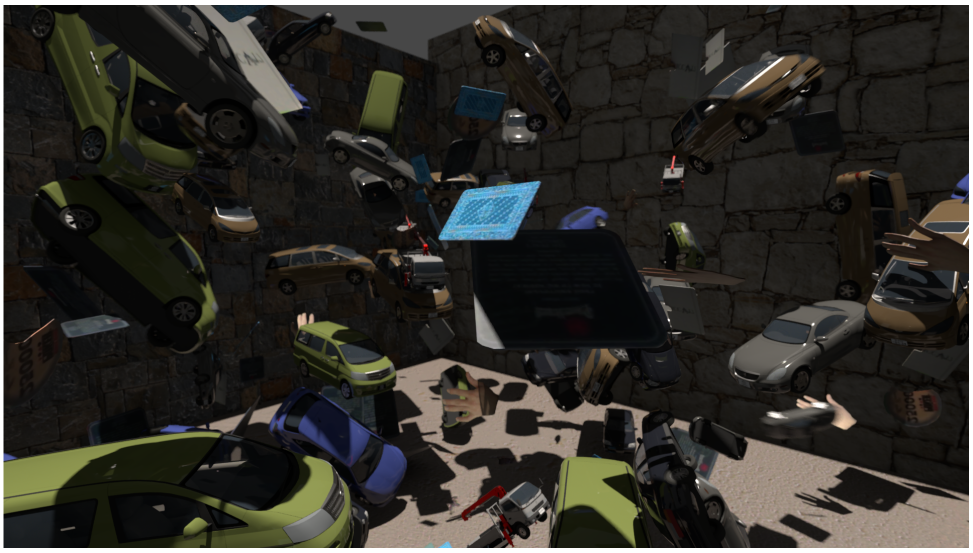
[1] Only sparse data available.

[2] Select categories are segmented by hand.

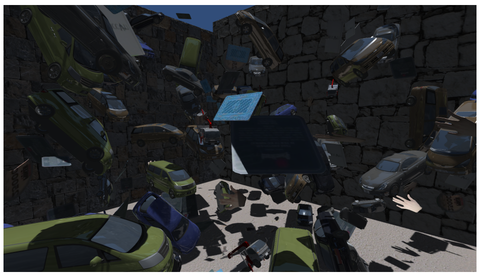
Table 6.1: Comparison of different ground truth outputs included in the example datasets and a selection of currently available datasets.

6.2.1 CTUFlyingThings

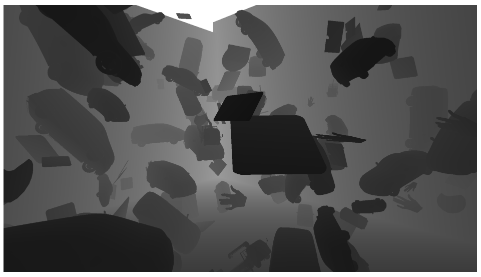
The first generated dataset, which we call *CTUFlyingThings*, is a dataset inspired by the FlyingChairs [10] and FlyingThings3D [25] datasets. In the original datasets, a set of random rigid objects float in front of the camera. The motion of the objects is always planar and the objects intersect each other. In our dataset, we use the rigid body simulation that is a part of the Unity game engine. Gravity does not influence the simulation, so they still fly around the view of the scene, but instead of objects intersecting, they bounce off each other. The objects' initial position, rotation, and scale is randomized and the camera moves on a hemisphere viewing the center of the simulated scene.



