Towards Synthetic Dataset Generation for Semantic Segmentation Networks

by

Samin Khan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering - Computer Software

Waterloo, Ontario, Canada, 2019

© Samin Khan 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Recent work in semantic segmentation research for autonomous vehicles has shifted towards multimodal techniques. The driving factor behind this is a lack of reliable and ample ground truth annotation data of real-world adverse weather and lighting conditions. Human labeling of such adverse conditions is oftentimes erroneous and very expensive. However, it is a worthwhile endeavour to identify ways to make unimodal semantic segmentation networks more robust. It encourages cost reduction through reduced reliance on sensor fusion. Also, a more robust unimodal network can be used towards multimodal techniques for increased overall system performance.

The objective of this thesis is to converge upon a synthetic dataset generation method and testing framework that is conducive towards rapid validation of unimodal semantic segmentation network architectures. We explore multiple avenues of synthetic dataset generation. Insights gained through these explorations guide us towards designing the ProcSy method.

ProcSy consists of a procedurally-created, virtual replica of a real-world operational design domain around the city of Waterloo, Ontario. Ground truth annotations, depth, and occlusion data can be produced in real-time. The ProcSy method generates repeatable scenes with quantifiable variations of adverse weather and lighting conditions.

We demonstrate experiments using the ProcSy method on DeepLab v3+, a state-of-the-art network for unimodal semantic segmentation tasks. We gain insights about the behaviour of DeepLab on unseen adverse weather conditions. Based on empirical testing, we identify optimization techniques towards data collection for robustly training the network.

Acknowledgements

I would like to thank my supervisor, Professor Krzysztof Czarnecki, for giving me this great opportunity of completing a research-based Masters degree at the University Of Waterloo in the domain of self-driving vehicles. His continuous support and guidance has been a cornerstone in the completion of this crucial chapter of my academic career.

I would also like to thank Professor Steven Waslander and members of the Waterloo Autonomous Vehicles Laboratory (WAVELab). Collaborating with WAVELab on various aspects of research has been a tremendous learning experience. Our URSA Dataset paper publication was made possible due to their continued support and collaboration. I also thank Rick Salay for his amazing help and guidance of our research for the ProcSy Dataset.

I thank my fellow lab buddies, Matthew Angus, Mohamed Elbalkini, and Buu Phan. They have each had significant contributions towards the joint research projects that we have been a part of.

I thank Michal Antkiewicz for his all his efforts in making sure WISELab is rolling like a well-oiled machine — whether it be the WISELab website, Slack, Basecamp, servers, hardware, or anything else relevant to the lab's needs. I would also like to thank Jean Webster for her consistent help with conference trip itinerary, reimbursements, and all things administrative.

Last but not least, I thank my fellow researchers at Waterloo Intelligent Systems Engineering Laboratory (WISELab). Working and socializing with them has been an amazing experience. I wish them all the best of the best in their current and future endeavours.

Dedication

This is dedicated to Sharmin Akhter and Mesbah Khan — my mom and dad.

Table of Contents

List of Tables ix

List of Figures x

Abbreviations xiv

1 Introduction 1

1.1 Subjective Perception 1

1.2 Semantic Segmentation 2

1.3 Research Motivation 3

1.4 Contributions 3

2 Background 5

2.1 Unimodal Semantic Segmentation 5

2.1.1 Artificial Neural Networks 5

2.1.2 Deep Neural Networks 6

2.1.3 Semantic Segmentation Networks 7

2.1.4 Semantic Segmentation Datasets 11

2.2 Procedural Modeling 12

2.2.1 L-systems 13

2.2.2 Shape Grammar 14

2.2.3 CityEngine . . . 15
2.3 Real-time Physics-Based Rendering . . . 18
2.3.1 Light as an Electromagnetic Wave . . . 19
2.3.2 Light as a Ray . . . 20
2.3.3 Conservation of Energy . . . 21
2.3.4 Light Interaction with Surfaces . . . 22
2.3.5 Shortcuts for Real-time Performance . . . 22

3 URSA Dataset **27**
3.1 Introduction to GTA5 . . . 27
3.1.1 Deferred Shading . . . 28
3.2 Ground Truth Annotation via Detouring . . . 29
3.3 Ground Truth Annotation via URSA Method . . . 30
3.3.1 Method Overview . . . 30
3.3.2 Amazon Mechanical Turk . . . 30
3.3.3 Data Generation . . . 33
3.3.4 Influence Factor Variations . . . 34

4 ProcSy Dataset **37**
4.1 CityEngine Workflow . . . 38
4.1.1 Data Priors . . . 38
4.1.2 Asset Generation . . . 44
4.2 Unreal Engine 4 Workflow . . . 47
4.2.1 Introduction to CARLA . . . 48
4.2.2 Depth . . . 49
4.2.3 Ground Truth Annotations . . . 49
4.2.4 Vehicle Instances . . . 51
4.2.5 Occlusion Maps . . . 52
4.2.6 Data Selection . . . 53
4.2.7 Influence Factors Variations . . . 55

5 Experimental Analysis **60**
5.1 Adapting to Cityscapes 60
5.1.1 Class Distribution 61
5.1.2 Closeness to Cityscapes 61
5.1.3 Insights on Fine-Tuning 64
5.1.4 Summary 67
5.2 Comparing URSA and ProcSy 69
5.2.1 Strengths and Weaknesses 69
5.2.2 Using High Fidelity Vehicle Geometry 70
5.2.3 Summary 71
5.3 Influence Factor Analysis 71
5.3.1 Effects of Environmental Influence Factors 72
5.3.2 Effects of Depth and Occlusion 76
5.3.3 Optimizing for Data Collection 77
5.3.4 Summary 79

6 Conclusion **80**

References **82**

List of Tables

5.1 Experiment results for section 5.1.2; CS columns are for reference, these are trained on Cityscapes training set . . . 63

5.2 frequency-weighted intersection over union (fwIoU) metric analysis for section 5.1.2 experiment . . . 65

5.3 Results of Fully Convolutional Network (FCN) fine-tuning experiments on Cityscapes (CS), Synthia (SY), Playing for Benchmarks (Playing For Benchmark (PFB)), and URSA . . . 66

5.4 Results of SegNet fine-tuning experiments on Cityscapes (CS), Synthia (SY), Playing for Benchmarks (PFB), and URSA . . . 66

5.5 Results of DeepLab v3+ fine-tuning iteration experiments on ProcSy datasets; fine-tune iterations are done with 10% of Cityscapes (CS) training set . . . 68

5.6 Comparison of metrics between URSA and ProcSy dataset generation methods 70

List of Figures

2.1 FCN predictive results at different fused configurations [45] 8

2.2 High-level diagram of FCN fusing [68] 8

2.3 Encoder-decoder architecture of SegNet [2] 9

2.4 Encoder-decoder architecture of DeepLab v3+ [6] 10

2.5 Modern blockbuster video games use procedural modeling techniques to conserve memory footprint [49]; image on the left shows an in-game render of a procedurally generated biome in Horizon Zero Dawn; image on the right shows a topographical density map axiom of an in-game area; axiom density tiles are approximately $\sim$4 mb/km^2, Horizon Zero Dawn's entire map size is roughly 13 km^2, and memory footprint for the game's procedural world map is about 52 mb — a miniscule number for a big-budget video game . . 13

2.6 Snapshot of 3 turtle graphics states of Sierpinski triangle L-system as defined by equation 2.2; the iteration step is indicated by n; the turtle starts its movement facing vertically from the bottom-left triangle corner; length of triangle sides = $1/n$ [84] 15

2.7 Five level of detail (LOD)-level iterations using computer-generated architecture (CGA) shape grammar for a skyscraper [56] 16

2.8 CityEngine procedural modeling pipeline [18] 16

2.9 Different categories of electromagnetic waves and their respective range of wavelengths [30] 19

2.10 Top: Spectral Power Distribution (SPD) of ambient white light on a clear noon day (D_{65}); Bottom: red, green, blue (RGB) dirac delta SPD that is visibly equivalent to D_{65} to human eyes [30] 19

2.11 Ideal reflection/refraction diagram of a light ray [30] 23

2.12 Visible surface roughness caused by microgeometry [30] 23
2.13 Stochastic specular and diffuse distributions in macrogeometry can emulate effects of surface microgeometry [30] 24
2.14 On the left is an environment cubemap [80]; on the right is the corresponding irradiance map generated using discretized irradiance Riemann sum equation 2.11 25
2.15 Example pre-filter environment map with multiple mipmap levels (courtesy of [80]); note that the information gradually gets blurrier at smaller mipmap scales 26
2.16 Pre-computed bidirectional reflectance distribution function (BRDF) integration map; saved into a two-dimensional lookup table as a function of $cos\theta$ and roughness factor [34] 26

3.1 Galileo Observatory is Grand Theft Auto V (GTA5)'s rendition of real-world Griffith Observatory [19] 28
3.2 Web-based interface exposed to Amazon Mechanical Turk workers to efficiently label road-going filepath, model, shader, sampler (FMSS) values that were captured from the game world [1] 31
3.3 Cost-benefit visualization of Amazon Mechanical Turk worker votes 32
3.4 URSA dataset generation pipeline 34
3.5 GTA5 weather configuration previews [83] 35
3.6 Example texture variations used in GTA5; left is used in summer weather configurations, right is used during winter 36

4.1 Generation pipeline for ProcSy dataset: (a) data priors are used for procedural modeling; (b) CityEngine is used to generate three-dimensional, procedural world map; (c) Unreal Engine 4 (UE4) and CARLA are used for realistic lighting and weather effects; (d) dataset generation through CARLA is controlled via Python-based scripting; (e) each frame of our dataset have the outlined images rendered [35] 37
4.2 View of the 3km^2 region-of-interest of Waterloo, Ontario for our ProcSy dataset as seen on openstreetmap.org online map [26]; Bathurst Drive and Colby Drive loops are captured in this region; the region also features a highway overpass (seen in red) 39

4.3 Building height data provided by City Of Waterloo [81] 40
4.4 Tree plantation data provided by City Of Waterloo [81] 41
4.5 Sample 3D tree and plant models that are used by CityEngine for populating environment with vegetation [17] 42
4.6 Mid-level zoom of Bathurst Drive as provided by Esri satellite imagery [16]; higher magnification allows for distinguishing lane-level features such number of lanes and width of lanes 43
4.7 Low-poly out-of-the-box pedestrian models provided by CityEngine 45
4.8 A sample of low-poly out-of-the-box vehicle models provided by CityEngine 46
4.9 A sample of high fidelity vehicles from GTA5 game world 47
4.10 ProcSy dataset sample frame: (a) RGB image, (b) GTID image, (c) depth map, (d) 1 occlusion map [35] 48
4.11 Visualization of increased precision of d around 0 and corresponding to the near plane as d approaches 1; note that ticks along y-axis are in alignment with corresponding ticks on the 1/z curve [58] 50
4.12 UE4 custom stencil buffer visualization showing asset tag values as predicated by folder organization; folders highlighted in yellow represent semantic classes that have tag associations in CARLA code modifications for ProcSy [15] 51
4.13 RGB channel packing of ProcSy dataset; R channel contains ground truth annotations (section 4.2.3); G channel contains vehicle model instancing (section 4.2.4); B channel contains edge detection 53
4.14 Path nodes graph in sandbox games such as GTA5 54
4.15 Binary mask representing road-going pixels (white) of the 3 km^2 map region-of-interest (figure 4.2) in top-down view [35] 56
4.16 Weather variations showing intensity levels in 3 categories — rain, cloud, and puddles; the variation ranges explored are 25%, 50%, 75%, and 100%; note that the scene is captured with the Sun positioned at a high-noon angle, hence causing a visible glare on the car near center of screen 57
4.17 Road scene frame approximately showing the Sun positions in and out of frame that are considered; red x's indicate Sun location and corresponding tuples show azimuth and altitude values used in CARLA [35] 59

5.1 Dataset distribution of 19 classes 62

5.2 Performance of model A and B with different factors (each row); here, due to space constraints, we only show samples at 100% level for each influence factor 72

5.3 mean intersection over union (mIoU) for each testing scenarios for two models, A and B; x-axis denotes the intensity level of a given influence factor in each scenario 73

5.4 intersection over union (IoU) values for 4 classes: person, sky, car and road; each row corresponds to each testing scenario (rain, cloud and puddle) and each column corresponds to each class 74

5.5 'a-e' show model's accuracy on vehicles according to occlusion level and depth; darker green color corresponds to higher accuracy; 'f' shows frequency of vehicles; scales for these plots are shown in color bars at the right. The color bar for the distribution plot represents the distribution density. . . . 75

5.6 Occlusion map of a vehicle showing 3 instances where the vehicle is fully or highly occluded; yellow indicates occluded pixels; red indicates visible pixels 77

5.7 mIoU values for each model of section 5.3.3 78

Abbreviations

AI artificial intelligence 2, 6, 33, 38, 40

AMT Amazon Mechanical Turk 4, 12, 30–33

ANN Artificial Neural Network 5, 6

API Application Programming Interface 6, 29, 49, 58

ASPP Atrous Spatial Pyramid Pooling 9, 10, 77

AV autonomous vehicle 2

BRDF bidirectional reflectance distribution function xi, 20–22, 26, 57

CGA computer-generated architecture x, 16–18

CNN Deep Convolutional Neural Network 6

CPU central processing unit 5

CUDA Compute Unified Device Architecture 6

CV computer vision 2, 3, 5

DEM Digital Elevation Model 42

DNN Deep Neural Network 3, 4, 6, 7, 60, 61, 63–65, 67, 68, 72, 80

FCN Fully Convolutional Network ix, x, 7–10, 61, 64–66

FMSS filepath, model, shader, sampler xi, 30–32

fwIoU frequency-weighted intersection over union ix, 64–68

GPGPU general-purpose computing on graphics processing units 6

GPU graphics processing unit 5, 6, 9, 18, 28, 29

GT_ID channel-packed ground truth annotations, vehicle per-model instance ids, and edge detections 52, 55

GTA5 Grand Theft Auto V xi, xii, 4, 12, 27–30, 34–36, 45–47, 54, 61, 64, 68, 69, 71, 80

HD high-definition 44, 46

IBL image-based lighting 24, 25

IoU intersection over union xiii, 73, 74

LOD level of detail x, 15, 16, 18, 46

LUT lookup texture 25

mIoU mean intersection over union xiii, 7, 9, 10, 63–68, 72, 73, 77, 78

MTS mesh, texture, shader 29, 30

ODD operational design domain 60, 61, 63, 64, 69

PBR physics-based rendering 21, 24, 51, 57

PFB Playing For Benchmark ix, 12, 29, 64–66

PNG Portable Network Graphics 49, 52

RGB red, green, blue x, xii, 19, 33, 48, 49, 52, 53, 55, 58

SPD Spectral Power Distribution x, 19, 20

SSS sub-surface scattering 22

SVM Support Vector Machine 6

UE4 Unreal Engine 4 xi, xii, 4, 37, 38, 47–52, 54, 57

UI user interface 30, 32, 57

Chapter 1

Introduction

As human beings, one of our most valuable faculties is the power of vision. Visual perception allows for interpretation of the world around us. We plan actions according to subjective realities as rendered by our perceptive abilities. Our subjective perception can be skewed by influence factors such as adverse weather and lighting conditions. During these moments of reduced visual quality, we rely more on our learned intuition and other senses to compensate.

1.1 Subjective Perception

Our subjective realities are shaped by experiences and knowledge that we accumulate since birth. Our understanding of the world around us varies from that of our next-door neighbour due to differences in individual experiences. For example, whereas some people may judge the smell of gasoline to be repulsive, I personally find the smell to be nostalgic because it serves as a reminder of one of my favorite pastimes — riding my motorcycle down a country-road on a warm summer evening.

In his 1974 journal article, "What Is It Like to Be a Bat?", Thomas Nagel posits the uniqueness of subjective experiences of organisms through the use of an extreme comparison between human beings and bats [52]. Sonar (used by bats) and human vision are two very different perceptual experiences. As human beings, while we can imagine what navigation via sonar might be like, we cannot objectively know what a bat's perspective is. This reasoning asserts that perception is inherently a subjective experience of the individual.

For self-driving vehicles, the task of visual perception is greatly simplified. The autonomous vehicle (AV) perception task is constrained to road-going scenes and identifying traffic semantics in a vehicle's surroundings. Yet, subjective perception still remains in this subset perception task as an intractable problem. On a day-to-day basis in reality, an AV has to objectively contextualize its surroundings in the midst of variational lighting conditions and adverse weather.

1.2 Semantic Segmentation

Given raw camera imagery, the artificial intelligence (AI) system in an AV is a blank slate. The system has no subjective understanding of the scene or objects that make up the scene. The perception stack of the AI has no prior knowledge or experiences to contextualize road scenes. To rectify this, over the past few decades, road scene understanding from camera imagery has developed into a few key branches of computer vision (CV) research. The 3 main branches in this regard are image classification, object detection, and semantic segmentation [78].

Image classification contextualizes an image as a sum of its parts. This is good for categorizing different classes of images such as high noon, sunset, or nighttime. Image classification is also used to identify what is in an image, irrespective of where the object may be within the image.

Object detection is the understanding of where objects are located within an image. In a road-scene, object detection would identify locations of vehicles and pedestrians. Regions where objects of interest exist are typically visualized with 2D bounding boxes in image space. Research also exists in generating 3D bounding boxes given additional scene data such as stereo imagery [43] or lidar point clouds [39].

The third significant CV research branch is semantic segmentation. This is the act of assigning class labels to all pixels in an image. For instance, in an image of resolution 2048x1024, full semantic segmentation would consist of labeling 2,097,152 individual pixels. Section 2.1 digs into the background of unimodal semantic segmentation. Unimodal refers to the scope of semantic segmentation research that is only reliant on camera imagery.

In AV domain, semantic segmentation enables tasks such as free-space estimation, whereby drivable and non-drivable areas are identified in image space. Pixel-level detection means that nuanced tasks such as pedestrian pose estimation can be attempted. Also, by understanding scene-level details from an image frame, the AI can pipe image

regions to more specialized CV algorithms for refinement. For instance, semantic segmentation can identify image regions with traffic lights and signs. These regions can then be passed along to a specialized traffic signal detection algorithm.

1.3 Research Motivation

Over the past few years, Deep Neural Network (DNN)s have achieved great results for unimodal semantic segmentation benchmark tasks (section 2.1). However, adverse lighting and weather conditions have become a sticking point for these networks. Semantic segmentation networks require consistent and reliable annotation data. As with most DNN architectures, these networks adhere to the "garbage-in, garbage-out" principle.

Ideal case scenario for research purposes requires perfectly annotated road scenes captured from the real world. The only way to accomplish this is by employing human labelers for the task. This has been done for certain benchmark datasets such as CamVid and Cityscapes with ideal daytime lighting and clear weather conditions [11, 3].

Human labeling of images with adverse weather and lighting is an error-prone task. Subjective perception of individual labelers differ significantly for scenes with noisy or dark data. This is a source of labeling bias in ground truth. Due to this, there is a scarcity of high-quality, ground truth labeled data of real-world adverse weather/lighting scenes.

Human-labeled real world data also lack repeatability, metadata, and quantification capabilities. It is next to impossible to capture a road scene in the real world multiple times where the only variation in the scene is weather and/or lighting. It is also very difficult to quantify scene influence factors such as the amount of rain, the amount of puddles, and percentage of cloud cover.

1.4 Contributions

Difficulty and scarcity of human-labeled ground truth annotation of real-world road-scene data (especially of adverse weather/lighting conditions, where subjective perceptions differ significantly) has led recent semantic segmentation research towards multimodal techniques of sensor fusion [57, 36, 79]. However, we believe there is great potential in exploring synthetic methods of generating semantic segmentation datasets.

Aside from the domain adaptation problem inherent to synthetic datasets [87], there are significant benefits that can be leveraged towards a deeper understanding of unimodal seg-

mentation network architectures. Synthetic datasets can produce perfect labels, metadata, and adverse weather/lighting conditions that are repeatable and quantifiable.

The primary contribution of this thesis work is in identifying a synthetic dataset generation method to efficiently study effects of influence factors on unimodal semantic segmentation DNN architectures.

Towards identifying a viable solution, two separate methods of generating semantic segmentation datasets are explored (chapters 3 and 4). Namely these methods are URSA [1] and ProcSy [35]. Benefits and drawbacks of these are summarized in table 5.6.

Human labeling is needed for aspects of URSA dataset creation. We utilize Amazon Mechanical Turk (AMT) for this. Our insights on the AMT interface is detailed in section 3.3.2. My primary contribution towards this project was in developing a labeling framework for human labelers. I also designed the data generation pipeline for URSA (figure 3.4).

In generating the ProcSy dataset, procedural modeling is leveraged along with usage of a physically-based rendering engine. Required background knowledge regarding these concepts is outlined in sections 2.2 and 2.3. My primary contribution was the entire data generation pipeline. I identified procedural modeling as a viable option for dataset generation. I generated the blueprint map for our experimentation and integrated the generated map assets into our UE4-based rendering ecosystem.

Various experiments are run on the generated datasets to gain further insight on factors that affect semantic segmentation network performance (chapter 5).

The layout of this thesis is as follows:

- Chapter 2 outlines unimodal semantic segmentation research landscape, procedural modeling paradign, and real-time physically-based rendering.

- Chapter 3 outlines the GTA5-based URSA dataset.

- Chapter 4 explores ProcSy — a novel workflow to replicate real-world operational design domains for data collection towards building a semantic segmentation reference dataset.

- Chapter 5 analyzes synthetic datasets and influence factors towards understanding effects on semantic segmentation networks.

- Chapter 6 concludes with a summary of the exploration towards improving semantic segmentation network robustness. Some observed shortcomings are outlined, and future directions for this research topic are speculated on.

Chapter 2

Background

2.1 Unimodal Semantic Segmentation

Semantic segmentation is the domain of CV research where all pixels in a given image frame are classified. Let alone human beings, this can be a very taxing image processing task even for sequential hardware such as a central processing unit (CPU). Labeling 2,097,152 pixels (2048x1024) sequentially is not a practical approach to the problem — especially if said task needs to be handled real-time in a self-driving car. However, graphics processing unit (GPU)-based developments in deep neural network architectures over the last 2 decades have been a boon towards image processing tasks such as this, since network architectures can be designed in a highly parallelizable manner.

2.1.1 Artificial Neural Networks

The advent of Artificial Neural Network (ANN) can be traced as far back as 1940s. In 1943, Warren McCulloch and Walter Pitts first formalized the idea of an artificial neuron [46]. Then in 1958, Frank Rosenblatt proposed his seminal work in modeling the perceptron [62]. Compared to the McCulloch-Pitts model of the neuron, the perceptron was more robust due to its use of real number inputs and weights instead of booleans. The design of the perceptron also allowed for learning the weights and biases as opposed to being hand-crafted. This concept generated a lot of hype in theoretical research of artificial neurons around the 50s and 60s. 'Connectionism' became a buzz-word in the scientific community.

In 1969, Minsky and Papert rather infamously emphasized that single-layer perceptrons cannot even model the XOR function [47]. This fact, along with the limited technology and

understanding of training multi-layer perceptrons [32] caused a general disenchantment of researchers in the field. There was an overall reduction in funding of AI research fields. In 1970s, this contributed towards the first AI-winter.

There came a resurgence in neuron-based AI research in the 1980s. Some of the world's top-tier gurus in the field gained their momentum in this decade - Yann LeCun, Yoshua Bengio, and Geoffrey Hinton to name a few [1]. The idea of back-propagation to train multi-layer networks was popularized in the research world with the seminal work of Rumelhart et al. [64] (back-propagation was first introduced by Bryson et al. in 1963 [4]).

The 90s saw a second waning of interest in ANNs. Hardware technology necessary for training ANN models was just not up to par, yielding to demotivating results. Around the same time, other machine learning approaches such as Support Vector Machine (SVM) gained a foothold in the scientific community [12].

2.1.2 Deep Neural Networks

In 1998, LeCun et al. published their seminal work on popularizing the Deep Convolutional Neural Network (CNN) [42] (CNNs were first proposed by Kunihiko Fukushima in 1980 [20]). ANN research saw a significant resurgence thereafter, and came to be known as Deep Learning.

Around the turn of the century, GPUs became more robust and programmable towards traditional computation tasks. Training of DNNs became feasible. DNN models trained by leveraging parallel computation capabilities of GPUs showed significantly better results than SVMs on classification tasks.

As GPU technology became more capable at handling traditional computation tasks in a parallelized manner, the process came to be termed as general-purpose computing on graphics processing units (GPGPU). In 2007, Nvidia Corportation began to invest in an Application Programming Interface (API), called Compute Unified Device Architecture (CUDA), to spear-head development of GPGPU programming using their hardware [67]. This turn of events was called the "big bang" of Deep Learning by Nvidia's CEO Jen-Hsun Huang [73].

Newer libraries such as Microsoft's DirectCompute [55] and Khronos Group's OpenCL [51] aim to make hardware-vendor-agnostic API layers. However, to this day, Nvidia and their CUDA architecture remains the dominant player in Deep Learning space.

[1] Commonly known as the fathers of Deep Learning Revolution, Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, jointly received the 2018 Turing Award for their significant work in the field.

2.1.3 Semantic Segmentation Networks

The task of semantic segmentation actually combines image classification and the localization aspect of object detection. With image classification DNNs, the deepest convolution layers carry the resultant values. This is not the case for semantic segmentation. This task is a balancing act requiring pixel-level classification from features of deeper-level convolutions, and localization of pixel predictions onto a full image space.

The typical approach of DNN-based semantic segmentation is to design encoder-decoder architectures. The encoder portion of these architectures is similar to classification DNNs. In classification, input images are downsampled through successive convolution layers. This process breaks spatial information in images, but increases feature-space understanding for classification categories. This same functionality is what enables pixel-level classification portion of the semantic segmentation task.

The task of the decoder portion of semantic segmentation networks is to map classifications from the encoders back to the full resolution image space. This is typically achieved by removing the fully connected tail layers of a classification DNN and inserting transposed convolution layers for upsampling to image resolution.

FCN

In 2015, Long et al. first leveraged the idea of using a modified classification DNN architecture for semantic segmentation [45]. Their FCN architecture replaces the tail end fully connected layers with fully convolutional layers to enable input images of arbitrary sizes. These fully convolutional layers are appended with an upsampling procedure to portray the final output in image space.

The FCN paper explores different configurations for the upsampling procedure. Long et al. note that direct upsampling of the final convolution layer (by using a transposed convolution layer with a stride of 32) produces rough results for object boundaries (figure 2.1). Instead, they propose using fused upsampling for refined object boundaries (figure 2.2). The underlying intuition is that shallower layers preserve more spatial information.

By training with different encoder backbones (AlexNet, VGG16, GoogLeNet [38, 69, 72]), Long et al. conclude that VGG16 achieves best mIoU. VGG16 is a bigger architecture than AlexNet and GoogLeNet, and hence the inference time is generally longer (∼150ms more than AlexNet and GoogLeNet). However with a VGG16 backbone, FCN mIoU is over 10% more than the others. This justifies the trade-off between speed and accuracy in using FCN-VGG16.