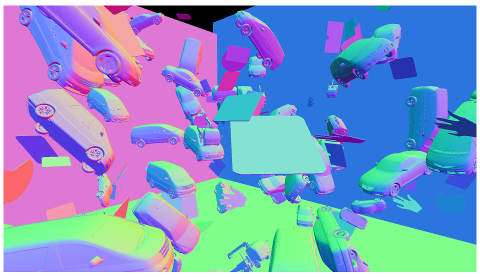
(a) : OctaneRender output



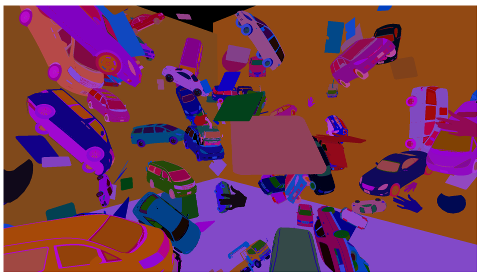
(b) : Unity RGB



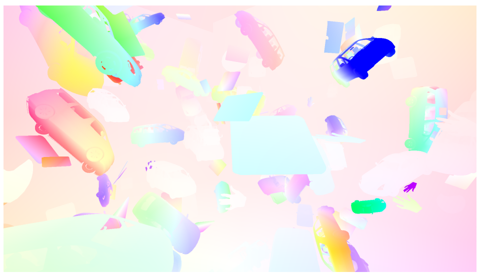
(c) : Depth



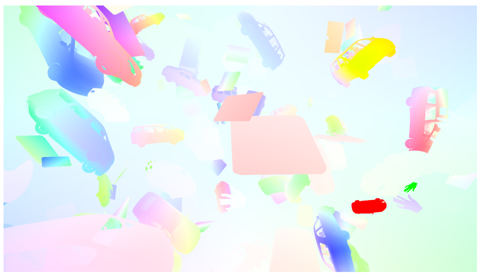
(d) : Normals



(e) : Object Segmentation



(f) : Backward optical flow



(g) : Forward optical flow

Figure 6.1: An example from the CTUFlyingThings-Example dataset with a selection of different ground truth outputs. The optical flow is visualised by color, hue encodes the direction and saturation encodes the vector size.

We generated a short sequence with a random distribution of objects and their motions. We provide an RGB view generated using the Unity rasterizer, instance segmentation masks, optical flow information based on modified unity motion vectors, normals, and depth information. We also include a realistic path traced view generated using OctaneRender and a Cryptomatte pass for all the images. The Cryptomatte output can be used to create a precise antialiased mask for each object in the image using Blender.[1] Example image outputs from the dataset can be seen in figure 6.1.

6.2.2 CTUDriving

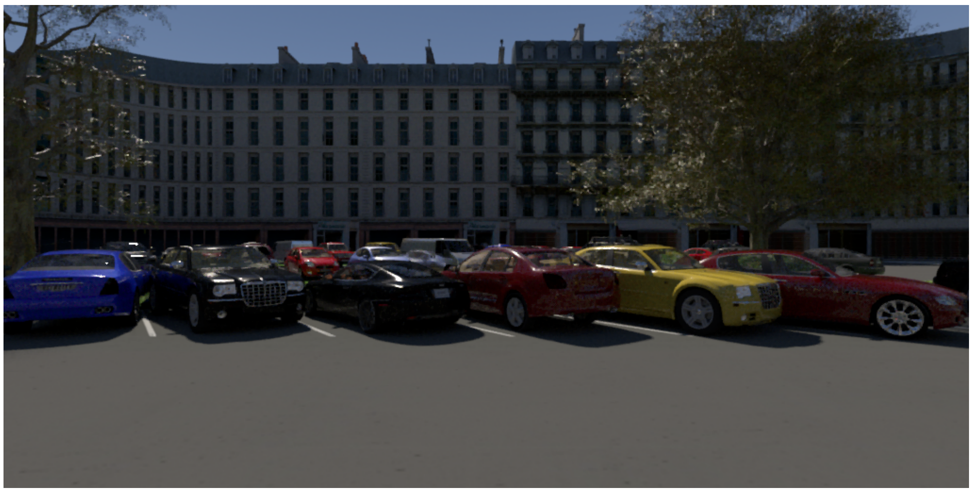
As part of the research at the Toyota Research Lab at the Czech Technical University, several other students have created a semi-procedurally generated city scene with traffic simulation. We have used a work-in-progress version of the scene to create two different example sequences. One focuses on traffic driving through the scene, and the other focuses on object detection and enumeration in dense scenes.

We base both of the scenes on a road network with buildings and road meshes generated procedurally using the CityEngine software, a modeling application designed to generate 3D urban environments.