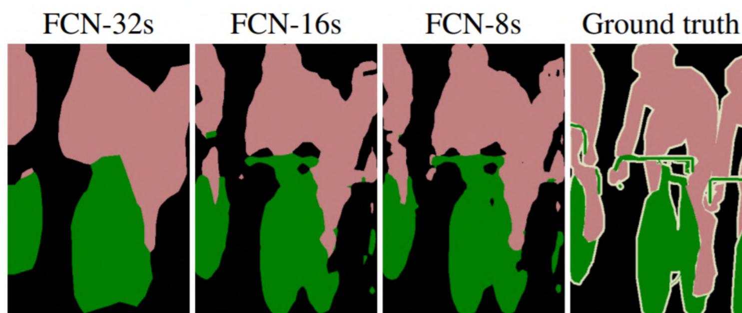


Figure 2.1: FCN predictive results at different fused configurations [45]

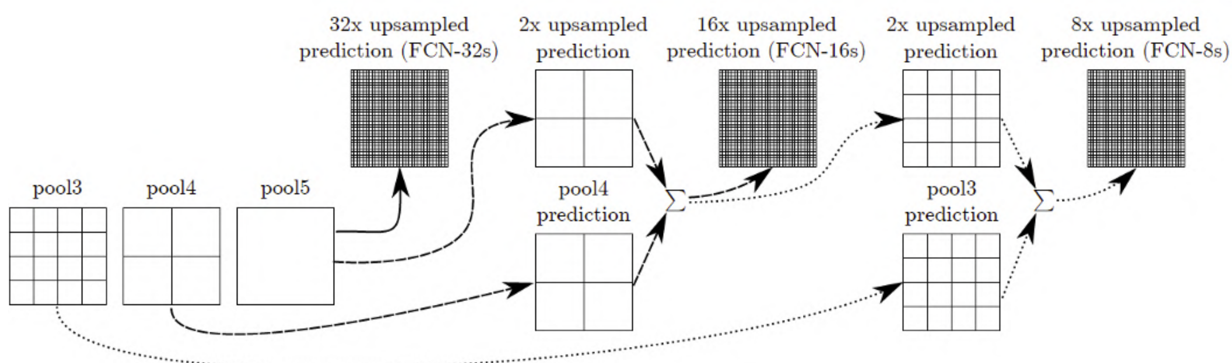


Figure 2.2: High-level diagram of FCN fusing [68]

SegNet

Later in 2015, Badrinarayanan et al. presented SegNet [2]. They use the same encoder-decoder approach as FCN. In fact, SegNet's encoder is topologically identical to VGG16. The novelty of SegNet is in the way upsampling is done. Their method achieves competitive inference times as well as most efficient memory usage at the time.

In SegNet, the decoder layers mirror the structure of the VGG16-based encoder layers (figure 2.3). Max-pooling indices from the encoder layers are stored. In the decoder layers, upsampling is not done using transposed convolutions. Instead, the stored max-pool indices from mirrored encoder layers are used to upsample from decoder layers. This negates the need for upsample layers to learn parameters. Hence, end-to-end training for SegNet requires fewer trainable parameters and converges faster than FCN.

Due to the unorthodox decoder architecture using max-pool indices, SegNet implemen-

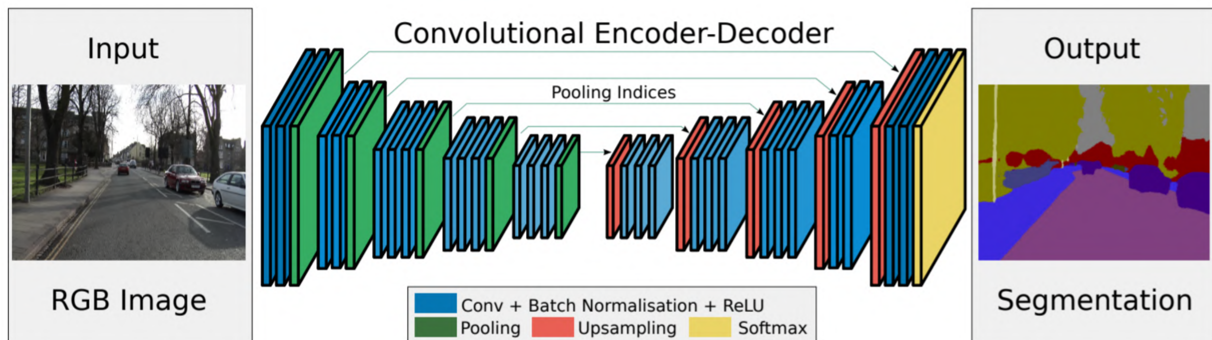


Figure 2.3: Encoder-decoder architecture of SegNet [2]

tations tend to have slightly slower (∼100ms) inference times than FCN. On the memory side, SegNet has around two-third the GPU memory footprint of FCN during training and inference. SegNet model size can be close to just 20% of FCN.

DeepLabv3+

Google's DeepLab has been a mainstay in semantic segmentation research since it was first proposed in 2014 [5]. The team has been iteratively improving on the model over the years. The latest version, DeepLab v3+ [6], is currently ranked amongst state-of-the-art for Cityscapes dataset's semantic segmentation task with a 82.1% mIoU score.

The success of the DeepLab network is credited to its usage of Atrous Spatial Pyramid Pooling (ASPP) architecture in conjunction with the aforementioned encoder-decoder architecture. The intuition behind spatial pyramid pooling is that objects should be seen at multiple scales by the network. Let's imagine that a training dataset contains some images with trains, but these trains exist at a distance such that they appear small in the image. During inference time, if a train is seen much closer to the screen, the network will perform poorly. With spatial pyramid pooling, incoming features are probed at multiple field-of-view rates. This captures multi-scale information during training, so the network is ultimately more scale invariant.

However, spatial pyramid pooling with multiple instances of the same architecture is expensive in terms of computation and memory. Atrous convolutions is a generalized version of the traditional convolution operation. A rate parameter is introduced to control the effective field-of-view, whereby a rate of 1 is equivalent to a traditional convolution. Atrous convolutions applied towards spatial pyramid pooling reduces the computation and

memory overhead.

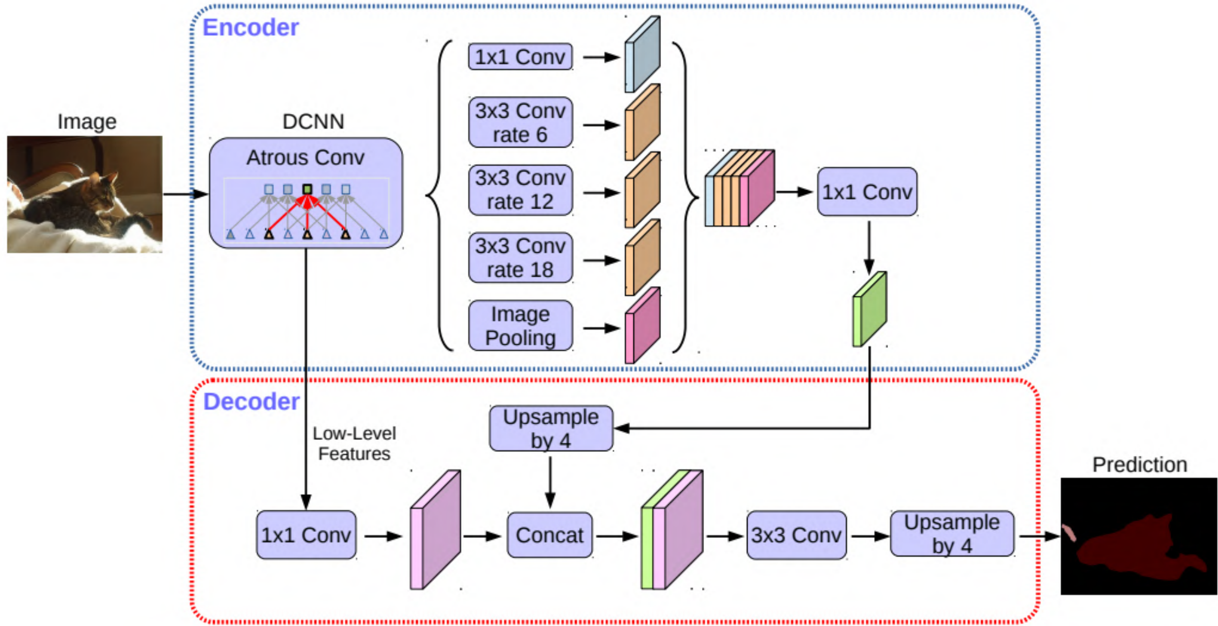


Figure 2.4: Encoder-decoder architecture of DeepLab v3+ [6]

Taking a page from inception modules [8], DeepLab also makes use of depthwise separable convolutions in place of max pooling operations to reduce the amount of computation. DeepLab uses a ResNet architecture modified with atrous convolutions as its feature extractor to retain more spatial context in deeper layers. ASPP and bilinear upsampling is used prior to the decoder layers. The decoder combines ASPP output (upsampled by 4x) with encoder features from a layer with equivalent spatial dimensions (figure 2.4). This is similar to FCN stitching shallower layer outputs for better object boundaries.

State-of-the-art DeepLab architecture uses a modified ResNet-101 or Xception encoder backbone to achieve high mIoU performance values. For our experiments in chapter 5, we opt to use ResNet-50 as this backbone allows for faster training iterations. This is sufficient for our experiments because we are interested in the relative metric differences rather than state-of-the-art mIoU scores.

2.1.4 Semantic Segmentation Datasets

Real-world Datasets

Standardized and publicly available datasets pertaining to semantic segmentation research have been around since 2008. Brostow et al. first introduced the CamVid Dataset [3]. This dataset is mostly collected around Cambridge, UK. Image frames are captured at 960×720 pixel resolution. Each frame in the dataset took an average of 60 minutes to be labeled by trusted human labelers. The dataset contains 701 finely annotated images with 32 different label categories. Only about 11 of these label categories are usable towards training neural networks due to the very limited size of the dataset. There is very little variation in the environments, lighting, and weather represented by CamVid dataset.

Introduced in 2015, Cityscapes Dataset was a significant step forward in terms of scale and quality of provided data [11]. The dataset is aggregated with road scenes from 50 different cities around Germany, Switzerland, and France. Cityscapes image frames are 2048×1024 pixels — an order of magnitude better quality than CamVid. Human labeling time for each frame was around 90 minutes on average. This dataset contains 5,000 finely-annotated and 20,000 coarsely-annotated images with 30 label categories. Like CamVid, the Cityscapes Dataset lacks quantifiable lighting and weather variations. Also, 5,000 images is still a relatively small amount of data.

Published in 2017, the Raincouver dataset [77] is the first public dataset containing rainy driving scenes at different hours of the day. This dataset contains only 326 finely-annotated. The dataset has been generated in such a way as to complement the aforementioned Cityscapes dataset.

Berkeley Deep Drive (BDD100K) [86] provides a finely-annotated dataset containing 5,683 images. Although equal in quantity to Cityscapes, this dataset has a lot more weather and lighting variations. However, the BDD100K dataset is known to contain labeling inconsistencies, especially in darker regions of images — a causation of subjective perception [65]. BDD100K also does not contain metadata to quantify the weather and lighting variations in its scenes.

Mapillary Vistas, published in 2018 [54], contains 25,000 finely-annotated, non-temporal images from various geographical locations and weather conditions. This dataset provides the annotation granularity of 66 different classes. However, similar to BDD100K, Mapillary Vistas provides no way to identify or quantify the weather and lighting variations presented in its scenes. Mapillary Vistas used the same sort of human labeling technique as Cityscapes, averaging at 94 minutes of labeling time per image frame.

Synthetic Datasets

The time, labeling cost, and inaccuracies of real-world semantic segmentation ground truth data have motivated researchers to explore dataset generation through synthetic means. To that end, promising results began to appear in 2016. Ros et al. published their SYNTHIA dataset [61]. The base dataset consists of 13,400 random, road-scene annotated images with various lighting and weather conditions, at a resolution of 1280×400 pixels. This dataset is generated using the Unity game development platform, so more road scene imagery can be generated on demand without additional annotation cost. In 2017, the dataset was appended with 2,224 new images representing a San Francisco-like environment [28].

Virtual KITTI, also released in 2016, is a re-creation of the KITTI dataset in a simulation environment [24, 21]. Like SYNTHIA, Virtual KITTI is also built with the Unity game development platform. Here, real-world KITTI video sequences are used as input to generate realistic-looking proxies in the virtual world. The dataset currently has over 21,000 frames [53], at a resolution of around 1242×375 pixels. This is comprised of 5 cloned worlds with 8 different weather and lighting variations for each.

While custom-built simulators provide flexibility and control over dataset generation, they suffer from a lack of detail and realism in comparison to those found in sandbox video games. Richter et al. performed a crowd-sourced (AMT), Turing-like experiment between SYNTHIA, Virtual KITTI, and their dataset (PFB) [59]. The experiment results showed that AMT workers found the PFB dataset to be more representative of the appearance of Cityscapes dataset than either SYNTHIA or Virtual KITTI.

PFB dataset was generated from the GTA5 video game world [59, 60]. Using a process called detouring (detailed in section 3.2), their research contains over 254,000 labeled frames, at a resolution of 1914×1052 pixels. To label 25,000 frames, their pseudo-automated labeler takes around 49 hours with some apparent labeling inconsistencies. Another drawback to this method of dataset generation is the challenge in decoupling the resource-shared optimization employed by the game world (for example, texture on side of a building wall can appear on a sidewalk). The bottom line is that GTA5 game world is not designed for perfect semantic segmentation data extraction.

2.2 Procedural Modeling

Procedural modeling is a blanket term in computer graphics for creating complex 2D and 3D patterns from a set of rules. L-systems, fractals, and generative modeling are all

examples of procedural modeling techniques. There are some unique advantages of using procedural modeling to generate complex geometry. For one thing, the procedure employed can be repeatable given a static initial seed. This can be great for storing a large amount of geometric information as a set of rules instead of actual mesh geometry. In fact, modern video games such as Horizon Zero Dawn does exactly this (figure 2.5).

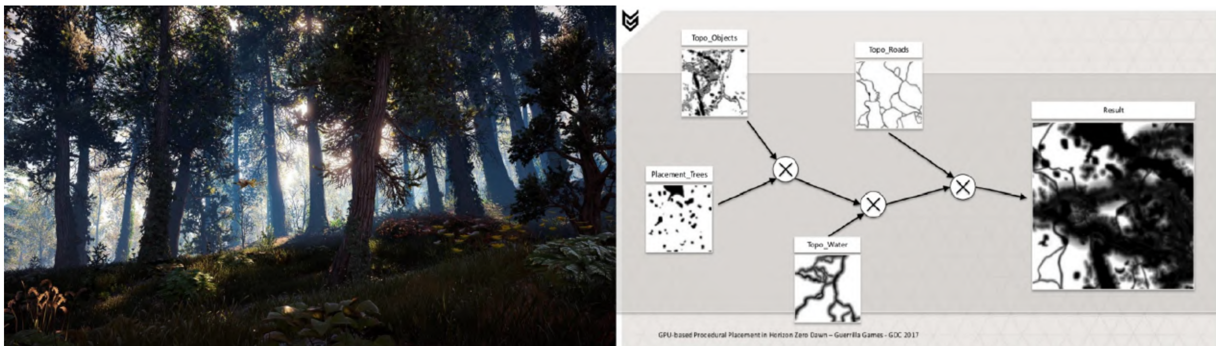


Figure 2.5: Modern blockbuster video games use procedural modeling techniques to conserve memory footprint [49]; image on the left shows an in-game render of a procedurally generated biome in Horizon Zero Dawn; image on the right shows a topographical density map axiom of an in-game area; axiom density tiles are approximately $\sim$4 mb/km^2, Horizon Zero Dawn's entire map size is roughly 13 km^2, and memory footprint for the game's procedural world map is about 52 mb — a miniscule number for a big-budget video game

Procedural modeling can also significantly reduce the time and human effort necessary to model a massive urban environment. To this end, our focus is on procedural modeling approaches that are derived from Lindenmayer systems.

2.2.1 L-systems

Lindenmayer system (L-system) is a type of formal grammar that lets us describe complex shapes and patterns through iterations that apply parallel rewrites to previous iterations. L-systems are used in mathematics to simplify the description of self-similar fractals. Famous structures such as the Sierpinski triangle can be described using elegant L-system definitions [48].

Aristid Lindenmayer was a 20th century theoretical biologist and botanist. He wished to mathematically model the development process of plant cells and structures. His seminal work in this field was formalized in the Journal of Theoretical Biology in 1968 [44]. This

became the foundation of L-system definitions. The system was later extended to have the representation ability of complex branch structures.

$$L = (V, \omega, P) \tag{2.1}$$

L-systems are generalized as the tuple in equation 2.1. V is the system's alphabet consisting of terminal and variable symbols. ω refers to the axiom — a sentence defined by symbols from alphabet V. P is a set of production rules that are used to mutate symbols in the axiom.

An L-system progression can be illustrated using the aforementioned Sierpinski triangle example as such:

$$\begin{aligned} V &= \{F, G, +, -\}, \\ \omega &= F - G - G, \\ P = \{F \longrightarrow F - G + F + G - F, G &\longrightarrow GG\} \end{aligned} \tag{2.2}$$

If we associate this L-system with turtle graphics [25], we can draw the Sierpinski triangle. F and G are variables indicating forward movement of distance proportional to inverse of the iteration step, where iterations are even numbers. Angle of the turtle is controlled by terminal symbols $+$ and $-$, defining $\pm 120°$ counter-clockwise respectively. With this understanding, figure 2.6 shows the result of iterating over the L-system using turtle graphics.

2.2.2 Shape Grammar

Shape grammars are formal grammars that are conceptually similar to L-systems. Used properly, shape grammars can be powerful mechanisms to facilitate architectural design workflows. Interpreting L-systems with turtle graphics (figure 2.6) is an example of geometric representation potentials inherent to these sort of formal grammars.

First proposed by Stiny and Gips in 1971 [71], shape grammars are a form of visual computation. When designing a shape grammar, the designer creates a system for design, which can then generate many parametrized variations.

Shape grammars differ from L-systems in some key ways. Instead of using language symbols to represent variables and terminals, shape grammars use shape primitives as the

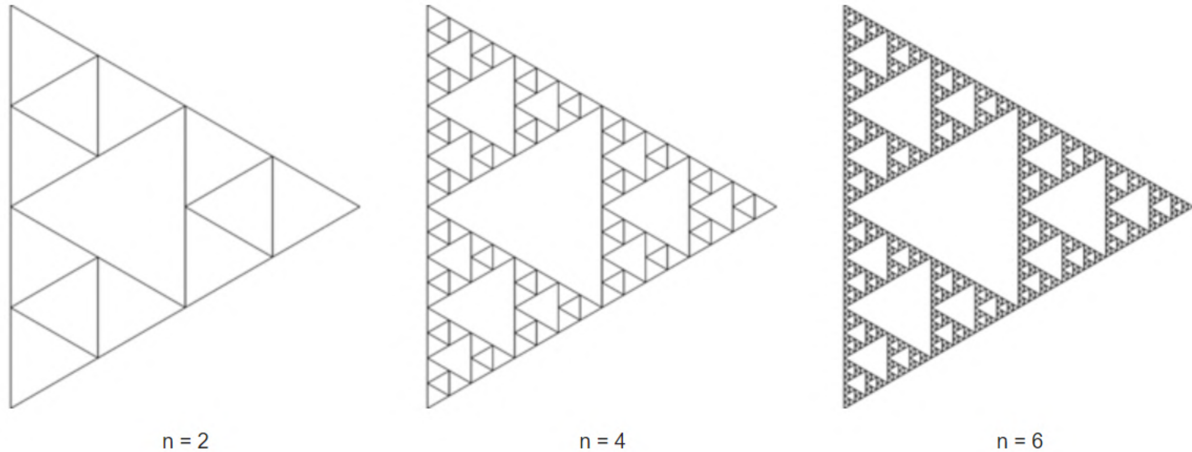


Figure 2.6: Snapshot of 3 turtle graphics states of Sierpinski triangle L-system as defined by equation 2.2; the iteration step is indicated by n; the turtle starts its movement facing vertically from the bottom-left triangle corner; length of triangle sides = $1/n$ [84]

basis for alphabet. These shapes can be one, two, or three dimensional depending on context. Shape grammars can also be composed of mixed dimensionality depending on iteration logic proposed by production rules.

Another key difference between shape grammars and L-systems is the sequential nature of shape grammar iterations. L-systems are iterated with parallel rewrites/mutations of its axiom. This is great for visualizing growth. For instance, L-systems work very well to create a timelapse of a plan life. Also, extended L-systems can be used to model urban sprawl over time. In contrast, shape grammars iterate sequentially. This means that each iteration relies on the resultant state of the previous iteration. This lends itself very well to the creation of buildings and other architecture at various LOD as seen in figure 2.7.

2.2.3 CityEngine

In 2001, Parish and Müller presented a hybrid procedural modeling approach for large urban environments [56]. They point to Wegener's work in suggesting that an urban environment can be split into functional subsystems [82]. Of these subsystems, road networks, land use, and housing are the slowest to change in an urban environment. With this insight, Parish and Müller propose that modeling an urban environment can be distilled down to 3 core tasks: traffic/road network, lot subdivision, and buildings (figure 2.8).

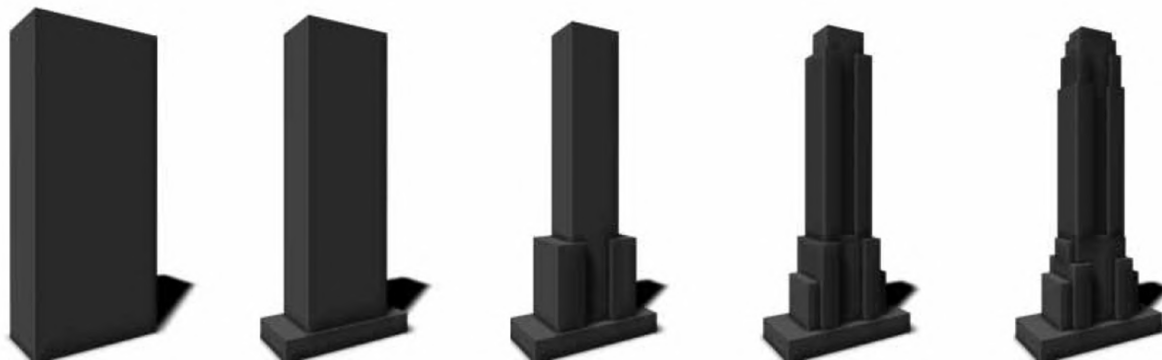


Figure 2.7: Five LOD-level iterations using CGA shape grammar for a skyscraper [56]

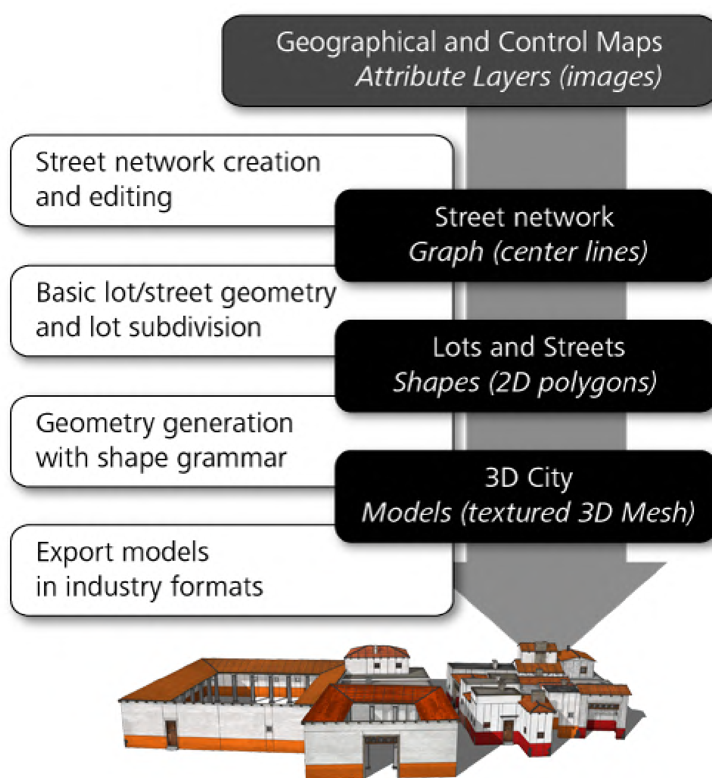


Figure 2.8: CityEngine procedural modeling pipeline [18]

CityEngine is a 3D modeling program built around the modeling paradigm that Parish and Müller proposed. It is primarily used to generate large, urban environments. CityEngine has found a niche within many different industries — urban planning, architecture, blockbuster movies, game development, etc.

The program's workflow contains its own rule scripting language called CGA, which is based on shape grammar. Using CityEngine's default CGA rules (or even custom-scripted ones) in combination with geographical (elevation, land, water, vegetation, etc.) and sociostatistical (street networks, zone maps, population density, etc.) data priors, a user can create semblances of a city environment in a matter of minutes.

Road Networks

To procedurally generate road networks, CityEngine ingests data priors from sources such as OpenStreetMap. This process is detailed in section 4.1. It then uses an extended L-system to generate a road network graph with the likeness of the data presented to it. The benefit of this graph network is that it allows for creation and modification of lots/parcels by leveraging L-system paradigms.

The parallel rewriting nature of standard L-system strings means that parameters embedded in the axiom string are mutated iteratively based on production rules. In a complex L-system, these mutations can easily get out of hand after a few iterations due to an excessive amount of parameters contained within the string.

With embedded parameters, modifications to the L-system become difficult. When new constraints are added, any successive rules need to be rewritten. For a development program such as CityEngine, this can be very compute-intensive and cumbersome. To this end, Parish and Müller propose to migrate parameters from the core L-system string to external functions. Their extended version of L-system strings only contain generic templates, which they call ideal successors. The external functions adhere to global goals and local constraints to operate on these ideal successors [56].

Lot Formations

Closed loops of a street network graph are referred to in CityEngine as blocks or parcels. These blocks need to be subdivided into lot-like shapes for architecture placements. These lot-like shapes are constrained to be convex polygons, primarily rectangles. To achieve this, CityEngine uses a recursive division algorithm that splits along the longest parallel edges.

Buildings

Blocks are first sufficiently subdivided into convex polygon lots. Then buildings and other architecture are procedurally created. To generate architectural models, CityEngine uses CGA shapes [50]. Unlike standard L-systems, CGA shapes rely on a sequential application of rules in order to specify structure. This is similar in scope to Chomsky grammars [9].

The building production process initializes with shape axioms. Instead of strings in L-systems, the set of shapes that production rules operate on is termed as a configuration. During each iteration, a preceding shape is marked as inactive, and a production-based successor is added to the configuration.

Since the production process is sequential, shapes within a configuration have to be prioritized. This is done by assigning rule priorities based on shape LOD. So the generation occurs in such a way that overall LOD geometry can be easily grouped into production iterations (figure 2.7).

Also, since there is an inherent requirement of structural variation in the procedurally modeled buildings, rules have an assigned probability. This ensures a stochastic production process that can be controlled by a seed value.

The generalization of the above concepts are formulated by Müller et al. [50] as such:

$$id : predecessor : cond \rightsquigarrow successor : prob \tag{2.3}$$

In equation 2.3, *id* is a rule identifier. $predecessor \epsilon V$ as portrayed in equation 2.1. *cond* is a logical expression. *successor* takes place of *predecessor* in the shape hierarchy. *prob* represents stochastic likelihood of the rule to execute.

2.3 Real-time Physics-Based Rendering

We need to develop a synthetic dataset generator that strives to achieve perceived realism. Hence, it is important that we understand how light works in the real world and how light is simulated on GPU hardware. Game engines and simulators have to mathematically model the interaction of light with materials in order to render perceptually-realistic scenes onto a viewer's screen. Modern game engines attempt to mimic real-world operational design domain by adhering to the conservation of light energy. In this section, this physically-based lighting approach is explored and certain shortcuts are identified that assist in real-time rendering performance.

2.3.1 Light as an Electromagnetic Wave

In the real world, visual perception is possible due to interaction of light with objects and particles in the environment. In physics and math, visible light is a subset of the electromagnetic spectrum — ranging between 390 nm to 700 nm wavelength (figure 2.9).

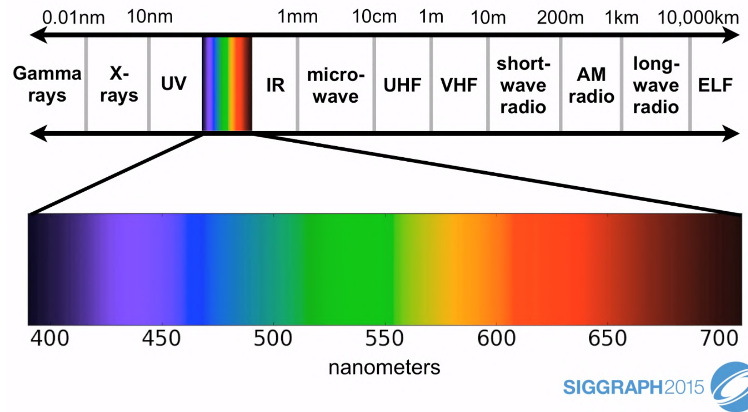


Figure 2.9: Different categories of electromagnetic waves and their respective range of wavelengths [30]

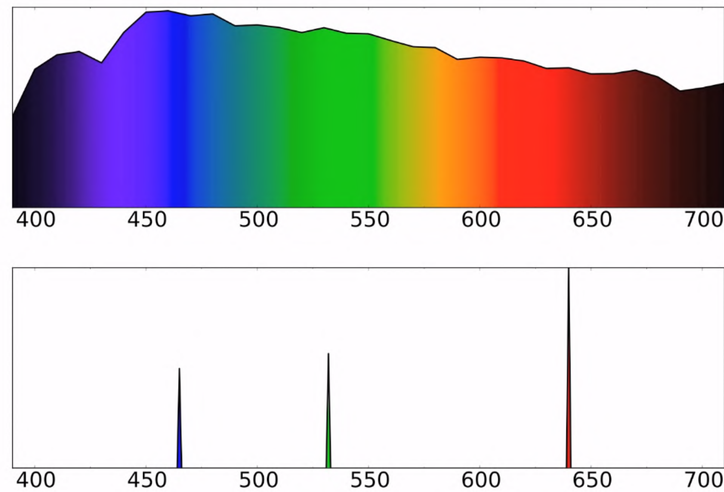


Figure 2.10: Top: SPD of ambient white light on a clear noon day (D_{65}); Bottom: RGB dirac delta SPD that is visibly equivalent to D_{65} to human eyes [30]

Typically, light is a collective energy sum of a distribution of wavelengths along visible band of the electromagnetic spectrum. For instance, figure 2.10 shows the SPD of D_{65} — a standardized approximation of the sky color temperature at noon on a clear day [10].

It should be noted that the perception system of human beings cannot distinguish between light waves representative of top and bottom in figure 2.10 even though the SPD characteristics of these two waves are vastly different. Our vision systems map an infinite-dimensional SPD to a three-dimensional color space. Human beings' subjective perception of visible light is very lossy and not at all representative of the objective information contained in most light waves.

When an electromagnetic wave collides with atoms, it polarizes the atoms. This causes atoms to absorb light energy and eventually release this energy as heat and light. This emitted light then propagates onward to interact with other objects in the environment. This atomic-level interaction between light and objects is what characterizes the objects' visibility properties.

2.3.2 Light as a Ray

As an abstraction to simplify light interaction model for computer graphics, light waves are represented as rays. The atomic properties of objects are abstracted into reflective and refractive properties. In 1986, Immel et al. and James Kajiya [33, 31] introduced the rendering equation 2.4 to model the behavior of light in a simulation environment. This equation is recursive in nature because light is presumed to bounce around an environment infinitely.

$$
\begin{aligned}
L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos\theta_i d\omega_i \\
p \text{ is the position vector of a point in space} \\
\omega \text{ refers to direction vector of a light ray} \\
L_o(p, \omega_o) \text{ is total radiance towards an output direction from a point} \\
L_e(p, \omega_o) \text{ is emitted light energy from the point itself} \\
\Omega \text{ is the unit hemisphere around surface normal} \\
f_r(p, \omega_i, \omega_o) \text{ is the BRDF function} \\
L_i(p, \omega_i) \text{ is radiance of an input direction towards the point} \\
\theta_i \text{ is the angle between surface normal and radiance input vector}
\end{aligned}
\tag{2.4}
$$

In equation 2.4, radiance is a measure of the total observed light energy over an area over a solid angle, given some radiant intensity. The unit of radiance is watt per steradian per square meter.

2.3.3 Conservation of Energy

A core concept of physics-based rendering (PBR) is conservation of energy. Outgoing radiance is never greater than the incoming radiance at a point p, unless the point itself is a light emitter. Modern rendering engines use a specialized version of the full rendering equation, called the reflectance equation (2.5).

$$L_o(p, \omega_o) = \int_{\Omega} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos\theta_i d\omega_i \tag{2.5}$$

In equation 2.5, the BRDF term $f_r(p, \omega_i, \omega_o)$ refers to BRDF. This is a function of the lighting and viewing geometry. It defines a ratio of the amount of radiance that exits an incident point from the incoming radiance at a given incident angle. There are many existing BRDFs that serve different purposes depending on time/realism requirements. For a BRDF to be physically-based, the following properties must be met:

$$\begin{aligned} \text{positivity: } & f(\omega_i, \omega_o) \geq 0 \\ \text{reciprocity: } & f(\omega_i, \omega_o) = f(\omega_o, \omega_i) \\ \text{energy-conserving: } \forall\omega_i, & \int_{\Omega} f(\omega_i, \omega_o) \cos\theta_o d\omega_o \leq 1 \end{aligned} \tag{2.6}$$

An example of a non-PBR BRDF is the Phong BRDF (2.7). This function is very fast to compute, but it is impossible to normalize in an energy-conserving way. Phong BRDF based rendering can have unpredictable behaviour in different lighting configurations. Oftentimes, this means that rendering engineers and artists have to account for these variations with separate texture and material properties for their renderable objects.

$$f_r(\omega_i, \omega_o) = \begin{cases} k_d + k_s \frac{\cos^P(\alpha_r)}{\cos\theta_i} & \theta_i < 90^\circ \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

An example of a PBR-based BRDF is the one used by Unreal Engine 4 — a modified version of the Cook-Torrance BRDF (2.8).

$$
\begin{aligned}
f_r(\omega_i, \omega_o) = \begin{cases} k_d \frac{c}{\pi} + k_s \frac{DFG}{4 \cos \theta_i \theta_o} & \theta_i < 90^\circ \\ 0 & \text{otherwise} \end{cases} \\
c \text{ is the albedo or surface color} \\
D \text{ is a Normal Distribution Function approximating microfacets} \\
F \text{ is Fresnel equation describing reflection ratio at different surface angles} \\
G \text{ is geometry function describing microfacet self-shadowing}
\end{aligned} \tag{2.8}
$$

2.3.4 Light Interaction with Surfaces

Reflection is defined as the portion of a light ray that bounces off a material surface. This is characterized by an equality between the angle of incidence of the light ray, θ_i, and the corresponding angle of reflection, θ_r. The color properties of the incident light ray is retained.

Surfaces that have irregularities with magnitudes smaller than the wavelength of light can be considered to be perfectly smooth for ray approximations. Such perfectly smooth surfaces would reflect light rays as seen in figure 2.11.

However, most material surfaces in the real world have microgeometry — irregularities with magnitudes greater than the wavelength of light. This causes an apparent roughness of the surfaces (figure 2.12). In a macroscopic sense, this can be modeled as light reflecting stochastically in a cone shape (figure 2.13). In the BRDF function, this stochastic cone of reflection is described by the specular term, k_s.

Refraction is the ability of light rays to penetrate beyond the surface of objects. Repeated collisions with objects' internal molecules scatter the light ray into smaller rays, while absorbing rays of specific wavelengths. Eventually, some of the non-absorbed, scattered light rays exit the material surface again irrespective to the initial angle and location of light ray incidence. This effect is called sub-surface scattering (SSS). Apparent in most non-metallic material, this is what characterizes the color properties of such objects. In the BRDF function, stochasticity of the phenomena caused by refraction is captured by the diffuse term, k_d.

2.3.5 Shortcuts for Real-time Performance

With today's technology, it is impossible to recursively compute the reflectance equation for every visible point from every screen-space pixel in real-time. This process is called

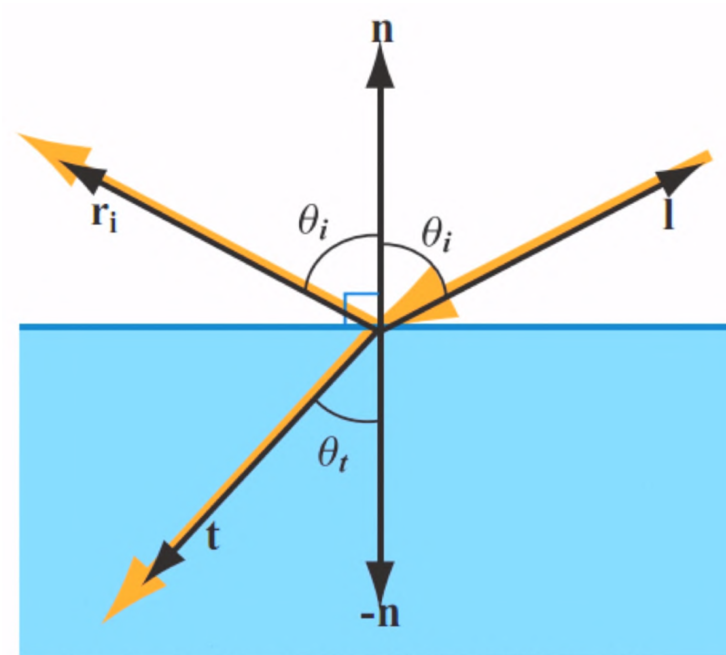


Figure 2.11: Ideal reflection/refraction diagram of a light ray [30]

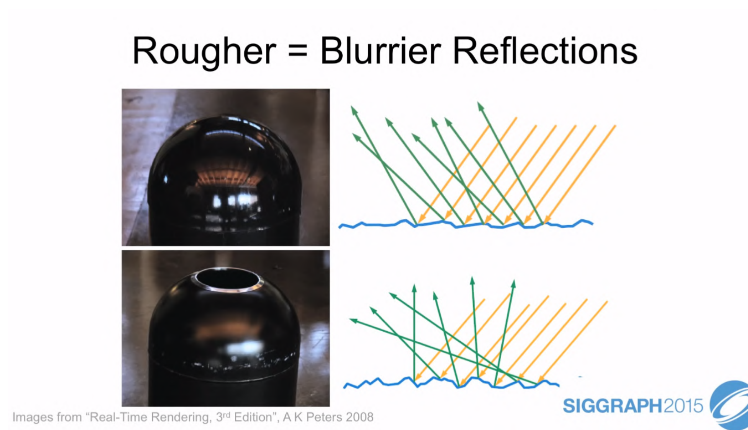


Figure 2.12: Visible surface roughness caused by microgeometry [30]

ray-tracing. Using Monte Carlo importance sampling, offline ray-tracing methods are practical in the film industry where realism is far more important than real-time performance. However, for a simulator or video game engine, ray-tracing is not feasible.

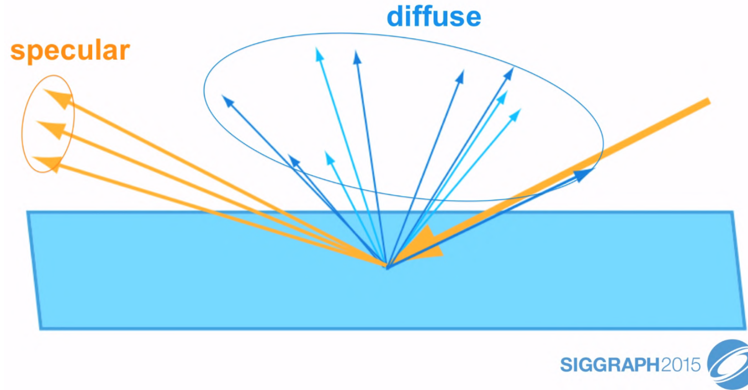


Figure 2.13: Stochastic specular and diffuse distributions in macrogeometry can emulate effects of surface microgeometry [30]

Epic Games introduced some approximation techniques in their rendering pipeline that make it possible to achieve real-time rendering performance with PBR. These approximation techniques revolve around image-based lighting (IBL).

IBL is a shortcut used by real-time rendering engines to approximate the effects of non-primary light bounces around the environment. The approximation is achieved by breaking down the reflectance integral equation into diffuse irradiance (equation 2.9) and specular IBL (equation 2.10).

$$L_o(p, \omega_o) = k_d \frac{c}{\pi} \int_\Omega L_i(p, \omega_i) \cos\theta_i d\omega_i \tag{2.9}$$

$$L_o(p, \omega_o) = \int_\Omega k_s \frac{DFG}{4\cos\theta_i\theta_o} L_i(p, \omega_i) \cos\theta_i d\omega_i = \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos\theta_i d\omega_i \tag{2.10}$$

The diffuse irradiance equation is approximated by rendering engines using pre-computed irradiance maps. The equation is discretized as a Riemann sum in polar coordinates (equation 2.11). An environment cubemap of a region-of-interest is captured. Then the irradiance map is convoluted by discretely sampling the environment cubemap using the Riemann sum-based equation 2.11. This irradiance map is then saved into a texture. During render time, diffuse irradiance becomes as efficient as a texture lookup (figure 2.14).

$$L_o(p, \phi_o, \theta_o) = k_d \frac{c}{\pi} \frac{1}{n1n2} \sum_{\phi=0}^{n1} \sum_{\theta=0}^{n2} L_i(p, \phi_i, \theta_i) \cos(\theta) \sin(\theta) d\phi d\theta \tag{2.11}$$

Figure 2.14: On the left is an environment cubemap [80]; on the right is the corresponding irradiance map generated using discretized irradiance Riemann sum equation 2.11

The specular IBL equation can be computationally simplified using split-sum approximation (equation 2.12), as proposed by Epic Games [34]. The left integral of the split-sum can be treated similar to diffuse irradiance. In this case, the generated texture is called a pre-filtered environment map. This consists of environment map at various mipmap levels. These mipmaps are interpolated to accommodate different roughness values (figure 2.15).

$$L_o(p, \omega_o) = \int_{\Omega} L_i(p, \omega_i) d\omega_i \int_{\Omega} f_r(p, \omega_i, \omega_o) \cos\theta_i d\omega_i \tag{2.12}$$

The right integral of the split-sum can be reduced to a function of $cos\theta_i$ and roughness. Using a quasi-Monte Carlo importance sampling technique, this function can be pre-computed into a 2D lookup texture (LUT) as in figure 2.16.

Figure 2.15: Example pre-filter environment map with multiple mipmap levels (courtesy of [80]); note that the information gradually gets blurrier at smaller mipmap scales

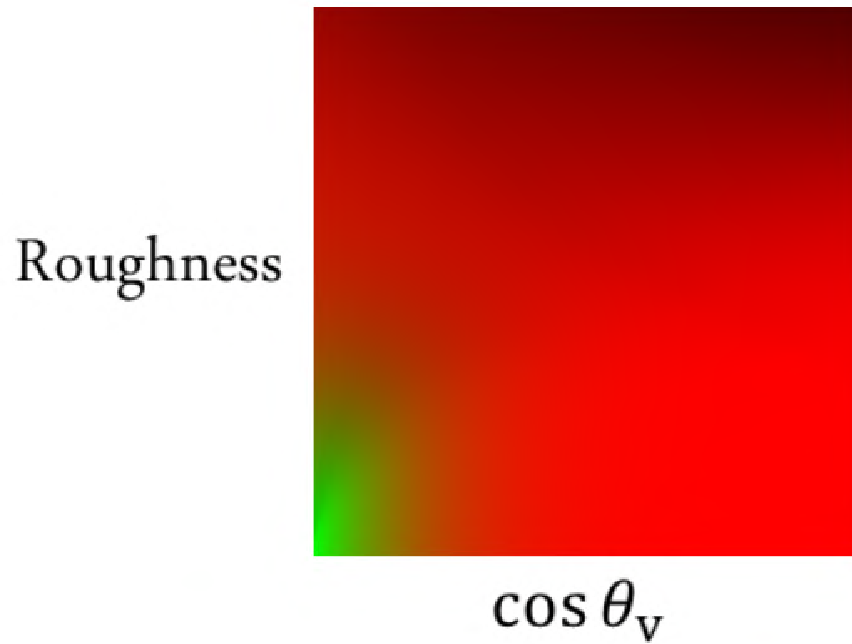


Figure 2.16: Pre-computed BRDF integration map; saved into a two-dimensional lookup table as a function of $cos\theta$ and roughness factor [34]

Chapter 3

URSA Dataset

One way to generate large-scale synthetic datasets is to leverage existing commercial video game products. GTA5 has proved to be very useful in this regard. This chapter identifies GTA5-based dataset collection approaches. We explore the feasibility in using the GTA5 game world and engine towards capturing and quantifying variation in weather and lighting effects.

My primary contribution towards this project was in developing a labeling framework for human labelers. I also designed the data generation pipeline for URSA (figure 3.4).

3.1 Introduction to GTA5

GTA5 takes place in the fictional city of Los Santos. This fictional city is a satire of real-world Los Angeles, California. Famous locations are easily recognizable. For instance, the famous Hollywood sign is instead Vinewood in the game world, and the in-game replica of the real-world Griffith Observatory is called Galileo Observatory (figure 3.1). Other environmental factors of the game world also reflect Southern California vibes, such as flora and fauna of the area. For all intents and purposes, the game world faithfully resembles the ODD of the city of LA.

The GTA franchise has a rich history in vehicle-based game missions and exploration. GTA5 follows suit in this regard with a very robust traffic system and upwards of 300 unique transportation modalities (bicycles, cars, buses, trains, planes, ships, etc.). The outdoor portion of the game world lends itself very well to data collection of road scenes.

Figure 3.1: Galileo Observatory is GTA5's rendition of real-world Griffith Observatory [19]

For this reason, the research community has shown a lot of interest in modifying GTA5 game engine to generate various datasets from object detection to semantic segmentation [59, 60].

3.1.1 Deferred Shading

GTA5 rendering engine uses a deferred shading pipeline [13]. This is a technique used by real-time graphics engines to reduce the number of fragment shader lighting operations that need to be handled per frame.

Let's first look at the case of traditional rendering pipeline, otherwise known as forward shading. In this case, all geometry meshes from the scene is handed to the GPU. These mesh vertices are passed through vertex shaders. Vertex shaders modify the location where the mesh vertices are visible in the scene. Geometry shaders are then optionally used to further modify mesh complexity (tessellation, shadow volume extrusions, etc.). Finally, a fragment shader rasterizes the mesh and applies shading to it based on all light sources in

the scene. This process then repeats for all other meshes that are in GPU queue for the frame. The time complexity of the final fragment shader can be represented as

$$O(num_meshes * num_lights) = O(n * n) \tag{3.1}$$

Deferred shading can provide a significant speed-up in render time by deferring the shading step done by final fragment shader. Prior to this, all meshes in GPU queue are passed through the 3D shaders and rasterized to screen space. Then the fragment shader applies lighting to only the visible pixels in the scene. Hence, time complexity of deferred shading is

$$O(screen_resolution * num_lights) = O(n) \tag{3.2}$$

For a video game such as GTA5, an order of magnitude improvement in rendering time is crucial when it has to deal with millions of polygons and numerous light sources being thrown on to the screen, all in real-time with virtually no loading screen.

While performance benefits are great, the main benefit for semantic segmentation research is found in the state of the graphics buffers during deferred rendering. More specifically, there exists a rasterized layer in the rendering pipeline, which contains mesh, texture, shader (MTS) values for game objects that are being thrown on to the screen.

3.2 Ground Truth Annotation via Detouring

Richter et al. take advantage of deferred rendering to produce their PFB dataset [60]. They tap into the game resources during render time by using detouring. Detouring is the act of injecting custom code that intercepts calls to Direct3D API. Using this code, they are able to sniff out and hash the MTS patches that the game is rendering.

MTS patches can be thought of as super-pixels. A few MTS patches together account for a scene object in the game. An MTS patch does not contain content from multiple objects. For instance, a car can be made of a few different MTS patches, but an MTS patch cannot contain a car and a sidewalk object.

By semantically labeling and propagating MTS patches, data collection from the game world is achieved for the purposes of semantic segmentation. When a rendered frame is composed of pre-labeled MTS patches, this annotation process can take around 30 seconds. Otherwise, labeling a frame with new MTS patches takes 7 seconds on average. According to Richter et al., this labeling time is at least 500 times faster than human labeling done for real-world datasets such as Cityscapes.

3.3 Ground Truth Annotation via URSA Method

30 seconds of annotation time is too long for something that is automated. We can do better. With this thought in mind, we tackled the challenge of GTA5 hacking with our own dataset. GTA5 (as well as previous games in the series) has a legion of fans who make up one of the most creative modding communities in the video game world. We decided to leverage the resources available for modding the single-player version of the game.

3.3.1 Method Overview

The core idea behind our URSA dataset is preemptively labeling all relevant game resources before beginning the data collection process. This way, we are able to generate, in real-time, a virtually infinite amount of data points from the game world for semantic segmentation research.

To achieve this, we created a modded copy of all road-scene game resources. Using Rockstar Editor [23], we first record and render over-the-hood footage of vehicles driving around in the stock game environment. We then load our modded game resources and re-render the recording, producing the corresponding ground truth annotations.

In our process, instead of aforementioned MTS values, we identify frame super-pixels with FMSS values. Having access to in-game assets grants us access to the persistent storage locations of textures, which is not possible with detouring method. The FMSS correlation results in 1,178,355 unique game resources. Of these, many are used for cutscenes, in-door gameplay, and otherwise scenes that are non-relevant to road-going events.

We identified that a total of 56,540 FMSS values are relevant for road-scene semantic segmentation, less than 5% of all FMSS assets. Even still, this is a significant amount of resources that are required to be hand-labeled. This is not time-feasible for a small group of graduate students.

3.3.2 Amazon Mechanical Turk

We employed workers from AMT. For this, we created a web-based labeling framework using JavaScript. The user interface (UI) is shown in figure 3.2.

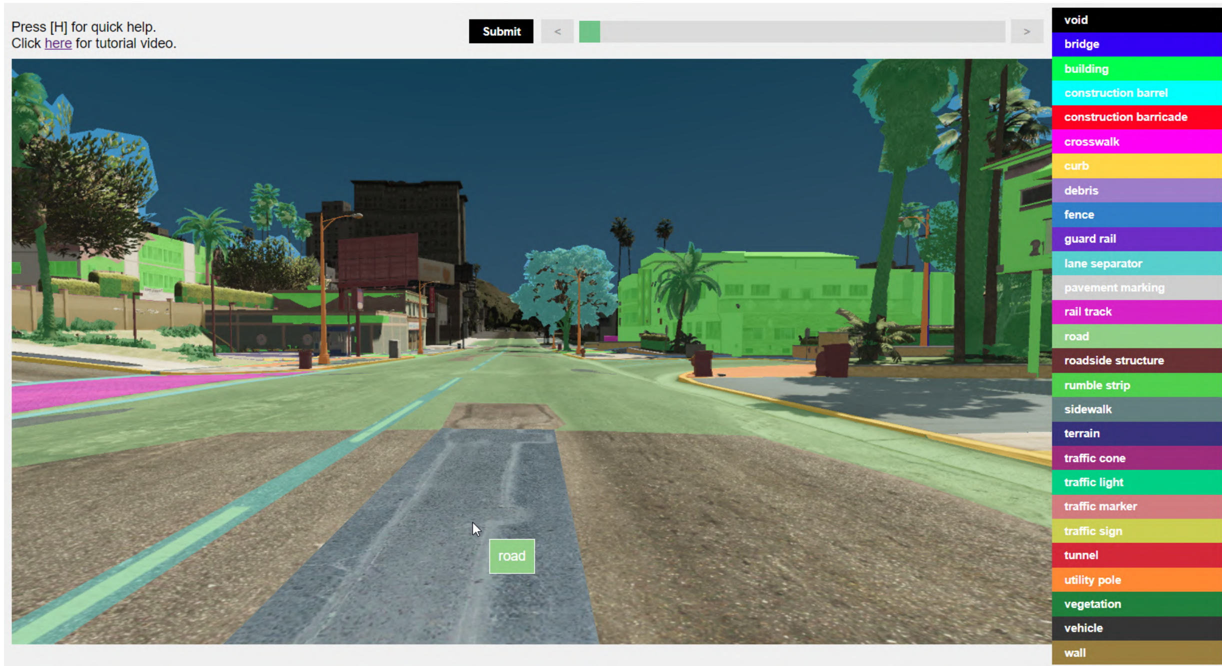


Figure 3.2: Web-based interface exposed to Amazon Mechanical Turk workers to efficiently label road-going FMSS values that were captured from the game world [1]

Task Setup

From the game world, we established that there exists a set of 3,388 scenes that has full coverage of the 56,540 road-scene FMSS values [1]. We had to be careful about the design of our tasks to find the right balance between worker satisfaction and label reliability. This task is inherently a task of subjective perception.

AMT tasks are time-sensitive with variable costs depending on task requesters. Workers are paid based on requesters' task acceptance criteria. An unsuccessfully submitted task by a worker can be rejected by the requester, hence penalizing the worker's rating.

Manually invigilating 3,388 unique tasks to verify annotations is almost as complex as the initial task. Instead we automate most of the invigilation by relying on a majority voting scheme. This has a two-fold benefit. The first is that workers reliability score is determined organically by the labels provided by other workers. The second is that the more votes accumulated towards the labels of FMSS values, the less effect subjective perception has in biasing our data.

As oracles, we hand-labeled a random sample of 200 FMSS values. We then designed a preliminary experiment with scenes containing these 200 samples. Through performance analysis of the labeling UI and AMT, we empirically determined that an average of 6-7 votes were needed per FMSS to achieve 75% label accuracy against the oracle labels. This is demonstrated in figure 3.3.

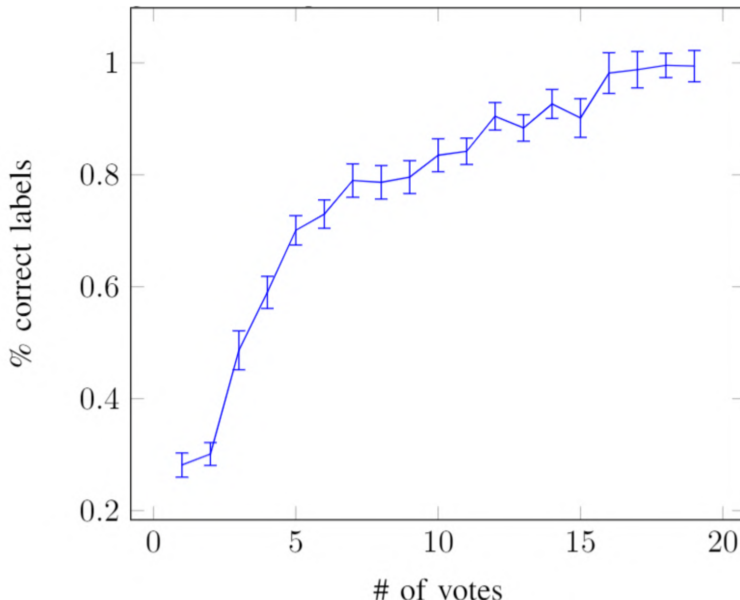


Figure 3.3: Cost-benefit visualization of Amazon Mechanical Turk worker votes

Data Labeling Analytics

With these priors about our labeling UI and AMT dynamics, we designed 2,312 tasks by various groupings of the 3,388 scenes. This means that, on average, each scene was repeated 1.465 times. Also, we identified that each FMSS value in question repeats about 4.21 times on average (with a median of 3) in the set of 3,388 scenes. Hence, with 2,312 tasks we average about 6.97 votes per FMSS value. This provided us with near 75% confidence with the AMT workers using our labeling tool.

Another noteworthy detail about our AMT tasks setup is that there were an average of 3.34 scenes with 30.46 FMSS values to be labeled per task (median of 3 scenes with 30 FMSS values). $0.97 was paid out on average. We experimented with paying $0.77 with 24

minute time limit, $0.96 with 26 minute time limit, and $1.20 with 38 minute time limit. Of these amounts, 20% of the fees go to AMT service and the rest is given to the worker.

Worker productivity was best with the $1.20 setup, as it gave the workers a stress-free environment. Average time-to-completion was 12.19 minutes with a median time-to-completion of 10.65 minutes. Prior research on AMT indicates that median wage of AMT workers is about $2/hour [27]. Our budget allocated towards the AMT labeling task aligns with this prior research.

3.3.3 Data Generation

Figure 3.4 shows a high-level overview of data generation using our URSA method. The labeling process, done by oracles and Amazon Mechanical Turk workers, is covered in the Labeler System of the URSA pipeline.

Scene Recorders simulate dashboard cameras placed on road-going, in-game vehicles. The AI is set to drive these vehicles along main roadways throughout the game world. This is achieved via a popular community mod called DeepGTAV [63].

As mentioned in section 3.3.1, gameplay clips are recorded with Rockstar Editor's in-game recording feature [22]. Rockstar Editor allows for offline, high quality rendering of pre-recorded scenes from the game world. Essentially, this removes the real-time rendering constraint on the game engine. As a result, scenes can be rendered with more realistic shader and lighting details. Characteristic nuances such as chromatic aberration and lens distortion can be removed in this offline rendering mode. This rendering task is delegated to the Realistic Renderers in figure 3.4.