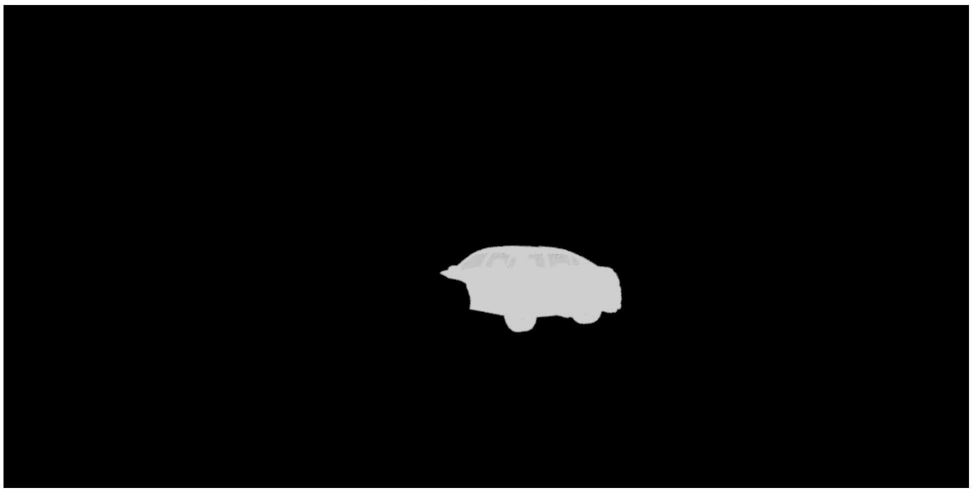
Dense Scene Dataset

The dense scene dataset shows a large parking lot, which is randomly populated with different cars. For each frame of the dataset, the camera selects a random position and rotation. For this dataset, we provide object segmentation map and 2D and 3D bounding boxes for all vehicles parked in the scene. We do not provide optical flow or other motion information, as the dataset does not contain an image sequence but many unique images instead. The scene contains around eighty cars and is targeted to train both

[1]We can automate the process of Cryptomatte extraction by using a plugin created by Jonáš Šerých, which is available at `https://gitlab.com/serycjon/cryptomatte_export/`



(a) : Path traced OctaneRender output



(b) : A mask of a single car created from the CryptoMatte output

Figure 6.2: The view of the Dense scene dataset from the CTUDriving scene

detection and density estimation. The dataset contains 290 images, There are several publicly available datasets for car density estimation based on images from real-life traffic cameras, but none of them as far as we know contains 2D bounding boxes for partially occluded vehicles. Example image outputs from the dataset, together with a mask that can be extracted from the CryptoMatte output is seen in figure 6.2.

Traffic Dataset

The example traffic dataset is a sequence of 700 frames, each of which shows several different from a car with four cameras attached to it. We generate RGB view using the Unity rasterizer, instance segmentation naps, motion segmentation, optical flow information, normals and depth information. We include realistic path traced view generated using OctaneRender including the Cryptomatte output. Example images from the dataset can be seen in figure 6.3.