Realistic shaders and lighting models cannot be used to render ground truth data. In fact, the ground truth rendering has to be stripped down to the barebones of the graphics engine. Texture blending shaders, self-shadowing effects, lens flares, etc. need to be removed to let ground truth annotations render without any artifacts. The modded texture pack generated via the Labeler System also has to be swapped in for ground truth rendering. This rendering task is delegated to Ground Truth Renderers.

Finally, the accumulation of rendered output data is stored on a server computer with a RAID 5 storage setup. A daemon script on this server prunes and sanity checks the stored dataset. Using this parallel data collection and rendering process, the URSA dataset is generated within 63 hours. This consists of 1,355,568 RGB images and their corresponding ground truth annotations.

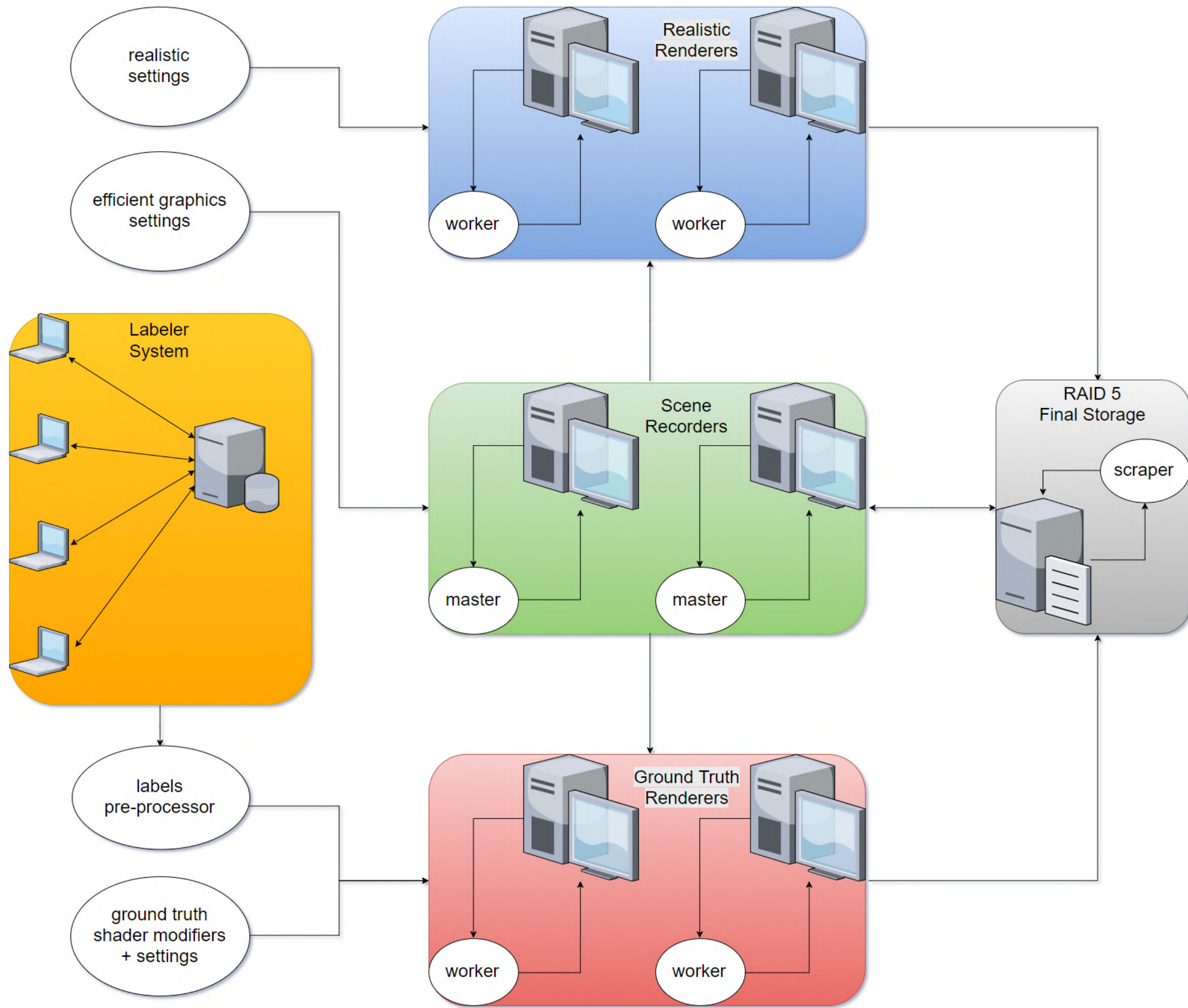
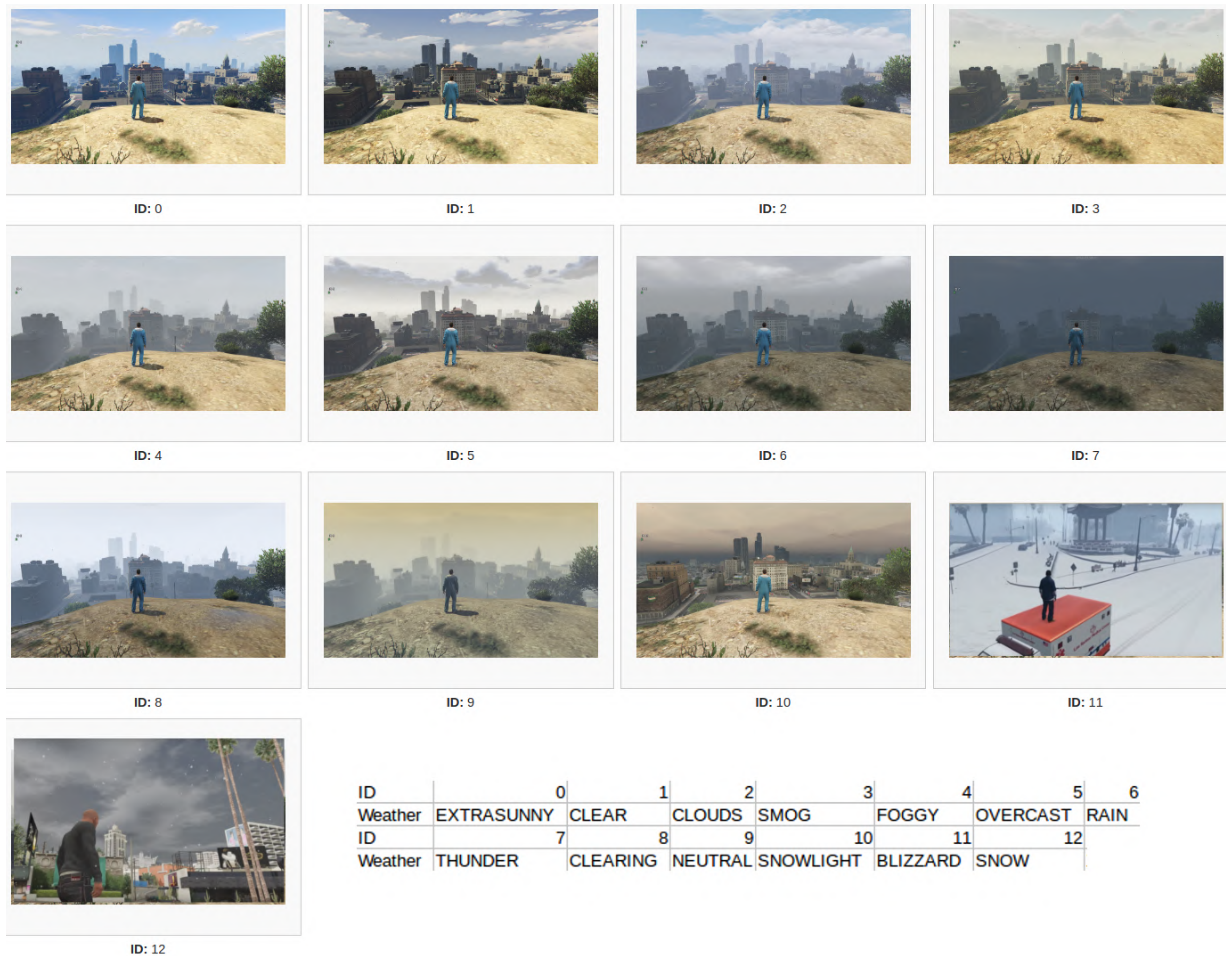


Figure 3.4: URSA dataset generation pipeline

3.3.4 Influence Factor Variations

GTA5 has scripted weather variations. There are 13 different usable preset weather configurations in the game world. Namely, these are "CLEAR", "EXTRASUNNY", "CLOUDS", "OVERCAST", "RAIN", "CLEARING", "THUNDER", "SMOG", "FOGGY", "SNOW-LIGHT", "BLIZZARD", "NEUTRAL", and "SNOW". Figure 3.5 shows previews of these. The game engine works such that shaders, texture lookups, etc. all adhere to the state of

the game weather. For instance, the textures in figure 3.6 apply to the same material but during different weather conditions.



ID	0	1	2	3	4	5	6
Weather	EXTRASUNNY	CLEAR	CLOUDS	SMOG	FOGGY	OVERCAST	RAIN
ID	7	8	9	10	11	12	
Weather	THUNDER	CLEARING	NEUTRAL	SNOWLIGHT	BLIZZARD	SNOW	

Figure 3.5: GTA5 weather configuration previews [83]

The game engine circulates through these weather configurations based on its time logic and scripted events. GTA5 game engine understands the transitions between certain weather configurations and has scripted behaviour as such. Some transitions are impossible. For example, the game world does not naturally switch from “BLIZZARD” to “EXTRASUNNY”. When this is forced using a mod, there is a visible flash on-screen akin to being overexposed to light when leaving a tunnel. This creates difficulties in shifting between different weather configurations during data collection.

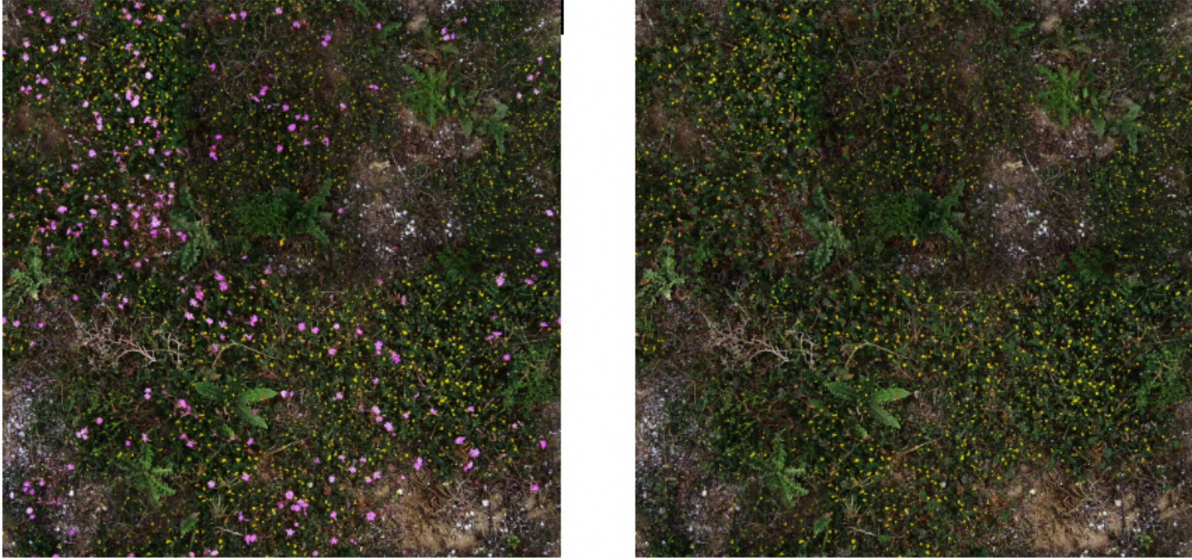


Figure 3.6: Example texture variations used in GTA5; left is used in summer weather configurations, right is used during winter

Worth noting is the fact that the game engine does not easily allow for modulating a specific weather parameter. For example, there is no simple way to quantifiably and uniquely (without affecting other influence factors) vary the amount of cloud coverage. The position of the Sun in the game world is also locked along its orbital axis. The only variable that directly influences Sun position is the time of day. This makes it problematic to capture the Sun's angle and shadows with respect to the view camera for data collection.

The biggest drawback towards capturing influence factor effects using the URSA method is in using Rockstar Editor. Once a scene is recorded from the game world, the weather parameters and Sun positions are baked into the recording. Our understanding is that this is done to reduce file sizes of the recordings. In essence, to collect data with different influence factors, new scenes have to be recorded from the game world with the specified weather and time-of-day variations. This means that scenes are not easily reproducible due to the pseudo-random events in the game world.

Chapter 4

ProcSy Dataset

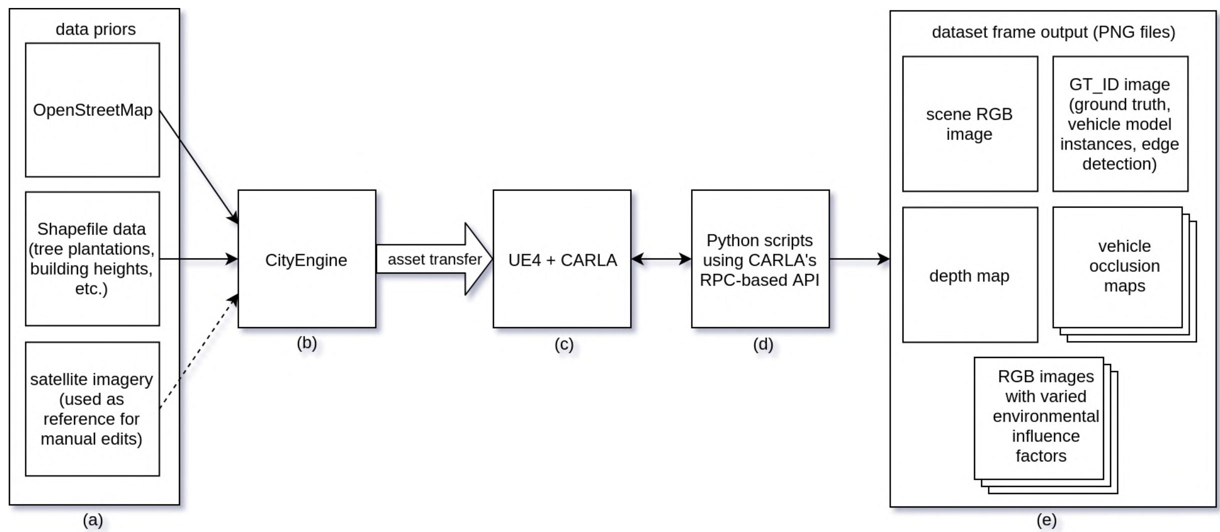


Figure 4.1: Generation pipeline for ProcSy dataset: (a) data priors are used for procedural modeling; (b) CityEngine is used to generate three-dimensional, procedural world map; (c) UE4 and CARLA are used for realistic lighting and weather effects; (d) dataset generation through CARLA is controlled via Python-based scripting; (e) each frame of our dataset have the outlined images rendered [35]

In this chapter, we expand on the methods and experiments discussed in our ProcSy dataset paper [35].

The name "ProcSy" (pronounced "proxy") is an amalgamation of two words — "procedural" and "synthetic". "Procedural" is an homage to the method of 3D modeling that is used to generate the environment. "Synthetic" is in reference to the dataset being generated from a virtually created world as opposed to being collected from the real world.

The goal behind ProcSy was to generate a dataset with quantifiable scene variations. Instead of temporal frames, where objects are in motion such as in our URSA dataset, we collect data by teleporting the camera around the 3D world. This allows us to collect data without the need of creating AI and animations for the virtual world objects to move around. This reduced the challenge at hand of making a virtual world from scratch for the purposes of semantic segmentation research towards influence factor variations.

My primary contribution was the entire data generation pipeline. I identified procedural modeling as a viable option for dataset generation. I generated the blueprint map for our experimentation and integrated the generated map assets into our UE4-based rendering ecosystem.

4.1 CityEngine Workflow

To generate the virtual environment, we make extensive use of CityEngine. As discussed in section 2.2, CityEngine employs procedural modeling methodologies to enable a small group of users to quickly generate reproducible, massive, city-like environments. We first need to feed CityEngine with prior data for this model generation.

4.1.1 Data Priors

We pick a 3km^2 region of Waterloo, Ontario (figure 4.2). This area was picked because of its relevance to our overarching Autonomoose research project [74]. Bathurst Drive and Colby Drive loops are heavily used to test various aspects of the Autonomoose platform. So it makes sense to use this hotspot as the initial blueprint for a synthetic semantic segmentation dataset. Data priors for CityEngine are accumulated from various sources.

OpenStreetMap

The first prior to be used is OpenStreetMap [26]. Primary road networks and building footprints data are gathered from this crowd-sourced mapping platform (figure 4.2). Ideally,

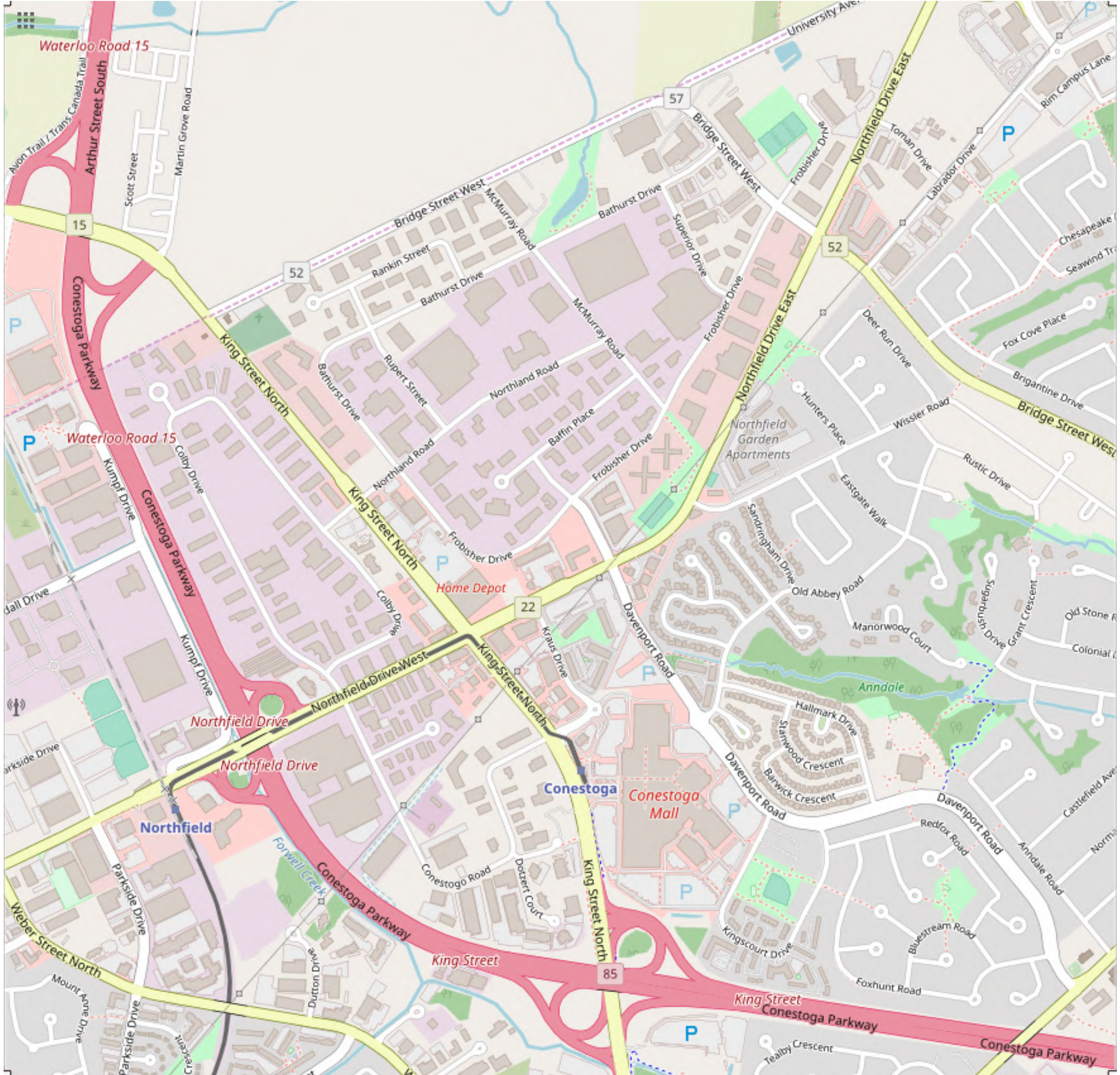


Figure 4.2: View of the 3km^2 region-of-interest of Waterloo, Ontario for our ProcSy dataset as seen on openstreetmap.org online map [26]; Bathurst Drive and Colby Drive loops are captured in this region; the region also features a highway overpass (seen in red)

we require High Definition map data to closely represent our AI. Such lane-level details as lane width and banking angles are not expressed in OpenStreetMap data. However, there are currently no standardized and open-source high definition maps available for the scope of a graduate student research project.

City of Waterloo Open Data

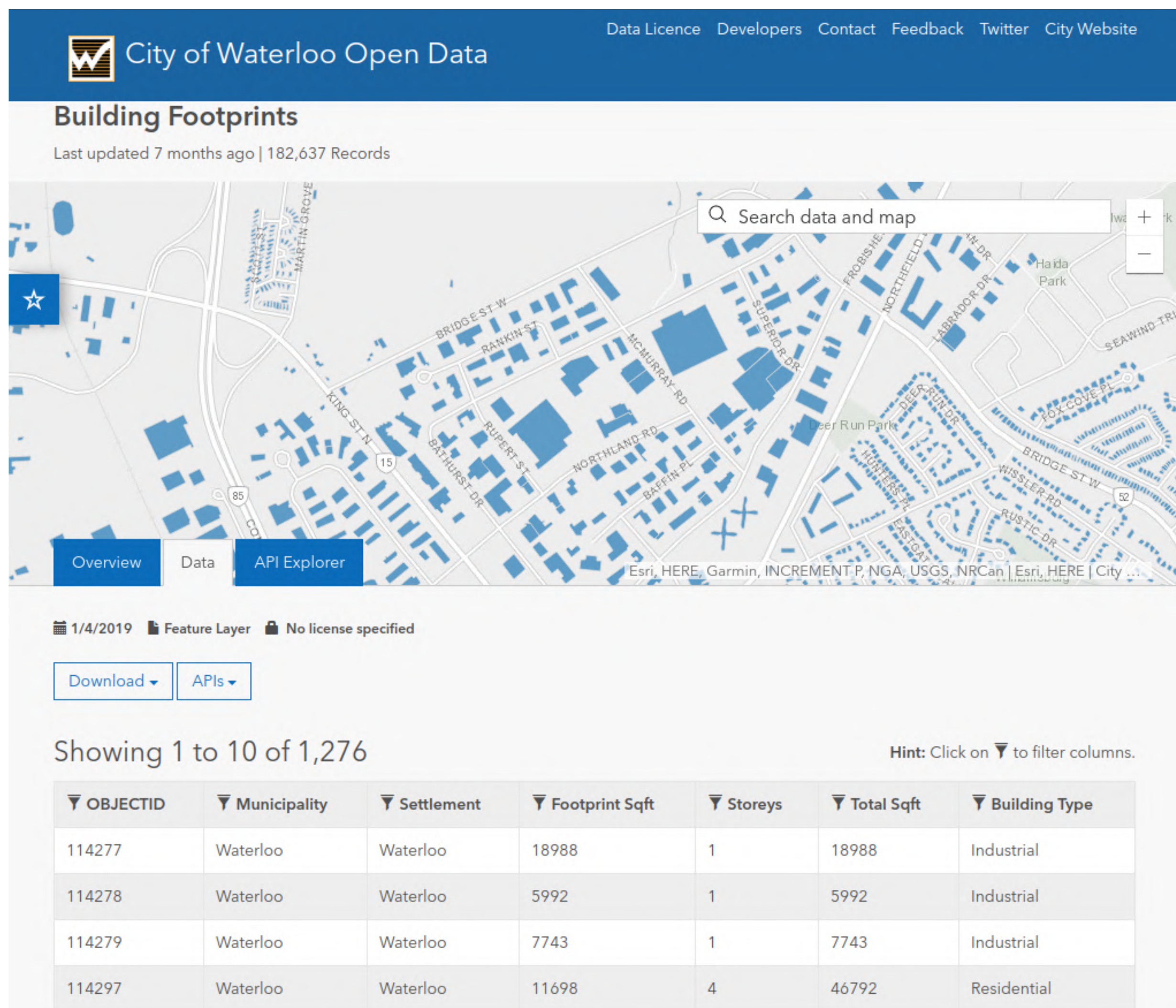


Figure 4.3: Building height data provided by City Of Waterloo [81]

City of Waterloo Open Data provides more specific details about our map region-of-interest [81]. OpenStreetMap does not provide accurate representation of building heights around Waterloo, Ontario. This is collected from the Building Footprints dataset (figure 4.3). This dataset provides consistent height representations in units of number of storeys.

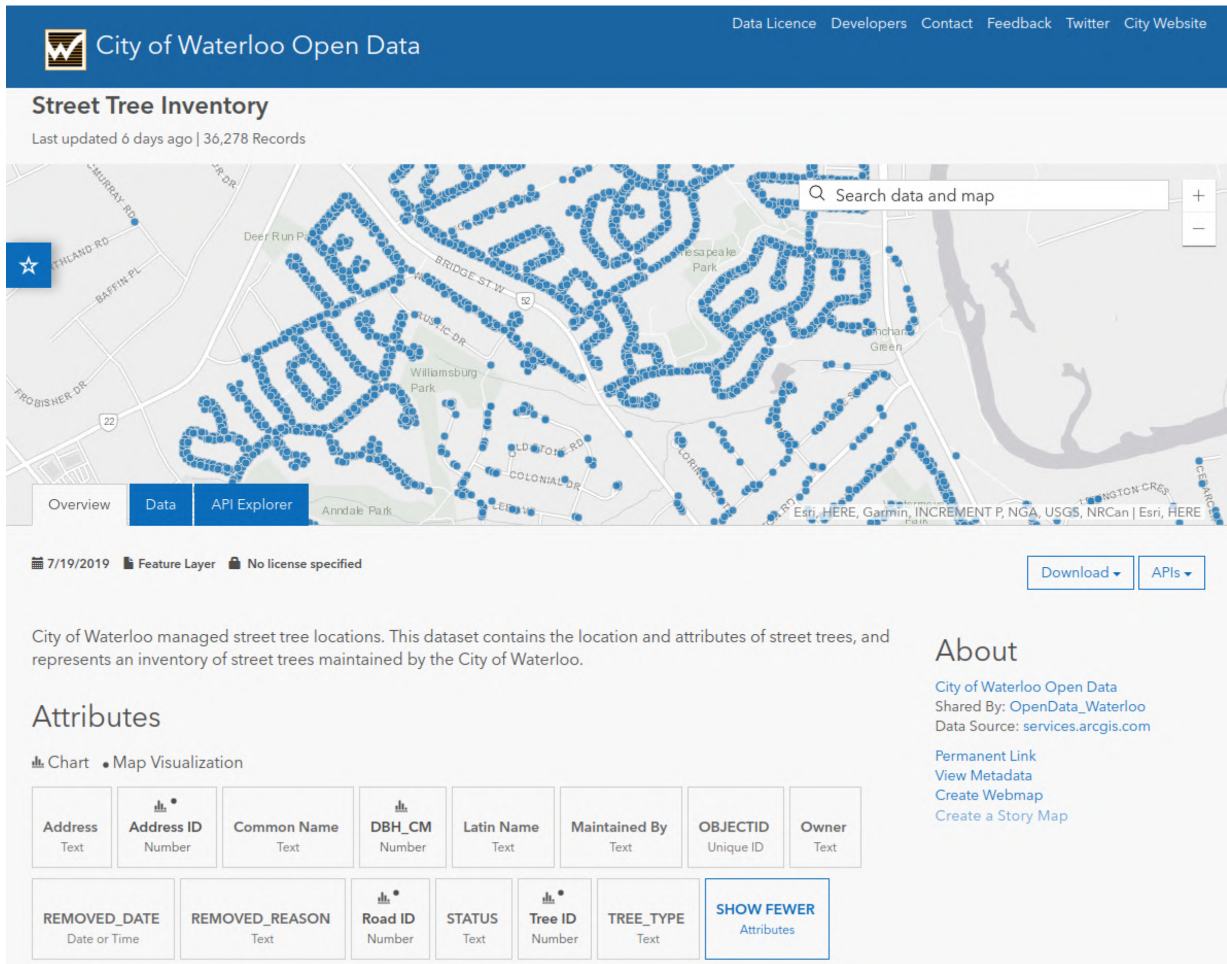


Figure 4.4: Tree plantation data provided by City Of Waterloo [81]

Another Waterloo dataset that we use is Street Tree Inventory (figure 4.4). This dataset provides street planting locations and types of trees that are managed by the City of Waterloo. CityEngine has 3D representations of around 80 different types of trees in its renderable assets library (figure 4.5). We gather the Waterloo data, and using a Python script, we map tree types to those represented by CityEngine. This allows for quickly

Figure 4.5: Sample 3D tree and plant models that are used by CityEngine for populating environment with vegetation [17]

populating the virtual environment with references to existing real-world plantations. We further populate the virtual environment with greenery in city parcels where there exist fertile terrain.

Digital Elevation Model

A third data point that can be used for environment generation is height map data. Digital Elevation Model (DEM) can be acquired from various sources. Scholars GeoPortal is one such source [29]. Organized by the Ontario Council of University Libraries (OCUL), Scholars GeoPortal allows academic researchers and students to query for geospatial data such as DEMs. The relative elevation differences within our region-of-interest are negligible. As a complexity minimization effort, we opt to consider the base terrain of our virtual world to be flat.

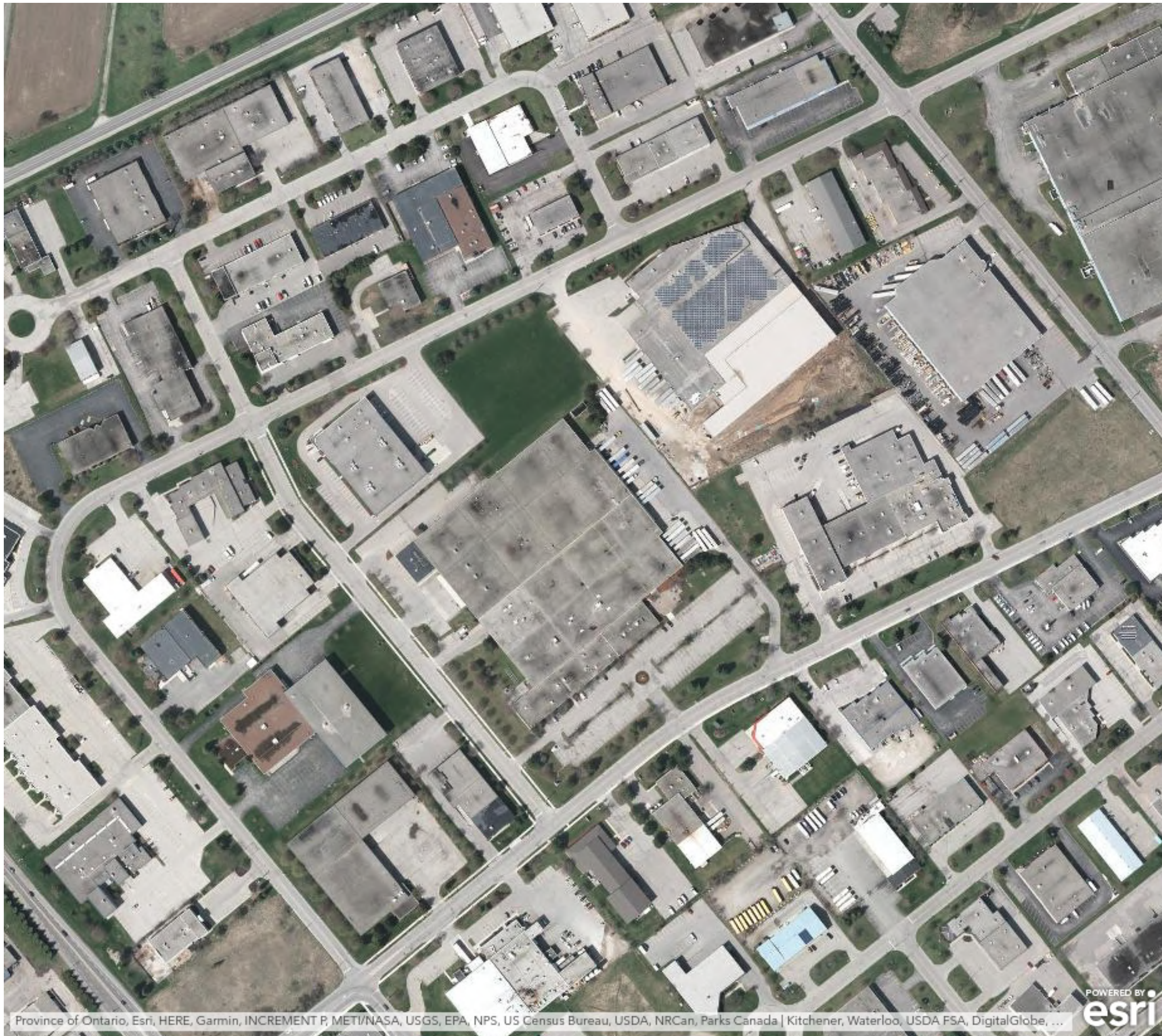


Figure 4.6: Mid-level zoom of Bathurst Drive as provided by Esri satellite imagery [16]; higher magnification allows for distinguishing lane-level features such number of lanes and width of lanes

Reference Satellite Imagery

CityEngine stitches data priors together in order to create a three-dimensional representation of the real-world counterpart. For ProcSy, it took about one-person hour to identify and collect the relevant priors that have been outlined.

Due to data unavailability, CityEngine cannot automatically create exact road network layouts. As mentioned previously, a potential solution may be to use high-definition (HD) maps with lane-level details. However, HD map data in an open-source format is still not readily available in the public domain. Currently, the most cost-effective approach to this problem is using openly available satellite imagery of road networks. Thus, we used satellite imagery provided by ESRI as reference to manually adjust road graphs in CityEngine (figure 4.6).

The time-consuming aspect of procedural modeling is to restructure road networks by accounting for lane-level details. This step took one person approximately 40 hours for the 3 km^2 geographical area. Details such as number of lanes, street/lane-width, and heights of overpasses had to be reconstructed or featured in manually.

4.1.2 Asset Generation

After the base layers are generated using data priors and satellite reference imagery, we create new layers in the CityEngine workflow to populate the environment with statically placed pedestrian and vehicle assets. Out of the box, CityEngine provides low polygon pedestrian and vehicle mesh models that can be freely used.

Pedestrians

There are 31 different pedestrian mesh models used in our dataset (22 adult males and 9 adult females). As can be seen in figure 4.7, the meshes represent a wide variety of standing/moving poses. These mesh models are randomized by CityEngine and populated along sidewalks and lots. The distribution of pedestrians is controlled by weights assigned to road network edges. For instance, a bigger road is likely to be populated with more pedestrians along its sidewalks than a small back-alley path.

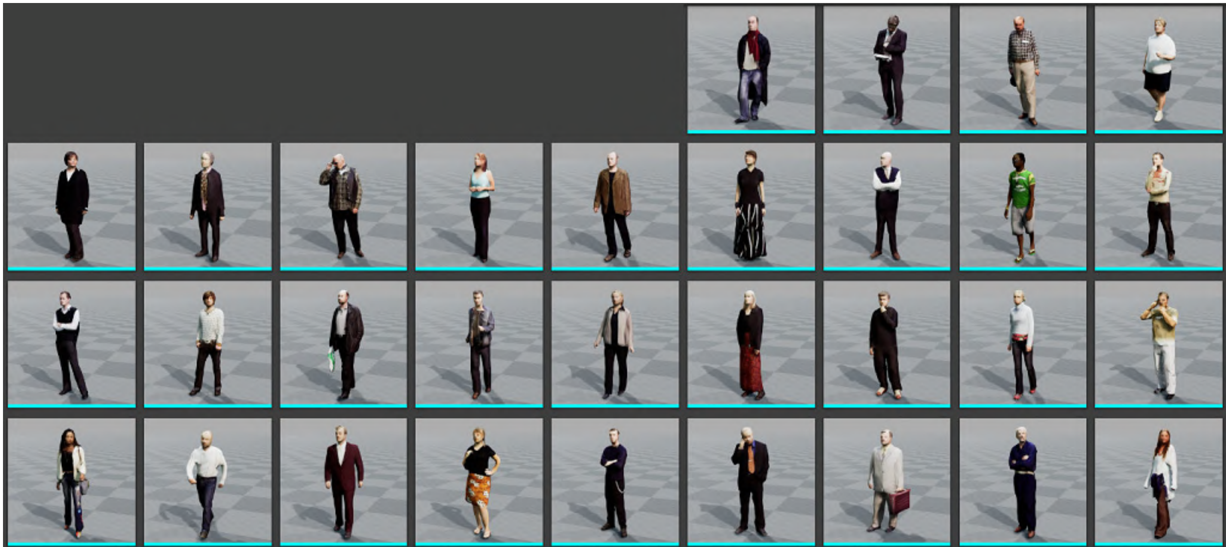


Figure 4.7: Low-poly out-of-the-box pedestrian models provided by CityEngine

Low-poly Vehicles

CityEngine provides 43 different vehicle models (40 small vehicles and 3 large vehicles) of low polygon count (figure 4.8). Using a similar procedure as that for pedestrians, these vehicles are populated on the roads and parking lots.

CityEngine-provided mesh models have a polygon count that is an order of magnitude less than that of commercial video games such as GTA5. The small vehicles have an average of 1,880 polygons (median of 1,729 polygons). Large vehicles have an average of 6,243 polygons (median of 7,180 polygons).

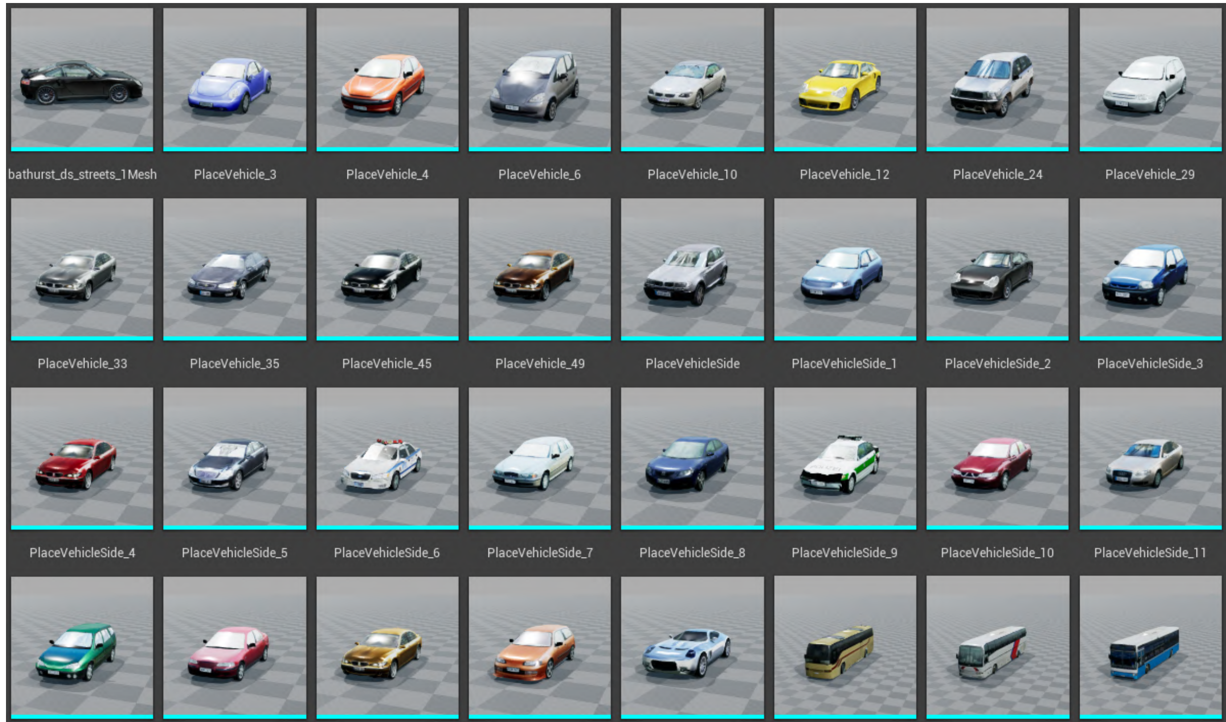


Figure 4.8: A sample of low-poly out-of-the-box vehicle models provided by CityEngine

GTA5 High Fidelity Vehicles

Using tools from GTA5's modding community, we extract mesh models from the single-player GTA5 game. A total of 202 relevant vehicle models are extracted — 155 small vehicles and 47 large vehicles (figure 4.9).

GTA5 has various LOD models of these vehicles that are strategically used by the game engine for run-time optimization. We forgo the use of high fidelity LOD models in favor of standard models. HD models are an order of magnitude greater in polygon counts than their standard counterparts. This would be very taxing for rendering our synthetic environment without sophisticated LOD management.

We note that small GTA5 vehicles have an average of 19,379 polygons (median of 18,921 polygons) and large vehicles have 19,322 polygons (median of 19,125 polygons). This shows the order of magnitude difference in mesh model fidelity between GTA5 and CityEngine. We populate the static environment by sampling from these GTA5 vehicles. This new population of vehicles is saved on a different layer in CityEngine's workflow than

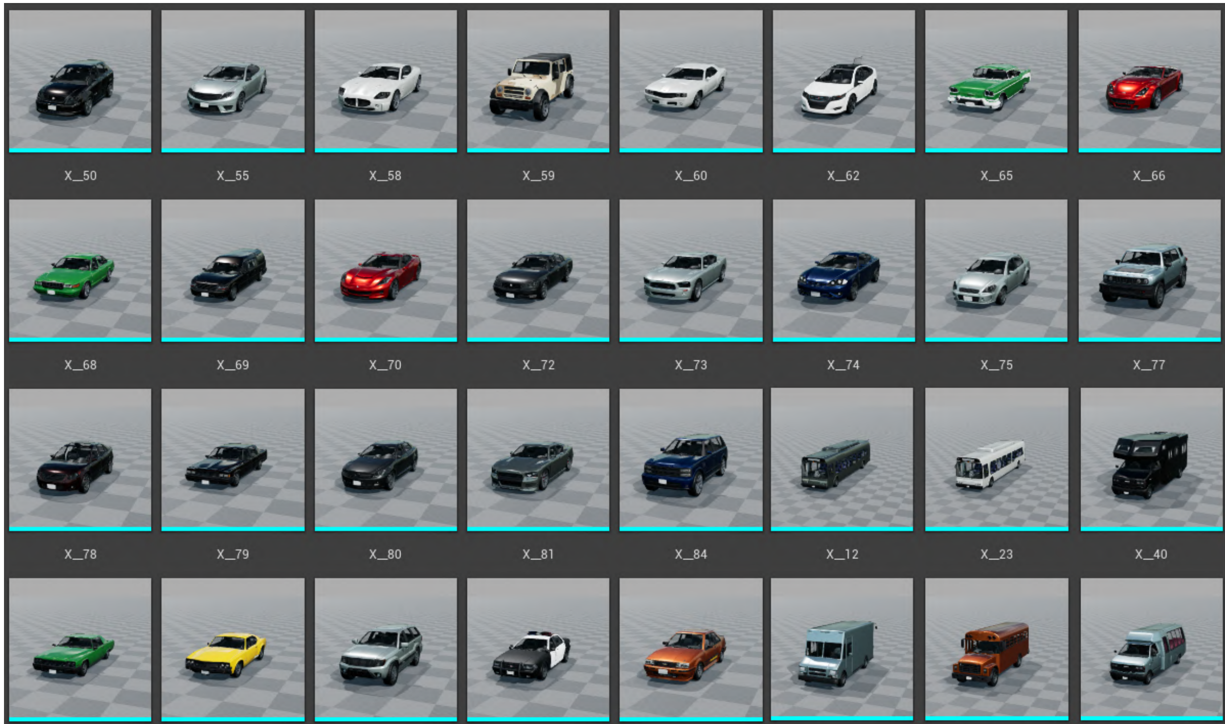


Figure 4.9: A sample of high fidelity vehicles from GTA5 game world

that generated by use of CityEngine vehicles.

4.2 Unreal Engine 4 Workflow

Once the CityEngine workflow layers have been completed, we are ready to progress to the next step in the pipeline — importing into UE4 (4.1c). Unreal Engine is a video game engine developed by Epic Games [66]. Initially released in 1998, the engine has gone through significant advancements in the past two decades to become one of the most utilized game engines in the world.

Developed since 2003, UE4 saw free-to-use, source-available release in 2015. This enabled students and researchers to jump on board the platform and design virtual worlds to fit their needs. One such research project serves as a key motivator for the use of UE4 in our research — the CARLA open-source simulator for autonomous driving research. This simulator is discussed in section 4.2.1.

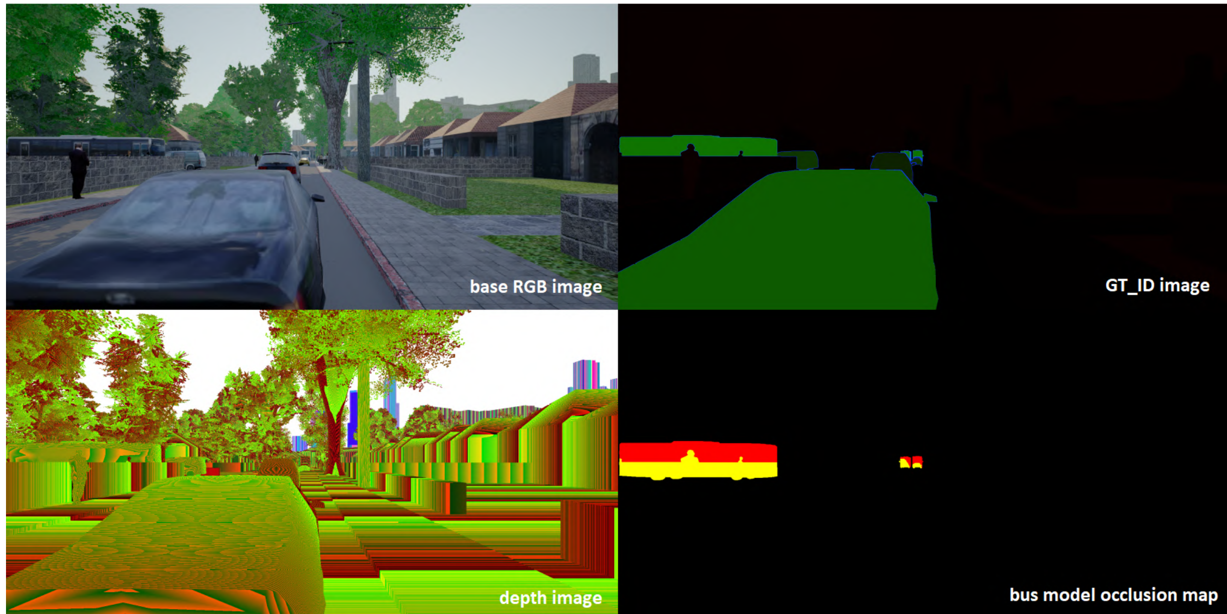


Figure 4.10: ProcSy dataset sample frame: (a) RGB image, (b) GTID image, (c) depth map, (d) 1 occlusion map [35]

CityEngine uses a traditional rendering approach with a specific set of goals catered towards its workflow. For instance, generating large city environments oftentimes requires handling a massive amount of polygons on screen — in the order of a few hundred million polygons. CityEngine does not focus on realistic lighting models. Instead, more emphasis is put towards a smoother, real-time workflow experience for its users. As such, asset materials and textures look rather primitive compared to the quality achievable with sophisticated rendering methods such as that provided by UE4. Among its many advancements, UE4 uses a physically based lighting and material workflow [34]. This is discussed in section 2.3.

4.2.1 Introduction to CARLA

In 2017, Dosovitskiy et al. introduced their work in developing an open-source simulator for autonomous driving research [15]. CARLA was created with the intent of supporting training and validation of autonomous driving systems. The CARLA project provides a plethora of freely usable digital assets such as buildings and vehicle models. CARLA also provides their server-client protocol and project source code.

4.2.2 Depth

UE4 provides access to the game world's depth buffer. CARLA has code to spawn depth camera object that can be invoked by Python-based server-client API. In terms of camera movement, this special camera can be a parasitic clone of the realistic game world RGB camera. However, its output generates Portable Network Graphics (PNG) files with depth encoded into the RGB channels.

UE4 stores its depth values with 32-bit floating point precision. Let us call this stored depth value d, which carries values within the range of $[0, 1]$. World depth z is represented in d by a linear transformation of the reciprocal $1/z$.

$$d = a\left(\frac{1}{z}\right) + b \tag{4.1}$$

In equation 4.1, a and b are constants that are representative of the near and far clipping planes of the rendered scene. Reciprocal world depth is used because it makes hardware rasterization a simple process due to the fact that straight lines are preserved by the transformation [58]. The projection matrix of $1/z$ can be easily chained with other matrix operations. On top of this, interpolation of d on polygons remains linear during screen space rasterization, hence easily computed.

Another point to note about the relation between z and d is that near and far clipping planes are in reversed order (figure 4.11). Far-away objects lie closer to 0 and nearby objects are represented by values closer to 1. Reversed Z buffer is a commonly used trick by game engines to avoid z-fighting that would otherwise be seen when distant polygons visibly flicker due to lack of depth resolution. There is an increased precision for values that are closer to 0, which lends itself well to discriminating depth values that are farther away. Values closer to the screen are easier to distinguish, hence this precision is not as necessary. So these values are mapped closer to 1, where less precision is permitted. This idea is explored in detail by Lapidous et al. [40].

When CARLA outputs depth value, it first scales the floating point value in the range of $[0, 1]$ to a 24-bit integer representation in the range of $[0, 255^3 - 1]$. This value is then split into RGB channels of a PNG image and stored alongside realistic and ground truth annotation images (figure 4.10c).

4.2.3 Ground Truth Annotations

We leverage the CARLA team's work on synthetic semantic segmentation annotations. We use version 0.9.2 of the CARLA project. UE4 makes use of many buffers that can be

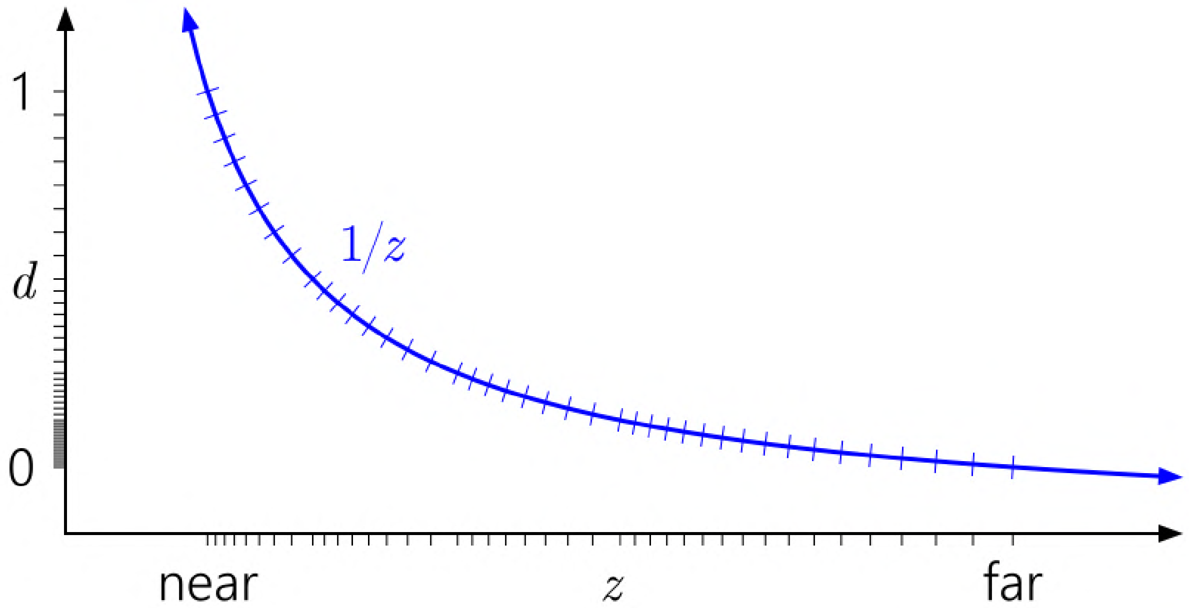


Figure 4.11: Visualization of increased precision of d around 0 and corresponding to the near plane as d approaches 1; note that ticks along y-axis are in alignment with corresponding ticks on the 1/z curve [58]

used for a variety of reasons. These include base color, depth, roughness, specular, surface normals, etc. Another buffer that UE4 exposes to developers is a custom stencil buffer. This lets developers tag game world objects with numeric IDs (figure 4.12).

The CARLA team saw potential in using the custom stencil buffer to automatically generate ground truth annotations for the corresponding realistically rendered frames. Essentially, they provide a simplified method by which assets organized into a specific folder structure get tagged with the associated identification value for that folder (figure 4.12).

Using a Python-based script, a semantic segmentation camera object is created similar to the aforementioned depth camera (section 4.2.2). This modified camera can generate ground truth annotations based on asset tags (figure 4.10b). A drawback of this method of annotation is the inability to have instance-level or panoptic segmentation [37].

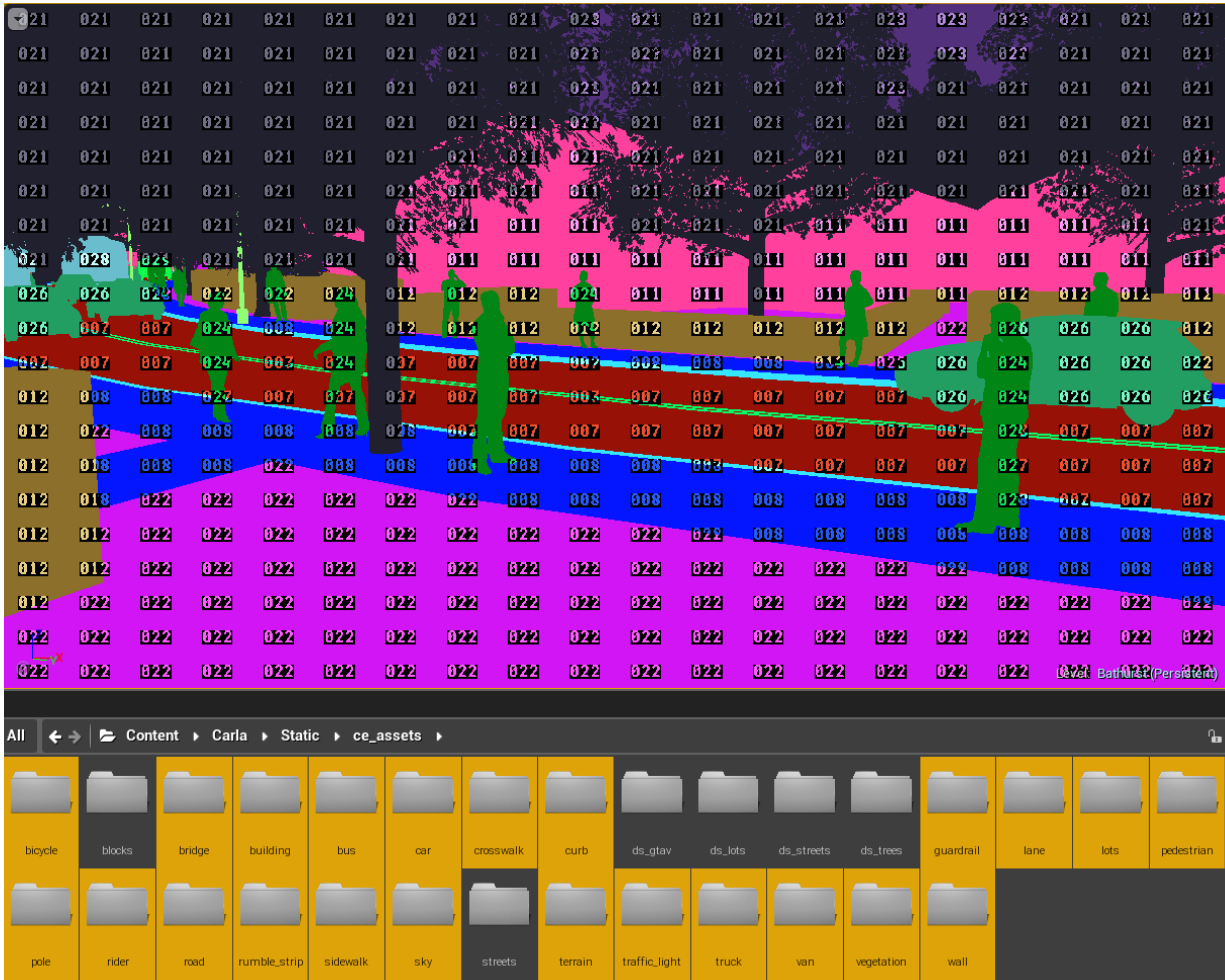


Figure 4.12: UE4 custom stencil buffer visualization showing asset tag values as predicated by folder organization; folders highlighted in yellow represent semantic classes that have tag associations in CARLA code modifications for ProcSy [15]

4.2.4 Vehicle Instances

We explore a method of generating instance-level segmentation for the vehicle class. Similar to the depth and custom stencil buffers, UE4 has the capability of rendering metallic material buffer output. With knowledge of UE4's usage of PBR materials (section 2.3), we modify the purpose of UE4's metallic buffer output for our needs. Unique vehicle mesh models are assigned different specularity values. This means that the metallic buffer output

for these vehicle mesh models would represent the specularity associated to them. With different specularity values, a different metallic buffer id is assigned to each vehicle model. All instances of a vehicle model share the same metallic buffer id. Thus, not every vehicle is uniquely represented in this way, rather they are seen in grouped instances (an example can be seen in the bus model occlusion map of figure 4.10).