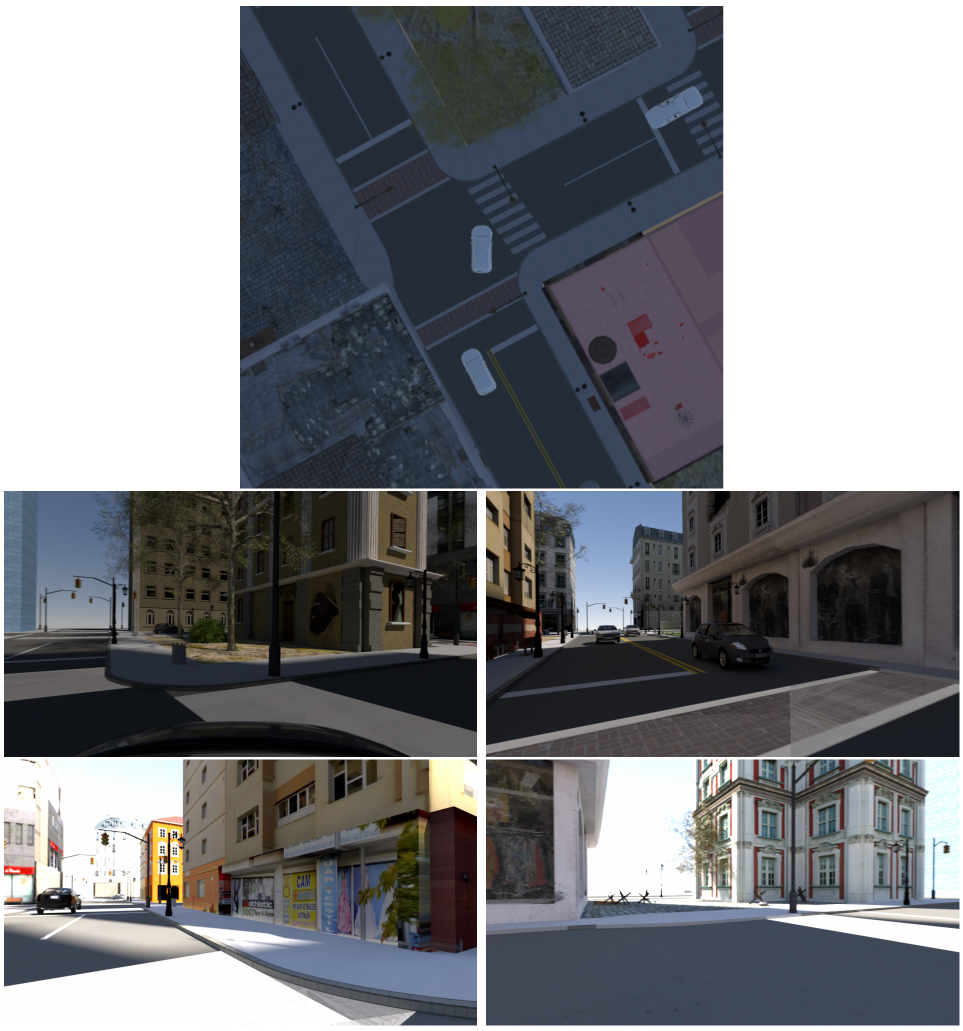


Figure 6.3: The traffic dataset includes four cameras attached to a car. The top image shows the current layout of the scene, while the four images below show front, back, left and right path-traced views of the scene.

This is currently the biggest dataset generated using the tool, and during the generation, some issues concerning the occlusions output were found. Therefore, we do not include the incorrectly calculated output. The issue is that we use the local coordinates output during the computation, but that output does not have enough of a floating point precision to correctly represent the large meshes of the buildings.

6.2.3 Performance

The hardware specification of the device the datasets were generated on are as follows:

- **CPU:** AMD Ryzen 7 3700X
- **GPU:** NVIDIA GeForce RTX 2080 Ti (2×)
- **RAM:** DDR4 64GB (4×16GB)
- **Storage:** Kingston SKC2000M8/2000G SSD
- **Motherboard:** Gigabyte X570 Aorus Pro

The performance of the utility heavily depends on which outputs we generate. When generating the images, the generation itself rund on the GPU and can be done in real-time. When saving the output, the texture must be read from the GPU memory and then encoded into a PNG or a different supported format using the CPU, which is a relatively slow operation. Using the Unity profiler, we can see that reading a 1920×1080 texture to the CPU memory can take between 40 to 200 milliseconds, possibly due to pipeline stall, as the CPU thread has to wait for the GPU to finish rendering. This issue could be optimized by batching all the rendering commands and all the memory read commands to run separately so that the pipeline stall is less likely to happen.

When encoding the images located in CPU memory and saving them to an SSD storage, we use the Unity included `EncodeToPNG()` function. The performance of this function is a bottleneck, as it can take around 100 to 200 milliseconds to store one FullHD image. Encoding optical flow information in a *.flo* file is also speed inefficient, as the file format uses no compression and the file can take up over 16 megabytes per one FullHD frame. If we want to include both forward and backward flow and simulate the scene at 30 frames per second, we get almost 1 gigabyte of data for one second of simulation. This value is greater than the write speed of most commonly available storage devices, and so a different format to represent optical flow might be necessary.