As an added effort towards true instance segmentation, we create a post-effect edge detection filter material with a combination of Sobel and Laplacian edge detection algorithms [76]. The resulting edges can be used to further break up instance groups into individual instances. This is seen as the blue edge lines in figure 4.13's packed channel-packed ground truth annotations, vehicle per-model instance ids, and edge detections (GT_ID) image.

We encode ground truth annotations, vehicle per-model instance ids, and edge detections in the same PNG file (called the GT_ID image in Fig. 4.1e), where each takes up one of the RGB channels (figure 4.13). This is done as a space saving measure. For our dataset of 11,000 frames, ground truth annotations took around 786 MB of space. Output of vehicle instances camera produced another 171.2 MB worth of data. After RGB channel packing with PNG compression level 9 (loss-less compression format), we achieved a final memory footprint of only 553.6 MB. Channel packing compressed the storage footprint of our data to around 58% of its original space requirements.

4.2.5 Occlusion Maps

In UE4, there are two different depth buffers at our disposal. The scene depth buffer was mentioned in section 4.2.2. UE4 exposes a secondary depth buffer similar to its custom stencil buffer (section 4.2.3). This second depth buffer is appropriately called the custom depth buffer. We can manipulate objects to be visible or invisible in the custom depth buffer. We remove every world object from the custom depth buffer, then iterate over visibility of individual vehicle models in the custom depth buffer.

We create a post-effects filter material to generate occlusion maps of vehicle models by masking scene depth with only the region visible in custom depth. The occluded portion in this masked region corresponds to all pixels in scene depth that do not equal to the corresponding pixel in custom depth. The filter outputs the entire object-of-interest in the R channel. The G channel contains region of the scene that occludes the object-of-interest (figure 4.10). We create a CARLA special camera object that generates this occlusion map output in PNG format.

Using this post-effects material, we iterate over each of the unique vehicle models, make that vehicle model visible in custom depth buffer, output occlusions for each frame that

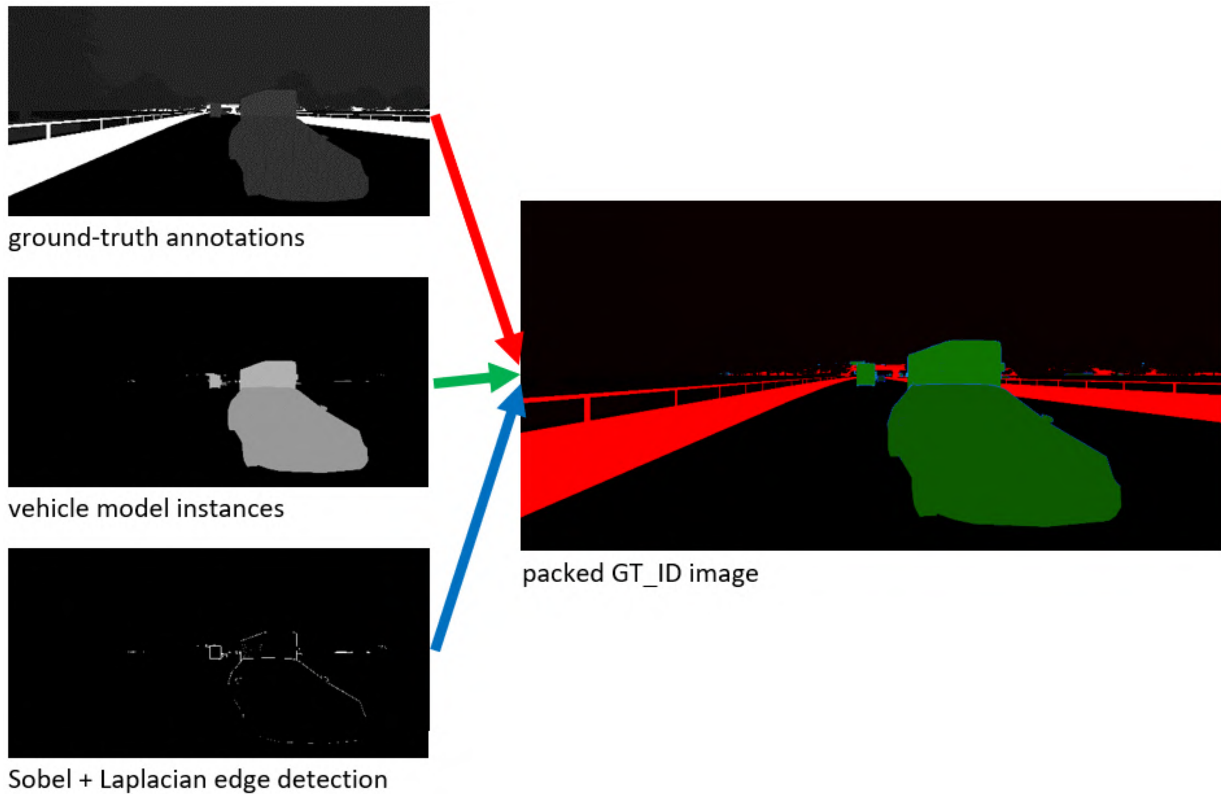


Figure 4.13: RGB channel packing of ProcSy dataset; R channel contains ground truth annotations (section 4.2.3); G channel contains vehicle model instancing (section 4.2.4); B channel contains edge detection

contains that vehicle model, then again remove that vehicle model from the custom depth buffer. For each frame of our ProcSy dataset, we have a folder containing n occlusion maps, where n is the number of different vehicle models that appear in frame. We ignore objects that are farther than 10,000 game world units from the screen. These occlusion maps are used towards our experimentation of the ProcSy dataset (section 5.3).

4.2.6 Data Selection

From the static world that is generated, images can be captured from anywhere. We are only interested in images that are captured from a road-driving perspective. So, there is a need to add logic to the simulator/data collection pipeline such that scenes that are

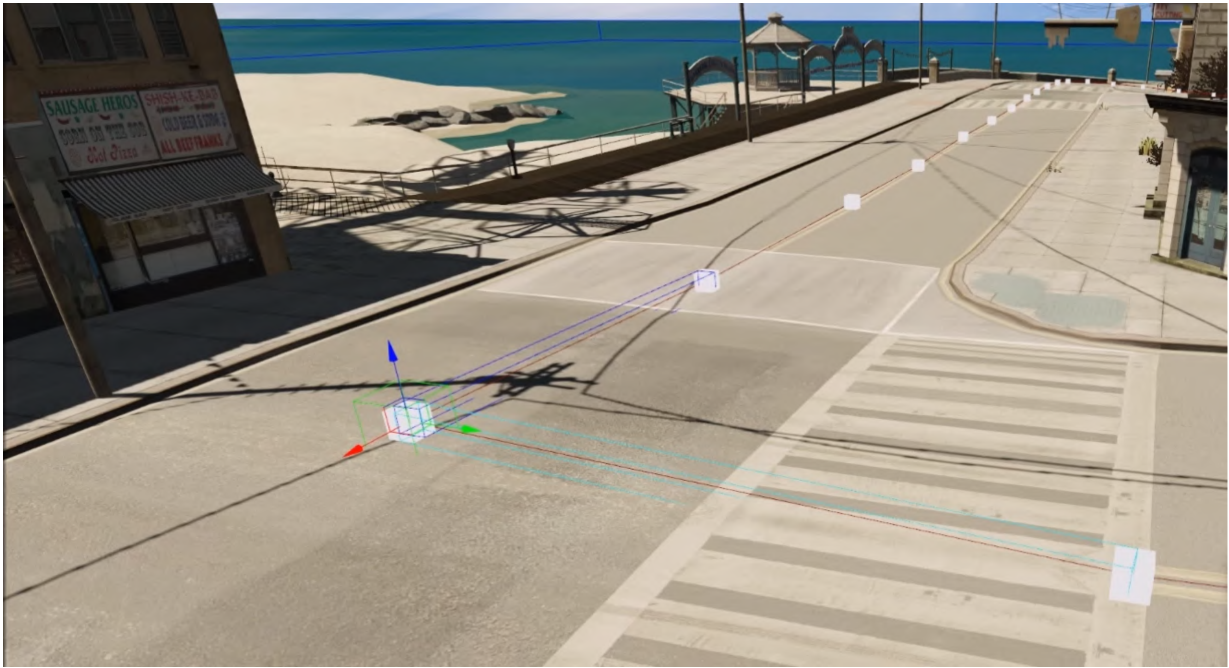


Figure 4.14: Path nodes graph in sandbox games such as GTA5

captured only acquire road scene imagery.

A robust way to ensure road-facing camera placements is by creating a spline graph of the entire road network. This can serve multiple purposes. Using UE4 Blueprints, data collection cameras can be spawned along the spline graph. These spline graphs can also serve as routes for traffic simulation in the environment. In fact, this is how macroscopic traffic simulation is handled in sandbox games like GTA5 (figure 4.14).

Manual spline creation for camera placements can be a very time-consuming process. It is essentially the same task as that outlined in section 4.1.1. Instead, another approach can be taken, whereby CityEngine exports path metadata along with the environment models. However, this would require significant know-how of the CityEngine custom CGA scripting language. So this method of spline creation has been kept out of scope for this thesis.

Ultimately, the approach taken to generate camera spawn locations relies on a binary mask of the 3km^2 region's road map (figure 4.15). This binary mask is derived from a top-down render of the region in UE4 Editor. In its captured resolution of 4545×4541 pixels, there are 1,349,841 distinguishable white pixels that can be chosen. These white

pixels represent road-going locations of the map.

At each white pixel, we need to find good scenes that are road-facing. It is of no use if the camera is on a white pixel, but is facing towards a building wall on the side of the road. We derive a brute-force, line-traced search to find the top 2 optimal orientations at each white pixel that ensures road-scene visibility.

The line trace algorithm works as such. For each white pixel, for each angle between 0° and 360°, we trace a line starting from the pixel going in the direction of angle until a non-white pixel is reached. By keeping track of the lengths of these line traces, we can determine the angles with the most promising road-scene view. A longer, unobstructed line track is assumed to be a more promising orientation.

Note that an underlying assumption is that the map is a flat terrain. Data collection using this binary mask approach does not work well if map regions are generated using elevation data priors as discussed in section 4.1.1. Also, in our map region, the overpass areas and associated on/off ramps are problematic data collection regions. We simply filter out data generated from these regions. This method is also not designed to allow for temporal data collection. With an extended research timeline, the more robust spline graph method should be explored for camera spawn points.

From the established set of 1,349,841 road scenes, uniform random samples were taken. These scenes were quickly rendered using a low resolution render pass. With a human-in-the-loop identification process, bad scenes were discarded. A scene was defined as bad if it caused clipping issues with static vehicles, pedestrians, or other assets placed in the environment. Once 11,000 clean frames were identified, this was established to be the core of the ProcSy dataset.

The ProcSy dataset was split into a training set of 8000 frames, a validation set of 2000 frames, and a test set of 1000 frames. In Fig. 4.15, frames from the bottom-right quadrant were used for validation and test sets. Frames from the other 3 quadrants were used for the training set. We rendered the entire dataset with Cityscapes resolution (2048×1024 pixels) and annotation scheme. For each of our rendered frame, we have the following output: RGB image (2.5mb), GT_ID image (50kb), depth map (790kb), and an occlusion maps folder (approx. 60kb) containing n images where n refers to the number of unique vehicles that appear in the frame (Fig. 4.1e).

4.2.7 Influence Factors Variations

A clear strength of synthetic semantic segmentation datasets, more specifically those based in simulators, is the ability to control and bias the dataset to fit specific needs (figure 4.16).

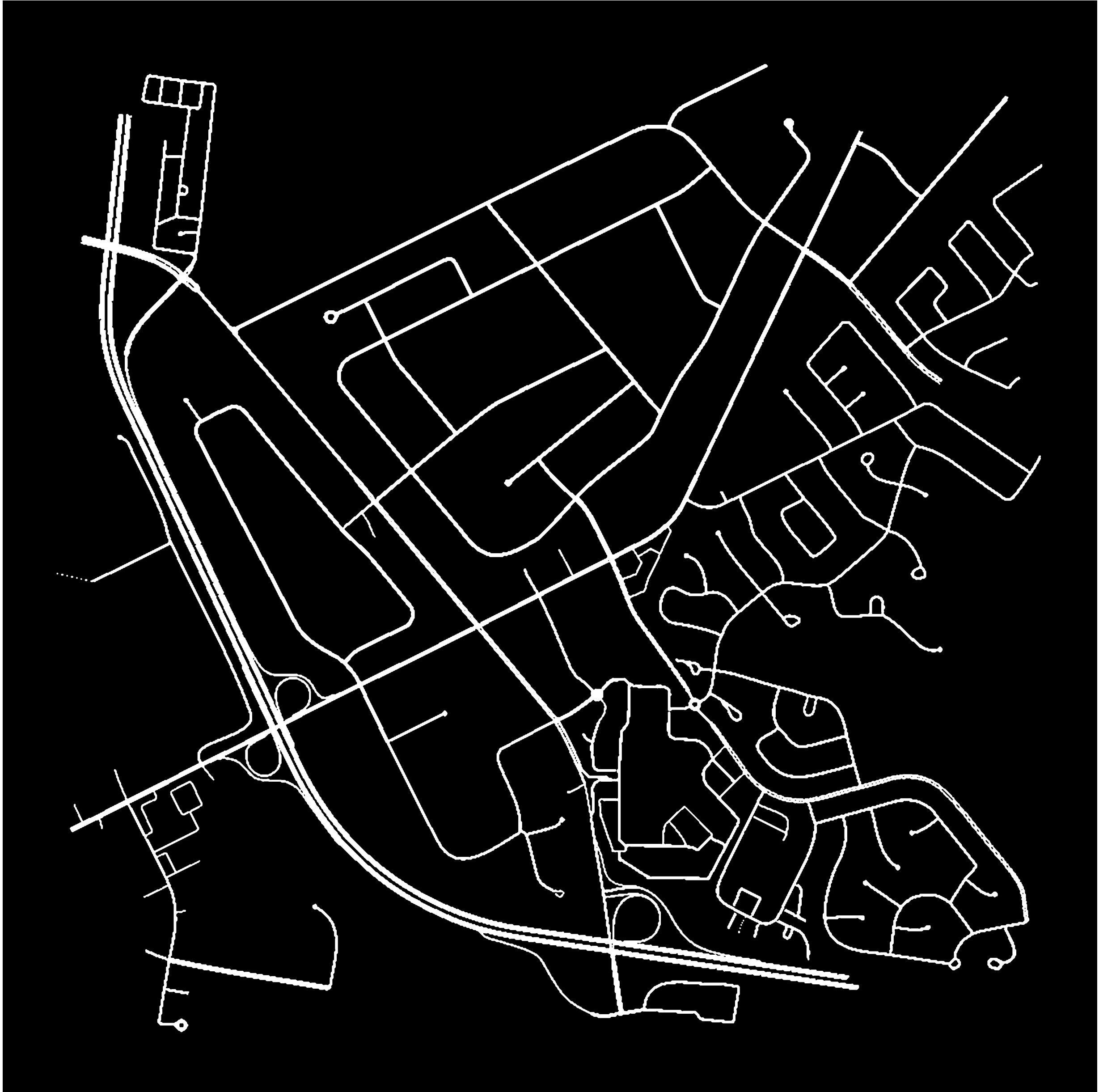


Figure 4.15: Binary mask representing road-going pixels (white) of the 3 km^2 map region-of-interest (figure 4.2) in top-down view [35]

Figure 4.16: Weather variations showing intensity levels in 3 categories — rain, cloud, and puddles; the variation ranges explored are 25%, 50%, 75%, and 100%; note that the scene is captured with the Sun positioned at a high-noon angle, hence causing a visible glare on the car near center of screen

For the ProcSy dataset, we vary the Sun light and weather effects. These are termed as influence factors. Our key experimental goals is to study the robustness of a neural network against quantified variations of influence factors (further discussed in section 5.3).

We use CARLA's capability to generate depth maps (section 4.2.2) and occlusion maps for vehicle class (section 4.2.5). Along with these gbuffer-based influence factors, we also focus on variations in environmental factors. UE4 employs energy-conservation based lighting models and PBR-based rendering techniques (section 2.3). This allows for easy access to a level of realism in rendered scenes that is simply not possible with the rendering engine of CityEngine.

CityEngine has a very different goal for its UI. Users need to be able to see macroscopic views of a city-scale group of polygons. The polygon count on screen can regularly be upwards of a million. At this scale, the user is not interested in a realistic and expensive lighting model. CityEngine uses a simplistic BRDF and rendering solution to allow for a smooth workflow for its users. UE4's BRDF is far more robust towards achieving better

realism of scenes.

CARLA, as of version 0.9.1, introduced server-client API calls to easily modify weather parameters and Sun position in real-time (Fig. 4.1d). For ProcSy, we selected three weather influence factors, namely rain, cloud, and puddle deposits (accumulation of water on road pavement). For each of these factors, we generate data for five different intensity levels of 0%, 25%, 50%, 75%, and 100%. These quantities are based on CARLA's predefined range values for the weather parameters.

We also use the Sun's position in the sky as another influence factor (Fig. 4.17), consisting of the Sun's azimuth and altitude angles. In order to reduce the amount of variations to study, we first make note of positions in a road scene frame where the Sun's angle and coincident shadow effects are expected to have a meaningful impact. We note that below the horizon the Sun's angle is irrelevant. A typical road scene is expected to have buildings or other architecture along the left and right sides. These generally approach a vanishing point near center of the frame. Therefore, we identify eight Sun positions within the frame that represent a V-shape in upper half of the frame.

We also consider four Sun positions outside the frame. The first of these is front and above (representing a high noon Sun angle). Another is off to the left casting harsh shadows to the right of scene objects. Another is off to the right, casting harsh shadows to the left. The last Sun position outside the view of the frame is above and behind the camera, casting forwards shadows of objects.

With the identified environmental influence factor variations, we have the ability to create $5 \times 5 \times 5 \times 12 = 1500$ unique RGB images for each frame of our dataset. In total, ProcSy dataset has the potential to grow to more than $1{,}500 \times 1{,}000{,}000 = 1{,}500{,}000{,}000$ frames of data (this is assuming about 34.98% of the scenes captured from binary mask data are bad scenes). Rendering all these scenes can be a time consuming process. So we choose to render specific subsets of this variational influence factor data based on our experimental needs.

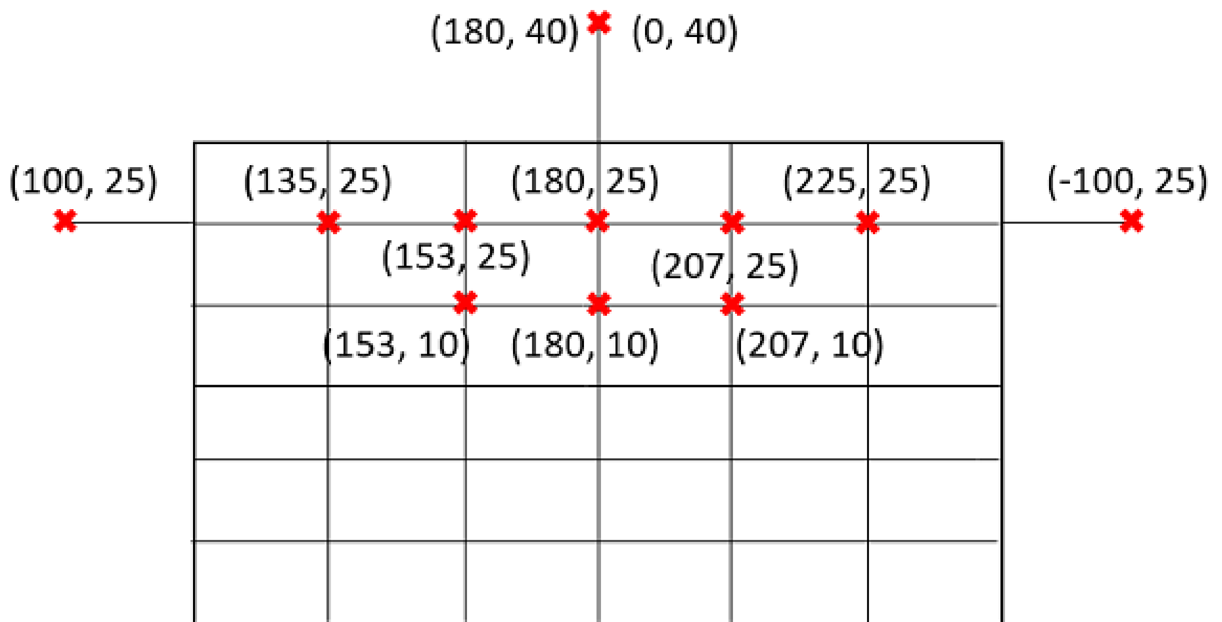


Figure 4.17: Road scene frame approximately showing the Sun positions in and out of frame that are considered; red x's indicate Sun location and corresponding tuples show azimuth and altitude values used in CARLA [35]

Chapter 5

Experimental Analysis

This chapter studies various facets of synthetic dataset generation through experimental analysis. First, we identify the problem of domain shift to the real world (our operational design domain (ODD) is Cityscapes dataset). We explore fine-tuning methods to mitigate for this.

We then compare strengths and weaknesses of URSA and ProcSy. We make an argument for ProcSy justifying its utility despite having lower geometric fidelity overall than URSA.

Ultimately, we explore 3 experimental setups towards understanding effects of influence factors on unimodal semantic segmentation model performance. Environmental influence factors used are amounts of rain, cloud, and puddles. We study effects of depth and occlusion as well. DeepLab v3+ is used as the testbench for these experiments.

5.1 Adapting to Cityscapes

The ultimate target of semantic segmentation networks is to perform reliably on a real-world operational design domain. For our research, the scope of this real-world ODD is the Cityscapes dataset.

We explore performance of DNNs trained purely on synthetic datasets. We identify the domain gap and reflect on a way to better account for class distribution differences between training and testing datasets.

Analysis and discussion follow regarding adaptation to Cityscapes dataset domain via fine-tuning. We experiment with fine-tuning using various proportions and iterations of Cityscapes training set.

5.1.1 Class Distribution

The objective of checking class distribution of datasets is to understand how closely the dataset represents a real-world ODD. In the real world, a DNN may be exposed to any distribution of classes depending on usage patterns and environmental biases. Thus, generalizing the class distribution of a real-world driving environment is an intractable problem.

To simplify the problem scope to something more quantifiable, we study class distribution of a pre-generated, real-world dataset containing scenery distribution with as much unbiased data as possible. We use Cityscapes for this purpose, because it contains aggregate data from 50 different European cities. Cityscapes' ground truth annotation data, although small (only 5000 images), is well-accepted and relied upon by the research community (over 1700 citations as of this writing). The class distribution of Cityscapes dataset can be seen in figure 5.1a. Note that roads, buildings, vegetation, and cars make up over 82% of pixels in Cityscapes dataset.

In comparison to Cityscapes, our generated datasets achieve much better class balancing. In figure 5.1, it is seen that URSA and ProcSy both have significantly reduced exposure to buildings, whereas sky is seen more prominently. Cityscapes dataset was collected in European cities with narrow roadways and buildings that are close together. In contrast, URSA and ProcSy are based on North American roadways. URSA/GTA5 is a mockery of real-world Los Angeles, while ProcSy uses a region of Waterloo, Ontario as a reference. In North American ODD, roadways tend to be much wider and buildings more spread out than the European counterparts — explaining the prominency of sky class in our datasets.

5.1.2 Closeness to Cityscapes

To validate synthetic datasets' closeness to Cityscapes dataset, we train FCN, SegNet, and DeepLab v3+ models (networks described in section 2.1.3). In order to allow for the most unbiased performance analysis, we use hyper-parameter configurations for these networks as recommended by the respective authors. We use Cityscapes validation set [11] as the testing set for our experimental analysis. This validation set consists of 500 labeled frames. FCN and SegNet are trained for 100,000 iterations each. DeepLab is only trained for 50,000

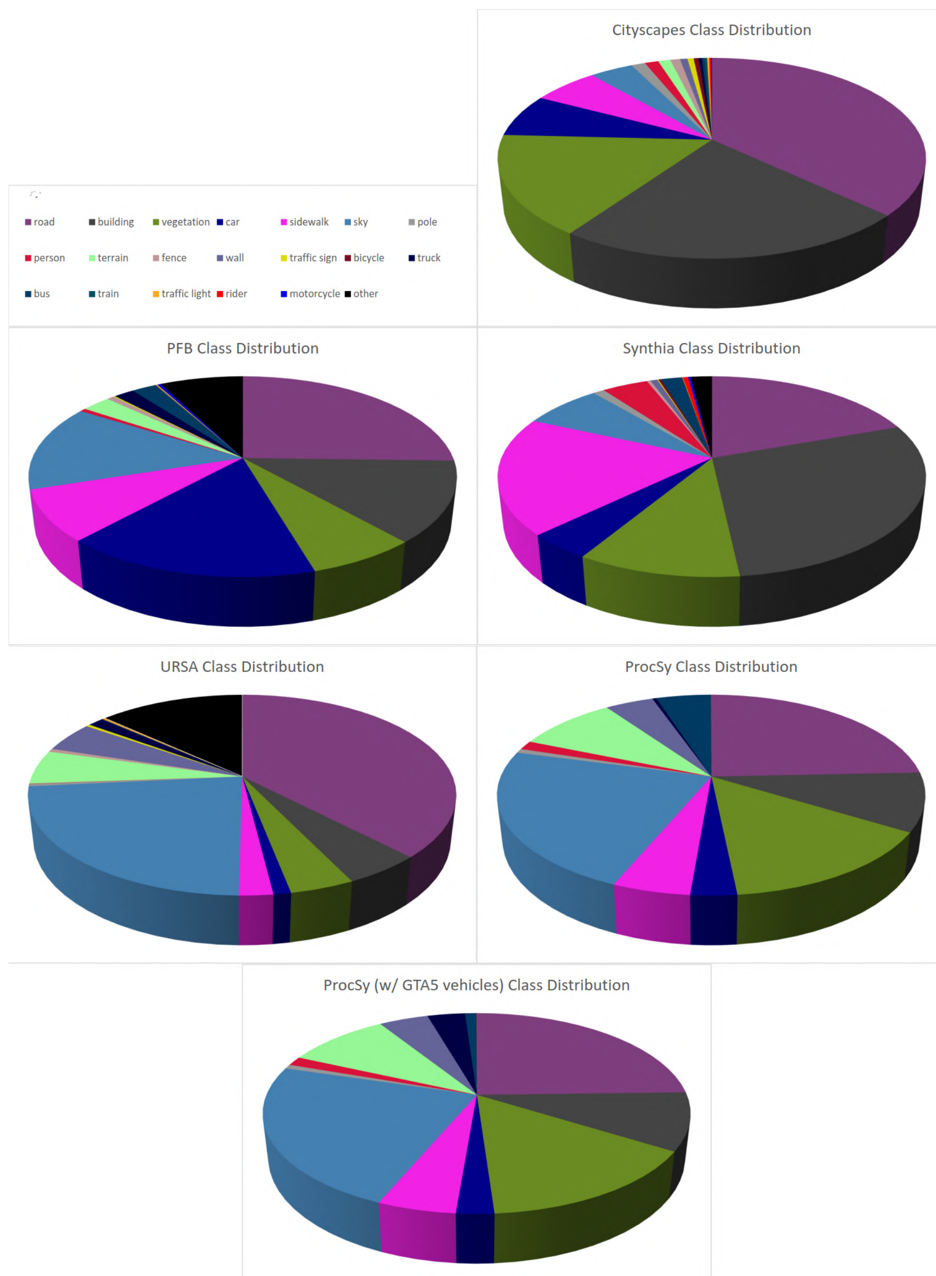


Figure 5.1: Dataset distribution of 19 classes

iterations. The results of training purely on synthetic datasets and testing on Cityscapes validation set are shown in table 5.1.