The most time-demanding part of generation occurs when we use the OctaneRender for Unity plugin to render realistic path-traced RGB camera outputs. The render time depends on the number of samples set in the render settings and the scene structure itself. When rendering, the plugin first imports the Unity scene hierarchy and materials and converts them to the internal representation used by OctaneRender. If there are many unique materials used in the scene, this can take several minutes, but it happens only once before the render starts. Rendering of individual frames can then run, which can take several seconds or even minutes per frame. We can speed up rendering by limiting the number of samples and enabling AI denoising inside OctaneRender, limiting the number of light bounces, lowering the resolution or by simplifying the geometry of mesh lights.

6.3 Limitations

The tool has several limitations, which could cause problems when generating datasets from certain scenes in Unity. First, the project is built for the Built-In Render Pipeline inside Unity and we have not fully tested support for the two other render pipelines currently distributed with Unity (the Universal Render Pipeline and the High Definition Render Pipeline). Therefore, scenes originally built using those render pipelines may have problems while rendering.

There are some limitations in the current implementation of the shader based ground truth outputs. Because of the way how Unity compiles shaders inside the editor and allocates buffers, the first two to three frames of every sequence are invalid and show many serious errors. In our implementation, the Occlusions output currently does not support nonrigid skinned meshes. Rendering this ground truth output for objects such as humans is not possible. The output also works incorrectly when larger meshes (such as buildings in the CTUDriving Traffic example dataset) are present in the scene, as it relies on precise local coordinates, which are not available due to floating point precision issues.

The segmenting into user defined categories is also currently limited by segmenting based on Unity layers, and therefore is limited to 32 unique labels, as that is the maximum number of layers Unity supports. When rendering the forward optical flow, the values in the skybox are incorrectly flipped and show backward flow, though this can be fixed by flipping the values when parsing, as skybox is colored black in the object segmentation mask.

Some techniques used to optimize rendering of scenes can also break the ground truth outputs. For example, a common technique relies on replacing the models in the scene for less complex models when the objects are further away from the viewer (level of detail). Our implementation of segmentation would consider the two different models of the same object as entirely different objects, instead of treating them as one instance. Similarly, optical flow would be incorrectly calculated, as it operates with the model which is currently used.

Many of the limitations also come from the reliance on OctaneRender for Unity plugin to generate the path traced images. The plugin itself also supports only the Built-In Render Pipeline, and can often crash when set up incorrectly. Our current implementation of the multi-camera recording (which the plugin officially doesn't support) is a relatively complex and requires a lot of human intervention (for example, manually running a script to merge the different ORBX files).

An issue also comes up when rendering a scene with a large number of objects through the plugin. During the set up of some outputs (such as Object Segmentation), we assign each material in the scene an unique property block that contains some information for ground truth generation, which causes the OctaneRender for Unity plugin to import them one by one, taking much longer than when such output isn't generated. This could be solved by saving the scene simulation as and animation and recording the ground truth data separately from the OctaneRender output during subsequent replays of the scene.

Chapter 7

Conclusion

In this chapter, we recapitulate the project. We describe the achieved goals and possible future work on the project.

7.1 Project Summary

We introduced the problem and included a short motivation for the project. We described our goal of creating a tool simplifying synthetic dataset creation. We explored different existing datasets, both real-life and synthetic, and explored which tools are used to create different datasets and how the different ground truth data can be represented.

We introduced different subjects of simulation. We discussed the RGB camera simulation, briefly talked about the world simulation, and described different ways we can measure ground truth data inside the simulation. We recounted depth, normals, bounding box, segmentation, optical flow, occlusions, and camera calibration outputs. We discussed different ways the outputs can be understood, and how we, based on our understanding, could generate the outputs and what pitfalls can occur during the generation.

We selected Unity as a framework on which we built the tool used to generate datasets. We outlined the structure of the individual components the tool consists of and which parts are responsible for generating which ground-truth outputs. We described which ground truth information we target and specified a few example scenes that demonstrate the tool's usage.

We detailed the implementation of the tool in section 5. We recounted the generation of each ground truth type, including code snippets. We described the interaction with OctaneRender for Unity plugin and describe the graphical user interface of the tool.

We presented the resulting tool and described its primary usage. We then presented three example datasets, each targeted at a distinct computer vision task. We compared the generated datasets with a selection of existing datasets. Finally, we described the performance of the tool.

7.2 Future Work

We plan to further work on the tool, by implementing additional outputs, such as the amodal segmentation mask output discussed in 3.3.5 or stereo disparity and occlusions. We also wish to allow users to define segmentation by using the tags system, instead of segmenting only based on the objects in Unity hierarchy. We also hope to automate the interaction with OctaneRender better, so that no human interaction is necessary when generating scenes with multiple camera views. Finally, we plan on using the tool to generate datasets based on more complex scenes at the Toyota Research Lab and release the datasets as part of a computer vision benchmarking challenge. We also wish to release the tool as part of the dataset, hoping that more Unity scenes benificial for machine learning will be created using the tool.

Appendix A

Bibliography

[1] Badrinarayanan, V., Kendall, A., and Cipolla, R. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence 39*, 12 (2017), 2481–2495.

[2] Baker, S., Scharstein, D., Lewis, J., Roth, S., Black, M. J., and Szeliski, R. A database and evaluation methodology for optical flow. *International Journal of Computer Vision 92*, 1 (2011), 1–31.

[3] Barron, J. L., Fleet, D. J., and Beauchemin, S. S. Performance of optical flow techniques. *International journal of computer vision 12*, 1 (1994), 43–77.

[4] Bovik, A. C. *The essential guide to video processing.* Academic Press, 2009.

[5] Butler, D. J., Wulff, J., Stanley, G. B., and Black, M. J. A naturalistic open source movie for optical flow evaluation. In *European Conf. on Computer Vision (ECCV)* (Oct. 2012), A. Fitzgibbon et al. (Eds.), Ed., Part IV, LNCS 7577, Springer-Verlag, pp. 611–625.

[6] Cabon, Y., Murray, N., and Humenberger, M. Virtual kitti 2, 2020.

[7] Chen, W., and Mied, R. P. Optical flow estimation for motion-compensated compression. *Image and Vision Computing 31*, 3 (2013), 275–289.

[8] Crow, F. C. Shadow algorithms for computer graphics. *Acm siggraph computer graphics 11*, 2 (1977), 242–248.

[9] Dai, A., Chang, A. X., Savva, M., Halber, M., Funkhouser, T., and Niessner, M. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 5828–5839.

[10] Dosovitskiy, A., Fischer, P., Ilg, E., Häusser, P., Hazirbaş, C., Golkov, V., v.d. Smagt, P., Cremers, D., and Brox, T. Flownet: Learning optical flow with convolutional networks. In *IEEE International Conference on Computer Vision (ICCV)* (2015).

[11] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning* (2017), pp. 1–16.

[12] Everitt, C. Interactive order-independent transparency. *White paper, nVIDIA 2*, 6 (2001), 7.

[13] Friedman, J., and Jones, A. C. Fully automatic id mattes with support for motion blur and transparency. In *ACM SIGGRAPH 2015 Posters*. 2015.

[14] Gaidon, A., Wang, Q., Cabon, Y., and Vig, E. Virtual worlds as proxy for multi-object tracking analysis. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (2016), pp. 4340–4349.

[15] Geiger, A., Lenz, P., and Urtasun, R. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012), IEEE, pp. 3354–3361.

[16] Guzmán, P., Díaz, J., Agís, R., and Ros, E. Optical flow in a smart sensor based on hybrid analog-digital architecture. *Sensors 10*, 4 (2010), 2975–2994.

[17] Horn, B. K., and Schunck, B. G. Determining optical flow. In *Techniques and Applications of Image Understanding* (1981), vol. 281, International Society for Optics and Photonics, pp. 319–331.

[18] Jalal, M., Spjut, J., Boudaoud, B., and Betke, M. Sidod: A synthetic image dataset for 3d object pose recognition with distractors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2019), pp. 0–0.