	FCN				SegNet				DeepLab			
classes	CS	SY	PFB	URSA	CS	SY	PFB	URSA	CS	SY	PFB	ProcSy
road	0.866	0.023	0.214	0.508	0.833	0.009	0.065	0.000	0.978	0.034	0.216	0.039
building	0.727	0.284	0.386	0.472	0.776	0.271	0.452	0.329	0.916	0.451	0.410	0.413
vegetation	0.813	0.315	0.409	0.217	0.842	0.589	0.566	0.153	0.918	0.492	0.542	0.343
car	0.832	0.408	0.560	0.291	0.723	0.238	0.191	0.199	0.936	0.438	0.326	0.357
sidewalk	0.600	0.112	0.216	0.115	0.464	0.034	0.071	0.004	0.826	0.084	0.116	0.078
sky	0.184	0.601	0.596	0.190	0.807	0.548	0.347	0.242	0.942	0.513	0.393	0.430
pole	0.305	0.076	0.000	0.085	0.406	0.098	0.000	0.022	0.594	0.165	0.098	0.074
person	0.614	0.371	0.372	0.295	0.522	0.206	0.306	0.238	0.772	0.360	0.303	0.262
terrain	0.230	0.000	0.070	0.039	0.301	0.000	0.055	0.011	0.589	0.000	0.049	0.033
fence	0.211	0.000	0.047	0.020	0.170	0.000	0.087	0.007	0.560	0.000	0.125	0.000
wall	0.224	0.007	0.000	0.035	0.078	0.005	0.000	0.012	0.503	0.077	0.067	0.040
traffic sign	0.506	0.066	0.077	0.025	0.502	0.031	0.229	0.019	0.725	0.111	0.071	0.000
bicycle	0.588	0.036	0.000	0.000	0.517	0.001	0.000	0.000	0.732	0.031	0.000	0.000
truck	0.213	0.000	0.065	0.008	0.075	0.000	0.004	0.004	0.603	0.000	0.024	0.015
bus	0.454	0.041	0.032	0.023	0.154	0.019	0.019	0.002	0.772	0.085	0.095	0.007
train	0.274	0.000	0.000	0.000	0.042	0.000	0.000	0.000	0.600	0.000	0.000	0.000
traffic light	0.293	0.008	0.102	0.014	0.314	0.027	0.089	0.018	0.622	0.031	0.098	0.000
rider	0.270	0.040	0.000	0.000	0.174	0.000	0.000	0.000	0.541	0.046	0.021	0.000
motorcycle	0.326	0.001	0.091	0.002	0.122	0.000	0.035	0.000	0.489	0.019	0.049	0.000
mIoU	**0.449**	**0.126**	**0.170**	**0.123**	**0.412**	**0.109**	**0.132**	**0.066**	**0.717**	**0.155**	**0.158**	**0.110**

Table 5.1: Experiment results for section 5.1.2; CS columns are for reference, these are trained on Cityscapes training set

The rows of table 5.1 are ordered by decreasing class frequency in Cityscapes dataset. We see that Deeplab performs noticeably well when trained with ODD data (71.7% mIoU). An interesting observation is that the trend of class accuracies in this column roughly reflects the frequency order of classes in Cityscapes dataset.

More generally, for all of the DNNs trained on ODD dataset, it is seen that road, building, vegetation, and car classes rank amongst the top-5 class accuracies for that network. These are the most frequent classes in the Cityscapes dataset. As we noted in section 5.1.1, these classes make up over 82% of pixels in Cityscapes dataset. So ODD-trained DNNs become heavily biased towards the 4 classes.

Another observation is with regards to the standard metric for measuring semantic segmentation network performance, the mIoU. This metric presents equal importance to performance of all classes, regardless of frequency in the training dataset. This is problematic in using a different training domain. For example, none of the synthetic datasets

contain any instances of trains (the transportation medium, not to be confused with training). Yet, this has a significant penalty on overall mIoU. While mIoU can be useful in identifying missing class representations during domain shift, this metric does not capture a true domain shift story of classes that are represented in the training dataset domain.

Table 5.2 shows the same data points using an alternate metric. fwIoU takes into account the presence of classes in training dataset. Whereas mIoU is a simple mean of the class accuracies, fwIoU weighs class accuracies based on their appearance frequency in the data [75]. With the exception of SegNet trained on URSA, we see that our datasets demonstrate performances similar to those of other synthetic datasets.

SegNet ranks ahead of FCN in terms of Cityscapes-trained model performance (fwIoU of 75.3% vs CS-FCN's fwIoU of 74.4%). However, all synthetically trained SegNet models have worse-performing fwIoU scores than the FCN counterparts. The behavior of URSA-SegNet is also very unusual (fwIoU of only 8.3%). The ODD of URSA — GTA5 game world — is the same as PFB, and training hyper-parameters and iterations were ensured to be the same for both cases. Yet, PFB-SegNet performs at a more respectable fwIoU score of 20.5%. The other DNNs do not demonstrate this sort of behaviour on our generated datasets. We conclude that the SegNet architecture itself has poor generalization across different dataset domains. This is a reflection of a smaller trainable parameter space in SegNet relative to the other networks. We see in section 5.1.3 that fine-tuning with portions of Cityscapes dataset is able to mitigate this.

5.1.3 Insights on Fine-Tuning

Fine tuning on the target dataset is a very simple way to adapt to the target domain. We use Cityscapes training set of 2975 frames for fine-tuning purposes. We study DNN fine-tuning along two trajectories.

First, we fine-tune on the synthetically-trained DNNs for a fixed number of iterations (10,000) with various portions of the Cityscapes dataset (25%, 50%, 75%, 100%). We observer the effects with regards to network mIoU and fwIoU scores.

We then fine-tune the original synthetically-trained DNNs with varying number of iterations (10,000 or 40,000), while using just around 10% of Cityscapes training dataset (300 frames).

	FCN				SegNet				DeepLab			
classes	CS	SY	PFB	URSA	CS	SY	PFB	URSA	CS	SY	PFB	ProcSy
road	0.320	0.004	0.054	0.192	0.308	0.002	0.017	0.000	0.362	0.007	0.055	0.003
building	0.165	0.082	0.050	0.024	0.176	0.078	0.059	0.017	0.208	0.130	0.053	0.037
vegetation	0.131	0.033	0.029	0.009	0.136	0.061	0.040	0.006	0.148	0.051	0.038	0.057
car	0.058	0.017	0.091	0.003	0.050	0.010	0.031	0.002	0.065	0.018	0.053	0.002
sidewalk	0.036	0.021	0.018	0.002	0.028	0.006	0.006	0.000	0.049	0.016	0.010	0.001
sky	0.007	0.043	0.083	0.044	0.032	0.039	0.048	0.056	0.037	0.037	0.055	0.117
pole	0.004	0.001	0.000	0.000	0.005	0.001	0.000	0.000	0.008	0.002	0.000	0.000
person	0.008	0.015	0.002	0.000	0.006	0.009	0.002	0.000	0.009	0.015	0.002	0.001
terrain	0.003	0.000	0.002	0.002	0.003	0.000	0.001	0.001	0.007	0.000	0.001	0.006
fence	0.002	0.000	0.000	0.000	0.001	0.000	0.001	0.000	0.005	0.000	0.001	0.000
wall	0.001	0.000	0.000	0.002	0.001	0.000	0.000	0.001	0.003	0.000	0.000	0.001
traffic sign	0.003	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.004	0.000	0.000	0.000
bicycle	0.003	0.000	0.000	0.000	0.002	0.000	0.000	0.000	0.003	0.000	0.000	0.000
truck	0.001	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.000
bus	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.002	0.002	0.002	0.000
train	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000
traffic light	0.001	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.001	0.000	0.000	0.000
rider	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000
motorcycle	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
fwIoU	**0.744**	**0.217**	**0.332**	**0.279**	**0.753**	**0.206**	**0.205**	**0.083**	**0.916**	**0.278**	**0.270**	**0.226**

Table 5.2: fwIoU metric analysis for section 5.1.2 experiment

Fixed Training Iterations with Variable Percentage of Cityscapes Training Dataset

Table 5.3 demonstrates the results of fine-tuning FCN models that were trained with synthetic datasets (Synthia, PFB, and URSA). Table 5.4 demonstrates the results of fine-tuning SegNet models that were trained with the same synthetic datasets. In these tables, the bolded fwIoU scores are a weighted average of Cityscapes and the training dataset's class distributions. This is to reflect the class distribution that is seen by the networks during training. Cityscapes is given a weight of 1/11 since there are 10,000 fine-tuning iterations over a total of 110,000 iterations. Likewise, the original training dataset is given a weight of 10/11 due to 100,000 iterations of initial training. This does not factor in effects of recency bias during the networks' fine-tuning iterations. So we also calculate fwIoU_CS and fwIoU_training to provide a range where true fwIoU would be found.

We notice that fine-tuning with just 25% of Cityscapes training set for 10,000 iterations drastically improves all DNN models' mIoU and fwIoU scores. The problematic URSA-SegNet model shows especially impressive results as its mIoU and fwIoU increase by 460% and 641% respectively. There is unanimous improvement on both networks by introducing

	Train Only				FT-25%			FT-50%			FT-75%			FT-100%		
class name	CS	SY	PFB	URSA	SY	PFB	URSA	SY	PFB	URSA	SY	PFB	URSA	SY	PFB	URSA
road	0.866	0.023	0.214	0.508	0.728	0.875	0.902	0.807	0.893	0.915	0.812	0.882	0.917	0.814	0.882	0.911
building	0.727	0.284	0.386	0.472	0.721	0.759	0.758	0.731	0.776	0.764	0.721	0.771	0.758	0.732	0.767	0.762
vegetation	0.813	0.315	0.409	0.217	0.655	0.815	0.853	0.728	0.844	0.855	0.730	0.829	0.849	0.732	0.833	0.856
car	0.832	0.408	0.560	0.291	0.767	0.808	0.798	0.785	0.812	0.810	0.795	0.823	0.820	0.799	0.829	0.823
sidewalk	0.600	0.112	0.216	0.115	0.500	0.558	0.538	0.526	0.558	0.564	0.536	0.585	0.580	0.538	0.599	0.586
sky	0.184	0.601	0.596	0.190	0.388	0.376	0.487	0.388	0.455	0.486	0.286	0.423	0.426	0.336	0.415	0.422
pole	0.305	0.076	0.000	0.085	0.261	0.251	0.272	0.280	0.254	0.279	0.287	0.270	0.292	0.287	0.269	0.292
person	0.614	0.371	0.372	0.295	0.609	0.600	0.579	0.584	0.587	0.584	0.604	0.593	0.594	0.611	0.606	0.598
terrain	0.230	0.000	0.070	0.039	0.218	0.216	0.258	0.281	0.207	0.324	0.324	0.214	0.319	0.307	0.196	0.233
fence	0.211	0.000	0.047	0.020	0.172	0.158	0.183	0.243	0.222	0.246	0.249	0.215	0.263	0.252	0.239	0.279
wall	0.224	0.007	0.000	0.035	0.145	0.172	0.198	0.158	0.173	0.189	0.169	0.189	0.189	0.182	0.201	0.205
traffic sign	0.506	0.066	0.077	0.025	0.393	0.419	0.358	0.432	0.442	0.387	0.448	0.441	0.396	0.455	0.457	0.407
bicycle	0.588	0.036	0.000	0.000	0.514	0.574	0.545	0.491	0.565	0.560	0.502	0.566	0.560	0.525	0.580	0.569
truck	0.213	0.000	0.065	0.008	0.128	0.195	0.224	0.136	0.220	0.242	0.153	0.248	0.266	0.154	0.260	0.289
bus	0.454	0.041	0.032	0.023	0.147	0.247	0.236	0.174	0.271	0.202	0.203	0.322	0.277	0.219	0.330	0.338
train	0.274	0.000	0.000	0.000	0.066	0.100	0.207	0.136	0.214	0.246	0.165	0.233	0.289	0.181	0.250	0.276
traffic light	0.293	0.008	0.102	0.014	0.183	0.217	0.196	0.181	0.234	0.235	0.175	0.238	0.226	0.166	0.218	0.222
rider	0.270	0.040	0.000	0.000	0.253	0.171	0.138	0.239	0.185	0.174	0.229	0.184	0.165	0.233	0.179	0.164
motorcycle	0.326	0.001	0.091	0.002	0.227	0.300	0.292	0.200	0.305	0.288	0.213	0.317	0.306	0.224	0.326	0.310
mIoU	**0.449**	**0.126**	**0.170**	**0.123**	**0.372**	**0.411**	**0.422**	**0.395**	**0.432**	**0.439**	**0.400**	**0.439**	**0.447**	**0.408**	**0.444**	**0.449**
fwIoU	**0.744**	**0.217**	**0.332**	**0.279**	**0.612**	**0.641**	**0.597**	**0.645**	**0.662**	**0.606**	**0.640**	**0.659**	**0.595**	**0.649**	**0.660**	**0.589**
fwIoU (CS)					0.659	0.754	0.773	0.707	0.774	0.784	0.705	0.768	0.782	0.711	0.769	0.782
fwIoU (training)					0.607	0.630	0.579	0.639	0.651	0.589	0.634	0.648	0.576	0.642	0.649	0.570

Table 5.3: Results of FCN fine-tuning experiments on Cityscapes (CS), Synthia (SY), Playing for Benchmarks (PFB), and URSA

	Train Only				FT-25%			FT-50%			FT-75%			FT-100%		
class name	CS	SY	PFB	URSA	SY	PFB	URSA	SY	PFB	URSA	SY	PFB	URSA	SY	PFB	URSA
road	0.833	0.009	0.065	0.000	0.739	0.650	0.815	0.676	0.694	0.759	0.610	0.604	0.684	0.641	0.659	0.695
building	0.776	0.271	0.452	0.329	0.726	0.702	0.754	0.714	0.734	0.768	0.730	0.735	0.772	0.717	0.736	0.773
vegetation	0.842	0.589	0.566	0.153	0.778	0.815	0.843	0.735	0.784	0.828	0.710	0.745	0.823	0.702	0.748	0.813
car	0.723	0.238	0.191	0.199	0.635	0.675	0.684	0.681	0.692	0.675	0.695	0.642	0.694	0.638	0.668	0.700
sidewalk	0.464	0.034	0.071	0.004	0.403	0.372	0.447	0.347	0.351	0.443	0.325	0.298	0.422	0.334	0.327	0.429
sky	0.807	0.548	0.347	0.242	0.765	0.723	0.800	0.693	0.812	0.776	0.694	0.773	0.813	0.675	0.787	0.834
pole	0.406	0.098	0.000	0.022	0.380	0.312	0.362	0.384	0.309	0.387	0.397	0.304	0.397	0.390	0.303	0.399
person	0.522	0.206	0.306	0.238	0.523	0.551	0.516	0.506	0.556	0.486	0.480	0.556	0.518	0.466	0.572	0.520
terrain	0.301	0.000	0.055	0.011	0.112	0.147	0.184	0.089	0.163	0.161	0.066	0.130	0.122	0.081	0.140	0.120
fence	0.170	0.000	0.087	0.007	0.154	0.176	0.109	0.166	0.206	0.110	0.190	0.206	0.117	0.185	0.213	0.129
wall	0.078	0.005	0.000	0.012	0.093	0.045	0.068	0.101	0.057	0.084	0.098	0.053	0.078	0.098	0.056	0.074
traffic sign	0.502	0.031	0.229	0.019	0.364	0.493	0.390	0.362	0.483	0.397	0.444	0.498	0.445	0.449	0.508	0.459
bicycle	0.517	0.001	0.000	0.000	0.481	0.324	0.397	0.545	0.338	0.382	0.553	0.334	0.406	0.553	0.363	0.388
truck	0.075	0.000	0.004	0.004	0.037	0.041	0.049	0.005	0.070	0.056	0.007	0.063	0.056	0.009	0.079	0.071
bus	0.154	0.019	0.019	0.002	0.048	0.062	0.061	0.096	0.100	0.074	0.122	0.109	0.155	0.118	0.117	0.182
train	0.042	0.000	0.000	0.000	0.000	0.000	0.036	0.000	0.000	0.028	0.000	0.000	0.078	0.000	0.000	0.085
traffic light	0.314	0.027	0.089	0.018	0.242	0.260	0.318	0.317	0.311	0.354	0.351	0.323	0.372	0.361	0.351	0.382
rider	0.174	0.000	0.000	0.000	0.171	0.131	0.084	0.204	0.170	0.077	0.228	0.184	0.170	0.245	0.205	0.172
motorcycle	0.122	0.000	0.035	0.000	0.128	0.156	0.113	0.159	0.135	0.113	0.182	0.110	0.143	0.187	0.124	0.147
mIoU	**0.412**	**0.109**	**0.132**	**0.066**	**0.357**	**0.349**	**0.370**	**0.357**	**0.367**	**0.366**	**0.362**	**0.351**	**0.382**	**0.360**	**0.366**	**0.388**
fwIoU	**0.753**	**0.206**	**0.205**	**0.083**	**0.627**	**0.574**	**0.614**	**0.593**	**0.602**	**0.588**	**0.578**	**0.558**	**0.565**	**0.577**	**0.582**	**0.574**
fwIoU (CS)					0.681	0.648	0.734	0.646	0.671	0.712	0.621	0.623	0.687	0.624	0.649	0.691
fwIoU (training)					0.621	0.567	0.602	0.588	0.596	0.575	0.574	0.551	0.553	0.573	0.575	0.562

Table 5.4: Results of SegNet fine-tuning experiments on Cityscapes (CS), Synthia (SY), Playing for Benchmarks (PFB), and URSA

just 25% of Cityscapes training data for fine-tuning.

In contrast, the successive additions of 50%, 75%, and 100% of Cityscapes training data demonstrate next to little effect on the DNNs' performance scores. Performance scores even see a reduction in the case of SegNet. This can be attributed to SegNet's smaller parameter space. fwIoU scores for all fine-tuned models tend to stagnate around the fwIoU of the respective DNN's 25% fine-tuned model.

Variable Training Iterations with Fixed Percentage of Cityscapes Training Dataset

Table 5.5 demonstrates fine-tuning results of DeepLab that was pre-trained on our ProcSy dataset. The fine-tuning experiments are run with just around 300 randomly-sampled images from Cityscapes training set. We first capture results of training for 10,000 iterations. We then proceed with training until 40,000 iterations and capture the results again.

We again observe a significant improvement in both mIoU and fwIoU scores over the first 10,000 iterations of fine-tuning (averaging around 229% increase in mIoU and 217% increase in fwIoU). This is impressive because we are using just 10% of the Cityscapes training set.

What is more noteworthy is the phenomenon that is seen through the fwIoU scores. The progression of fwIoU_bold and fwIoU_CS signify that as fine-tuning proceeds with a small subset of Cityscapes training set, the network weights are improving gradually towards Cityscapes target domain.

5.1.4 Summary

It is not necessary to use the full extent of a target dataset domain for good domain adaptation. While this gives the highest possible results at a set number of fine-tune iterations, the performance gains seen are sub-optimal. In the real world, data can be gathered boundlessly. It is not possible to train semantic segmentation networks without an upper bound on the training dataset.

From experiments in this section, we understand that unimodal semantic segmentation network architectures have a diminishing return of performance with respect to increasing the amount of data used for fine-tuning. With this intuition, we can identify an optimal amount of data to be acquired from a target domain for fine-tuning a DNN.

In order to do this, we first fix the number of training iterations as in section 5.1.3 experiment. Using a surrogate target real-world dataset such as Cityscapes, we identify the point of diminishing return. Using this amount of the surrogate dataset, we proceed with the experiment in section 5.1.3 to find the point of diminishing return for an ideal number of fine-tuning iterations for the respective DNN. This is an efficient way to alleviate the domain adaptation problem from a generated synthetic dataset towards a target domain.

IoU (%) of DeepLabv3+ tested on Cityscapes validation set							
		ProcSy w/ CityEngine vehicles			ProcSy w/ GTA5 vehicles		
classes	CS	no FT	10k FT	40k FT	no FT	10k FT	40k FT
road	97.80	3.90	92.30	92.90	1.30	91.20	91.80
building	91.60	41.30	78.00	81.20	39.80	78.30	80.80
vegetation	91.80	34.30	84.40	86.00	38.00	84.40	86.00
car	93.60	35.70	73.00	77.70	6.40	70.80	76.00
sidewalk	82.60	7.80	55.20	58.10	1.60	51.60	55.50
sky	94.20	43.00	75.10	82.10	50.30	79.40	83.10
pole	59.40	7.40	32.80	37.40	5.00	35.00	38.50
person	77.20	26.20	46.60	49.90	7.60	48.10	50.10
terrain	58.90	3.30	29.30	34.50	6.60	28.00	31.40
fence	56.00	0.00	0.00	12.70	0.00	0.40	14.60
wall	50.30	4.00	14.10	17.00	1.80	16.60	14.30
traffic sign	72.50	0.00	0.00	6.20	0.00	0.40	17.50
bicycle	73.20	0.00	8.40	28.60	0.00	5.10	27.40
truck	60.30	1.50	1.20	10.70	0.40	1.90	9.70
bus	77.20	0.70	11.10	9.00	1.00	0.10	1.50
train	60.00	0.00	0.00	0.20	0.00	0.00	6.50
traffic light	62.20	0.00	0.00	4.90	0.00	0.00	2.80
rider	54.10	0.00	1.40	0.10	0.00	0.00	0.10
motorcycle	48.90	0.00	0.00	0.00	0.00	0.00	0.00
mIoU	**71.70**	**11.00**	**31.70**	**36.30**	**8.40**	**31.10**	**36.20**
fwIoU	**91.55**	**22.06**	**70.48**	**75.90**	**22.56**	**70.89**	**75.47**
fwIoU (CS)			78.23	80.66		77.70	79.95
fwIoU (ProcSy)			68.93	72.09		69.53	71.88

Table 5.5: Results of DeepLab v3+ fine-tuning iteration experiments on ProcSy datasets; fine-tune iterations are done with 10% of Cityscapes (CS) training set

5.2 Comparing URSA and ProcSy

Chapters 3 and 4 explore two very different methods of generating ground truth annotation data for semantic segmentation research. In this section, we explore the merits of each method towards our research goals. We gain an insight regarding the mesh geometry complexity with respect to perceived realism and domain transfer capabilities.

5.2.1 Strengths and Weaknesses

Towards comparing the merits of the two dataset generation methods, we establish a set of metrics that are relevant to the task of semantic segmentation research. These are ordered in terms of importance towards our ultimate goal of a dataset generator that can flexibly produce annotation data for varying influence factors. Table 5.6 shows this comparison. Justifications are given for the winning choice for each metric. Where applicable, references are made to appropriate sections of this thesis.

A quick glance at table 5.6 shows that there is an overall draw between the two dataset generation methods. There are meritable benefits and concerning drawbacks to each method. For our goals, ProcSy is a clear fit. This is due to the fact that ProcSy method has perfect ground truth annotations, and lets us create quantifiable influence factor variations with relative ease. Also, it is observed to be much cheaper than URSA. This can be hugely beneficial for research teams.

Perceived realism is a metric where URSA has a significant leverage over ProcSy. Rockstar Games spent multiple years of development time to faithfully create a replica of real-world Los Angeles in the game world (section 3.1). As useful as procedural modeling is to quickly generate a map region for rapid prototyping, the scene realism and nuances found in GTA5 is not achievable. If such was the case, then developers at Rockstar Games surely would have used the method to make their lives easier.

ProcSy shines with respect to URSA when it comes to adapting to a new ODD. GTA5 game world, Los Santos, is populated with Southern California-like architecture, flora, and fauna. Reverse-engineering the game world and its myriad assets in order to inject a completely different ODD (let's say a Brazilian favela) requires an insurmountable amount of work that is out-of-scope of a master's research project. As mentioned, procedural generation via CityEngine can be leveraged for this purpose. Swapping out world resources in the UE4 environment is trivial to do as it is one of the primary functionalities of a game engine editor.

metric	URSA	ProcSy	justification
cost		X	AMT labeling cost of GTA5 data was $2448.15 USD. ProcSy is virtually cost-less (UE4 is free-to-use; CityEngine has a free 30-day trial period).
ground truth annotations		X	ProcSy has perfect labels (4.2.3). GTA5 annotation methods provide flawed and incomplete results (3.2 and 3.3).
influence factors		X	As per sections 3.3.4 and 4.2.7, ProcSy is the clear winner here.
perceived realism	X		GTA5 looks much more life-like.
mesh geometry fidelity	X		GTA5 mesh models have an order of magnitude higher fidelity than ProcSy (4.1.2).
choice of ODD		X	It is not trivial to replace the game world in GTA5. ProcSy is fully versatile in this regard.
depth	X	X	Both methods grant access to depth buffers.
occlusion maps		X	ProcSy allows for producing occlusion map data, GTA5 does not.
terrain elevation	X		GTA5 is a mockery of real-world LA including hills, valleys, and mountains. ProcSy base map is flat (4.1.1).
nighttime scenery	X		GTA5 nighttime scenes have emissive windows, car lights, streetlamps, etc. ProcSy only has Sun as its light source.
temporal data	X		ProcSy world is strictly static. GTA5 is a full-fledged game with a dynamic world.

Table 5.6: Comparison of metrics between URSA and ProcSy dataset generation methods

5.2.2 Using High Fidelity Vehicle Geometry

An interesting metric in table 5.6 is mesh geometry fidelity. Prior intuition suggests that having access to higher quality mesh models can be a key way to boost a dataset towards achieving realism. In section 4.1.2, we mention that we are able to extract around 300

vehicle models from the GTA5 single-player game. We observe that these mesh models are an order of magnitude greater in polygon-count than CityEngine-provided low-poly models.

The intuition led us to design an experiment to observe the effects of substituting higher fidelity mesh geometry for dataset generation. We generate a secondary ProcSy dataset with GTA5 vehicles (ProcSy-GTA5) in-place of CityEngine ones. Figure 5.1 shows the respective dataset distribution. Of note is that the distribution is almost identical to the original ProcSy distribution. One discernible point is that there is a significantly higher amount of trucks in ProcSy-GTA5.

We then proceed to train DeepLab v3+ in the same way that the original model was trained for section 5.1.3. The results are quite surprising. Not only did the GTA5 vehicles fail to provide better generalization to the Cityscapes domain, the network performance actually degraded from the original with CityEngine vehicles. More specifically, the car class performance is seen to drop from 35.7% accuracy to just 6.4% (see table 5.5).

Further investigation can provide more insight as to the reason why better quality mesh models do not account for better domain shift performance. For the scope of this thesis, this insight validates the use of CityEngine-provided low-poly assets towards dataset generation and influence factors experiments.