[19] Jiang, F. Unreal optical flow demo. `https://doi.org/10.5281/zenodo.1345482`, Aug. 2018.

[20] Jiang, F., and Hao, Q. Pavilion: Bridging photo-realism and robotics. In *Robotics and Automation (ICRA), 2019 IEEE International Conference on* (May 2019).

[21] Kainz, F., Bogart, R., and Stanczyk, P. Technical introduction to openexr. *Industrial light and magic* (2009), 21.

[22] Li, K., and Malik, J. Amodal instance segmentation. In *European Conference on Computer Vision* (2016), Springer, pp. 677–693.

[23] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *European conference on computer vision* (2014), Springer, pp. 740–755.

[24] Markovich, S. Amodal completion in visual perception. In *Visual Mathematics* (2002), vol. 4, p. 15.

[25] Mayer, N., Ilg, E., Häusser, P., Fischer, P., Cremers, D., Dosovitskiy, A., and Brox, T. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)* (2016). arXiv:1512.02134.

[26] Menze, M., and Geiger, A. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)* (2015).

[27] Navarro, F., Serón, F. J., and Gutierrez, D. Motion blur rendering: State of the art. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 3–26.

[28] Pharr, M., Jakob, W., and Humphreys, G. *Physically based rendering: From theory to implementation, Third Edition.* Morgan Kaufmann, 2016.

[29] Qiu, W., Zhong, F., Zhang, Y., Qiao, S., Xiao, Z., Kim, T. S., Wang, Y., and Yuille, A. Unrealcv: Virtual worlds for computer vision. *ACM Multimedia Open Source Software Competition* (2017).

[30] Schröder, G., Senst, T., Bochinski, E., and Sikora, T. Optical flow dataset and benchmark for visual crowd analysis. In *IEEE International Conference on Advanced Video and Signals-based Surveillance* (2018).

[31] Shade, J., Gortler, S., He, L.-w., and Szeliski, R. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), pp. 231–242.

[32] Shah, S., Dey, D., Lovett, C., and Kapoor, A. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics* (2017).

[33] Sun, P., Kretzschmar, H., Dotiwalla, X., Chouard, A., Patnaik, V., Tsui, P., Guo, J., Zhou, Y., Chai, Y., Caine, B., Vasudevan, V., Han, W., Ngiam, J., Zhao, H., Timofeev, A., Ettinger, S., Krivokon, M., Gao, A., Joshi, A., Zhang, Y., Shlens, J., Chen, Z., and Anguelov, D. Scalability in perception for autonomous driving: Waymo open dataset, 2019.

[34] To, T., Tremblay, J., McKay, D., Yamaguchi, Y., Leung, K., Balanon, A., Cheng, J., Hodge, W., and Birchfield, S. NDDS: NVIDIA deep learning dataset synthesizer, 2018. `https://github.com/NVIDIA/Dataset_Synthesizer`.

[35] Verri, A., and Poggio, T. Motion field and optical flow: Qualitative properties. *IEEE Transactions on pattern analysis and machine intelligence 11*, 5 (1989), 490–498.

[36] Xiao, L., Kaplanyan, A., Fix, A., Chapman, M., and Lanman, D. Deepfocus: learned image synthesis for computational display. In *ACM SIGGRAPH 2018 Talks*. 2018, pp. 1–2.

[37] Zioma, R. Image synthesis for machine learning. `https://bitbucket.org/Unity-Technologies/ml-imagesynthesis/`. Accessed: 2020-08-01.

Appendix B

Content of the Accompanying Medium

- `thesis-Bubenicek.pdf` – the PDF of this document
- `latex\` - LaTeX source code (compiled in Overleaf with pdfLaTeX and TeXLive 2019).
- `src\` - Source code of the Unity plugin, install by copying inside the Assets folder.
- `wiki\` - Offline copy of the gitlab wiki documentation of the tool, including a version exported to html and pdf.
- `example-project\` - Example project with the CTUFlyingThings scene.
- `example-datasets\` - Small selection of the generated datasets. As the full datasets take up several gigabytes, contact me if you wish to access the full version (bubentom@fel.cvut.cz or tombuben@gmail.com).