5.2.3 Summary

We pick the ProcSy dataset generation method towards our goal of analyzing influence factor effects on unimodal semantic segmentation networks. We identify that higher mesh geometry fidelity is not necessarily a boon for domain adaptation to the real-world. This insight allows us to proceed designing our method and experiments with out-of-the-box assets provided by CityEngine.

By comparing URSA and ProcSy, we understand that the two approaches have unique benefits and drawbacks. The ease of controllability of influence factors is ultimately the deciding factor towards moving forward with ProcSy.

5.3 Influence Factor Analysis

Here we present three experiments to demonstrate the usage of our ProcSy synthetic dataset generator for understanding effects of different influence factors on a unimodal

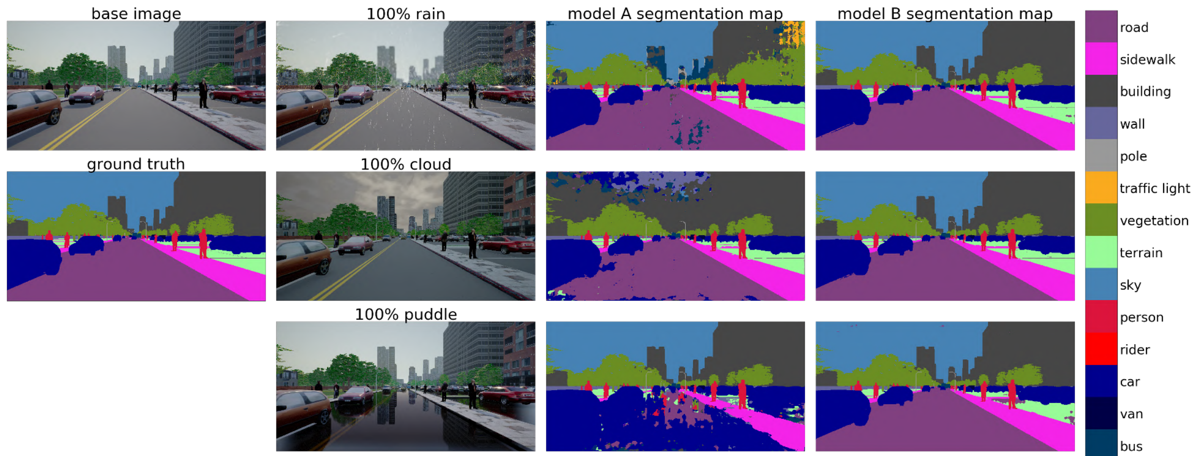


Figure 5.2: Performance of model A and B with different factors (each row); here, due to space constraints, we only show samples at 100% level for each influence factor

semantic segmentation model's performance. For each of the three experiments we give the experimental objective, details, results, and actions suggested by the results. We use DeepLab v3+ as the target DNN.

5.3.1 Effects of Environmental Influence Factors

In this experiment, we train and compare the performance of two DeepLab v3+ models. We train model A with 8000 clean images, and model B is trained with 8000 images with equal proportion of three influence factors (rain, cloud, and puddles). More precisely, model B's training images are split into three equal parts, one per factor. Then each part is split in five equal sub-parts, one for a different level of the given factor to be applied: 0%, 25%, 50%, 75%, and 100%. Each model is trained with 140,000 iterations with a batch-size of 16 and crop-size of 512×512. The performance of each model is evaluated by testing with different influence factors. The purpose of this experiment is two-fold. First, we would like to observe how the influence factors degrade the performance of model A. Second, it is important to know if model B is able to generalize across different influence factors and is more robust than model A. Otherwise, we would need to explore more advanced training methods to improve robustness [88].

In figure 5.3, we show mIoU of two models under different testing conditions, where all test images have a certain level of an influence factor. For model A, as we increase the

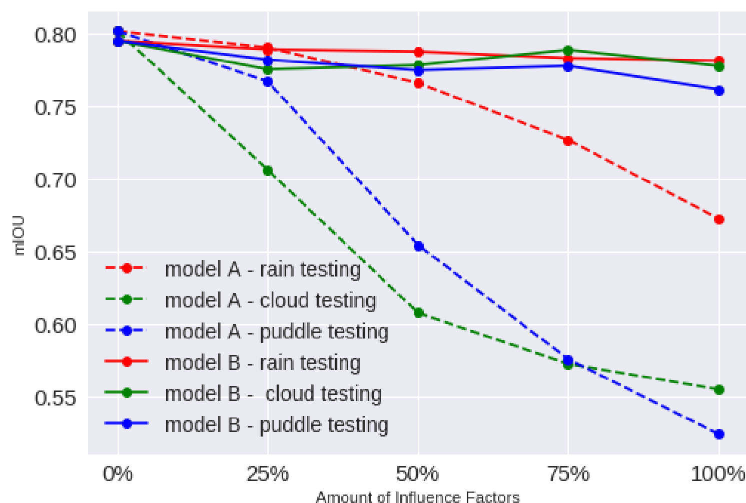


Figure 5.3: mIoU for each testing scenarios for two models, A and B; x-axis denotes the intensity level of a given influence factor in each scenario

level of each influence factor, its performance worsens. Initially, we found it surprising that cloud and puddle factors decrease the performance more than the rain factor does. The reason for this is largely due to the model inaccurately predicting sky as another class, such as car. This kind of error can be seen in figure 5.4, where we plot the IoU of several classes for each model. For example, in figure 5.4, the IoU values of car and sky classes reduce significantly with increased clouds compared to the other classes. Similarly, we can also see that puddles have a strong negative effect on person, car, and road classes, because the puddle factor creates reflection of objects on the ground.

In contrast to model A, model B has generally stable performance across all influence factors and at different intensity levels. This suggests that the model can generalize from the exposure to these factors in training. Figure 5.2 shows samples of each model's prediction under different situations. We note that there is a small gap in overall and class-wise performance when testing on clean images. However, this can be explained by the fact that model B was not trained with as many clean images as model A. Our results show that model B is more robust than model A.

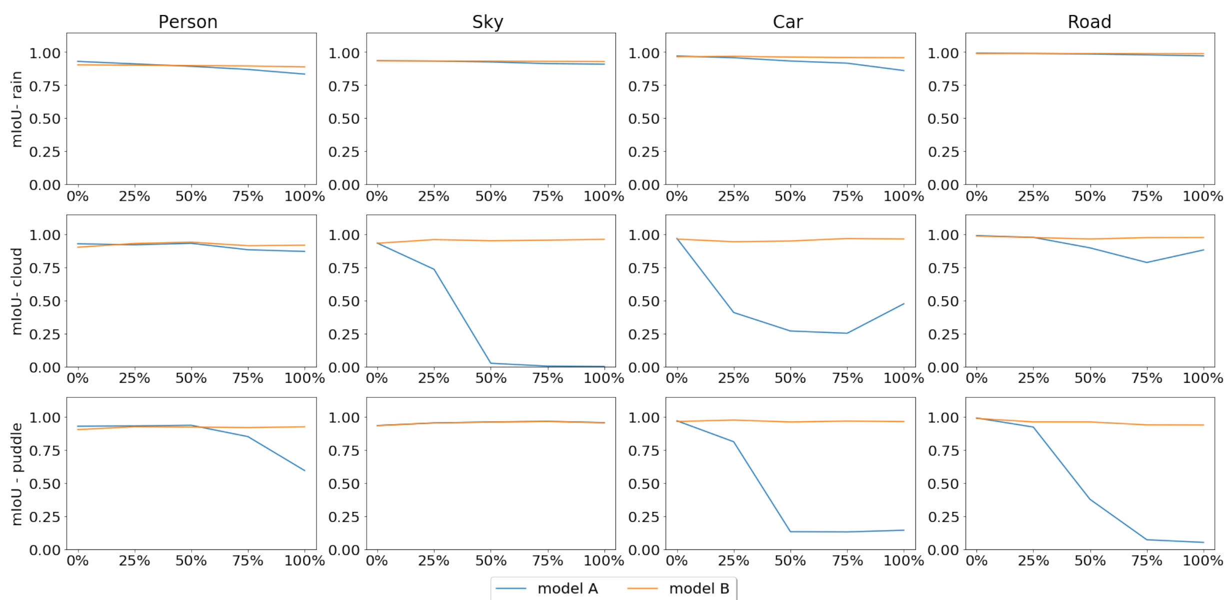


Figure 5.4: IoU values for 4 classes: person, sky, car and road; each row corresponds to each testing scenario (rain, cloud and puddle) and each column corresponds to each class

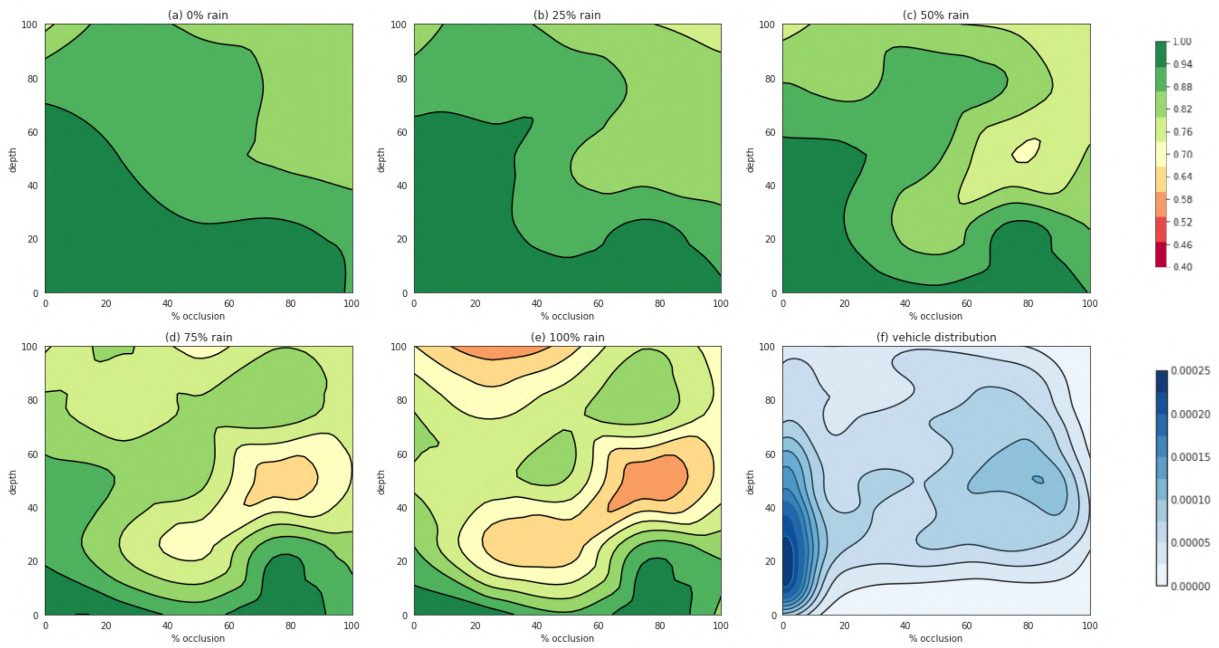


Figure 5.5: ‘a-e’ show model’s accuracy on vehicles according to occlusion level and depth; darker green color corresponds to higher accuracy; ‘f’ shows frequency of vehicles; scales for these plots are shown in color bars at the right. The color bar for the distribution plot represents the distribution density.

5.3.2 Effects of Depth and Occlusion

The ProcSy dataset allows for the assessment of a network's prediction quality with respect to depth and occlusion. Performing this analysis on an existing real-world, autonomous driving dataset such as Cityscapes [11] is difficult since occlusion information is not available. However, this information generated from a synthetic dataset can help us understand more about the reliability and weaknesses of a unimodal semantic segmentation model.

In this experiment, we randomly choose 270 ProcSy training dataset images with a total of 1200 vehicles in the test set. We test these images on model A from the experiment in section 5.3.1. Similar to Synscapes [85], we divide predicted pixels into subsets of $[0\%, 20\%], [20\%, 40\%], [40\%, 60\%], [60\%, 80\%], [80\%, 100\%]$ according to depth and amount of occlusion of each vehicle. Then, we calculate accuracy for each subset, after which we use cubic spline interpolation to get a contour plot. We repeat the same process for different levels of rain in the images, as shown in figure 5.5a-e. Also, we plot the distribution of vehicles in the training set according to occlusion and depth (figure 5.5f).

Occlusion Map Artifacts

The vehicle distribution shown in figure 5.5f indicates a source of error in the ProcSy dataset occlusion maps. Ideally, the distribution should represent a gradient that diminishes as depth and/or amount of occlusion increases. The beginnings of this pattern is observed within the zone boxed by $[0\%, 20\%]$ occlusion and $[0\%, 60\%]$ depth. However, there is a characteristic hotspot around 80% occlusion and 50% depth.

Our method of occlusion map generation, as outlined in 4.2.5, involves a hard-stop at a distance of 10,000 game world units. This distance is used as the 100% depth amount in figure 5.5. At this distance, vehicles appear very small in the image frame. These vehicles also tend to have only a few visible pixels, if any. Without the hard-stop implemented, the line of horizon in occlusion maps would be littered with many such fully or almost-fully occluded vehicles.

While fully occluded (or almost fully-occluded) vehicles are removed from the horizon, they are not removed from the dataset at medium depth values. Figure 5.6 shows an example image frame where 3 of the vehicle instances are either fully or almost-fully occluded by surroundings. Existence of fully or highly occluded vehicles around medium depth range is the root cause for the characteristic hot-spot around 80% occlusion and 50% depth. This is an indication that a better approach is needed to filter out fully occluded vehicles than simply relying on a far clipping plane.

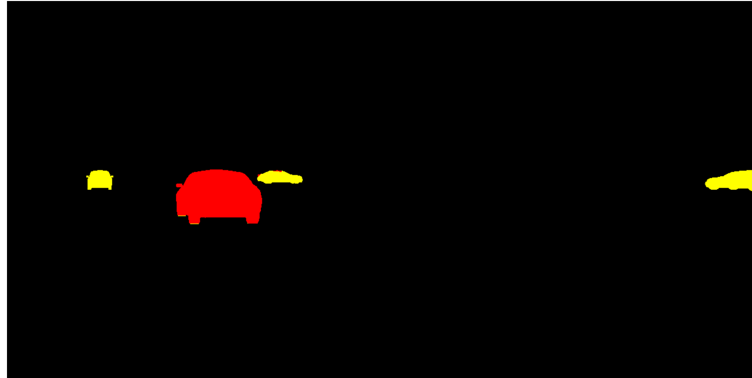


Figure 5.6: Occlusion map of a vehicle showing 3 instances where the vehicle is fully or highly occluded; yellow indicates occluded pixels; red indicates visible pixels

Insights on Model

In the case of no rain (figure 5.5a), we see that DeepLab performs really well on vehicles across a wide band of depth and occlusion. This directly reflects DeepLab's use of atrous spatial pyramid pooling (section 2.1.3) in its architecture. ASPP allows DeepLab to have good scale-invariance in predicting object classes. Generally speaking, the model's accuracy decreases as both depth and occlusion increases.

We also observe that in the region bounded by [0%, 20%] occlusion and [0%, 60%] depth, the model's accuracy is quite stable up until 50% rain amount (figure 5.5c). This region corresponds to the left-most cluster in the distribution map (figure 5.5f), which is the expected behaviour of the dataset distribution.

Increasing from 50% rain intensity, we see a phenomenon developing around the region centered near 40% occlusion and 20% depth. This growing hot-spot characterizes model A's weakness in segmenting vehicles in images that are filled with rain noise. This complements the observation in figure 5.3, which demonstrates a significant degradation of model A's overall mIoU performance to increased rain intensity.

5.3.3 Optimizing for Data Collection

Collecting and labeling more data in different weather conditions is the simplest way to improve a model's robustness. However, this is a costly and laborious task, and one may

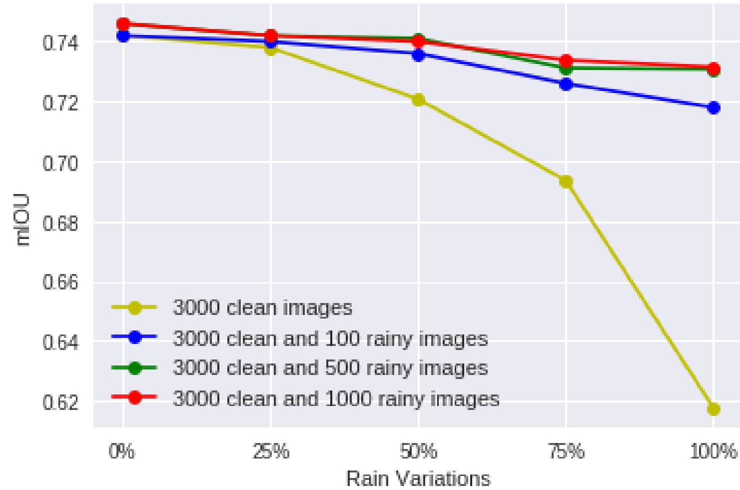


Figure 5.7: mIoU values for each model of section 5.3.3

want to know the optimal amount of data to collect. We build on our intuition from experiments in section 5.1.3. With this experiment, we demonstrate that the ProcSy dataset can be used towards estimating an optimal amount of data to be acquired for making a model more robust to unimodal semantic segmentation of adverse conditions.

We train and compare the performance of four different models — one with 3000 clean images, and the other three with the same 3000 clean images plus an additional 100, 500, or 1000 rainy images (each additional set containing an equal amount of images at a given level: 0%, 25%, 50%, 75%, 100%) respectively. To ensure that rain is the only differentiating factor in performance of these models, we constrain rainy image generation by random sampling from just the original 3000 clean images.

The results are shown in figure 5.7. We see that most of the model's improvement is obtained by adding just 100 rainy images. For instance, when the amount of rain is 100%, it improves the performance by 10% (from 61.8% to 71.8% mIoU), whereas adding another 900 rainy images only improves performance by an additional 2%. Also, adding 1000 rainy images only gives us a slight increase in mIoU compared to adding only 500 images. Since raindrops cause occlusion effects in the images (a kind of irreducible source of error), this result suggests that adding more rainy images will likely not further improve the model's performance.

From this experiment, we posit that given the metric-based performance requirement

for a semantic segmentation task, a threshold can be identified (such as 500 images is good enough, 1000 images is overkill) through rapid training on influence factor variations of a synthetic dataset. By collecting, annotating, and training real-world data indicated by the threshold amount, an unimodal semantic segmentation network architecture can be optimized towards the respective influence factor. Assuming other influence factors to be orthogonal, experiments run on those can yield similar threshold values. One can use these thresholds as reasonable estimates for amount of real-world data collection towards improving the robustness of a given model.

5.3.4 Summary

In this section, we experiment on DeepLab v3+ using influence factor variations from our ProcSy dataset generation method — namely rain, cloud, and puddles. We see that increased variations in cloud and puddles affect DeepLab's performance much more significantly than rain. We identify that a root cause is due to the model's propensity to mis-classify sky with increase in cloud coverage. For classes that tend to be spatially close to the ground, increased reflectivity of road surface due to puddles becomes a negative factor for the model's performance.

We produce a dataset distribution heatmap for vehicle class relative to depth and occlusion from image space. For a DeepLab model trained on clean images, we produce similar heatmaps for model accuracy on rain intensity variations. From these heatmaps, we gain insight towards the way in which the model fails to classify vehicles as amount of rain is increased. This experiment also highlights an issue with our occlusion map generation technique. There is an identifiable need to have a robust way of removing fully occluded objects from the occlusion map data.

Finally, we expand on our intuition gained through fine-tuning models in section 5.1.3. We see that for DeepLab, introducing as little as 3% of rainy images in the training set led to significant gains (around 10%) in the network's robustness towards such imagery. In contrast, adding more than 15% of rainy images had diminishing returns on network performance. We posit that training data for a unimodal semantic segmentation network can be acquired optimally from the real-world by first gaining such insight from synthetic influence factor variations.

Chapter 6

Conclusion

This thesis hones into a synthetic dataset generation method that is conducive to rapid validation of unimodal semantic segmentation network architectures. The method is capable of scene repeatability towards generating influence factor variations in the form of rain, cloud, puddles, and Sun positions. The method also allows for gaining further insight on DNN behaviour by using depth and occlusion maps.

Existing research in semantic segmentation has progressed towards multimodal techniques due to a lack of quality and quantity of ground truth data for adverse weather and lighting conditions. To our knowledge, this is the only such research work in this field that proposes to use synthetic influence factor variations as an empirical measure towards optimizing the training and robustness of unimodal semantic segmentation networks.

We justify ProcSy dataset's quality by benchmarking performance against pre-existing synthetic datasets, including our own GTA5-based URSA dataset. We identify that higher quality mesh geometry in such synthetic datasets does not necessarily have a direct correlation towards performance against a real-world operational design domain.

We demonstrate the ProcSy method's usefulness by performing experiments on DeepLab v3+, a state-of-the-art network for unimodal semantic segmentation tasks. We gain insights about the network's behaviour on unseen adverse weather conditions. Based on empirical testing, we identify a point of diminishing return towards optimizing the network's training with additional adverse condition data samples.

Future work extending from this thesis should explore the effects of combining influence factors. In the scope of this thesis, influence factors such as rain, cloud, and puddles were treated orthogonally. This was useful in identifying methodologies and validation

procedures. However, in the real world, these factors work in some combination of each other.

Further work is needed to generate occlusion maps that are more robust. Our experiments show clear value in studying occlusion data. However, the current approach is cumbersome and does not produce true instance-level occlusion data. Also, fully occluded objects remain as undesirable artifacts in current occlusion maps.

Significant next steps of the dataset generation process would be to introduce elevation data in the procedurally-created world. Also, a time-consuming but worthwhile endeavour would be to transition from a static 3D environment to a fully-dynamic world with camera/traffic movement along roadways and pedestrian animations. This opens up possibilities for temporal data collection with nuances such as motion blur and greater scene variation.

A future work to further validate these influence factor studies would be to first annotate real-world data with and without influence factor variations. By training networks on real-world data, it can be observed whether the networks behave similarly to ones trained on synthetic data. The Autonomoose project is poised for data collection towards this goal.

References

[1] Matt Angus, Mohamed ElBalkini, Samin Khan, Ali Harakeh, Oles Andrienko, Cody Reading, Steven Waslander, and Krzysztof Czarnecki. Unlimited road-scene synthetic annotation (ursa) dataset. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 985–992. IEEE, 2018.

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

[3] Gabriel J Brostow, Julien Fauqueur, and Roberto Cipolla. Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, 30(2):88–97, 2009.

[4] Arthur E Bryson, Walter F Denham, and Stewart E Dreyfus. Optimal programming problems with inequality constraints. *AIAA journal*, 1(11):2544–2550, 1963.

[5] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. *arXiv preprint arXiv:1412.7062*, 2014.

[6] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[7] Yuhua Chen, Wen Li, and Luc Van Gool. Road: Reality oriented adaptation for semantic segmentation of urban scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7892–7901, 2018.

[8] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[9] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[10] Robert L Cook and Kenneth E Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)*, 1(1):7–24, 1982.

[11] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[12] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[13] Adrian Courrges. Gta v - graphics study, Nov 2015. http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/.

[14] David Daniel Cox and Thomas Dean. Neural networks and neuroscience-inspired computer vision. *Current Biology*, 24(18):R921–R929, 2014.

[15] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[16] Esri. World imagery, Dec 2009. https://arcg.is/1vSv81.

[17] Esri. Cityengine 2013 release notes, 2013. https://doc.arcgis.com/en/cityengine/latest/whats-new/what-s-new-in-cityengine-2013-.htm.

[18] Esri. About cityengine, 2019. https://doc.arcgis.com/en/cityengine/latest/get-started/get-started-about-cityengine.htm.

[19] GTA Fandom. Galileo observatory, Sep 2013. https://gta.fandom.com/wiki/Galileo_Observatory.

[20] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[21] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual worlds as proxy for multi-object tracking analysis. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4340–4349, 2016.

[22] Rockstar Games. Rockstar Editor. https://tinyurl.com/y8sw83aa.

[23] Rockstar Games. Introducing the rockstar editor, Apr 2015. https://www.rockstargames.com/newswire/article/52416/introducing-the-rockstar-editor.

[24] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[25] Ron Goldman, Scott Schaefer, and Tao Ju. Turtle geometry in computer graphics and computer-aided design. *Computer-Aided Design*, 36(14):1471–1482, 2004.

[26] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[27] Kotaro Hara, Abigail Adams, Kristy Milland, Saiph Savage, Chris Callison-Burch, and Jeffrey P Bigham. A data-driven analysis of workers' earnings on amazon mechanical turk. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 449. ACM, 2018.

[28] Daniel Hernandez-Juarez, Lukas Schneider, Antonio Espinosa, David Vázquez, Antonio M López, Uwe Franke, Marc Pollefeys, and Juan C Moure. Slanted stixels: Representing san francisco's steepest streets. *arXiv preprint arXiv:1707.05397*, 2017.

[29] Elizabeth Hill and Leanne Trimble. Scholars geoportal: A new platform for geospatial data discovery, exploration and access in ontario universities. *IASSIST Quarterly*, 36(1), 2012.

[30] Naty Hoffman. Introduction to physically based shading in theory and practice, Jul 2016. https://www.youtube.com/watch?v=j-A0mwsJRmk.

[31] David S Immel, Michael F Cohen, and Donald P Greenberg. A radiosity method for non-diffuse environments. In *Acm Siggraph Computer Graphics*, volume 20, pages 133–142. ACM, 1986.

[32] AG Ivakhnenko and Valentin Grigor'evich Lapa. Cybernetic predicting devices. Technical report, Purdue University School of Electrical Engineering, 1966.

[33] James T Kajiya. The rendering equation. In *ACM SIGGRAPH computer graphics*, volume 20, pages 143–150. ACM, 1986.

[34] Brian Karis and Epic Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 4, 2013.

[35] Samin Khan, Buu Phan, Rick Salay, and Krzysztof Czarnecki. Procsy: Procedural synthetic dataset generation towards influence factor studies of semantic segmentation networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[36] Dong-Ki Kim, Daniel Maturana, Masashi Uenoyama, and Sebastian Scherer. Season-invariant semantic segmentation with a deep multimodal network. In *Field and Service Robotics*, pages 255–270. Springer, 2018.

[37] Alexander Kirillov, Kaiming He, Ross Girshick, Carsten Rother, and Piotr Dollár. Panoptic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9404–9413, 2019.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven L Waslander. Joint 3d proposal generation and object detection from view aggregation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–8. IEEE, 2018.

[40] Eugene Lapidous and Guofang Jiao. Optimal depth buffer for low-cost graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '99, pages 67–73, New York, NY, USA, 1999. ACM. http://doi.acm.org/10.1145/311534.311579.

[41] Fahad Lateef and Yassine Ruichek. Survey on semantic segmentation using deep learning techniques. *Neurocomputing*, 338:321–348, 2019.

[42] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[43] Peiliang Li, Xiaozhi Chen, and Shaojie Shen. Stereo r-cnn based 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7644–7652, 2019.

[44] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[45] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[46] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[47] Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 1969.

[48] Jibitesh Mishra and Sarojananda Mishra. *L-system Fractals*, volume 209. Elsevier, 2007.

[49] Jaap van Muijden. Gpu-based procedural placement in horizon zero dawn, Mar 2017. https://www.guerrilla-games.com/read/gpu-based-procedural-placement-in-horizon-zero-dawn.

[50] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Acm Transactions On Graphics (Tog)*, volume 25, pages 614–623. ACM, 2006.

[51] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[52] Thomas Nagel. What is it like to be a bat? *The philosophical review*, 83(4):435–450, 1974.

[53] Naver. *Proxy Virtual Worlds*. http://www.europe.naverlabs.com/Research/Computer-Vision/Proxy-Virtual-Worlds.

[54] Gerhard Neuhold, Tobias Ollmann, Samuel Rota Bulo, and Peter Kontschieder. The mapillary vistas dataset for semantic understanding of street scenes. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4990–4999, 2017.

[55] Tianyun Ni. Direct compute: Bring gpu computing to the mainstream. In *GPU Technology Conference*, page 23, 2009.

[56] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[57] Andreas Pfeuffer and Klaus Dietmayer. Robust semantic segmentation in adverse weather conditions by means of sensor data fusion. *arXiv preprint arXiv:1905.10117*, 2019.

[58] Nathan Reed. Depth precision visualized, Jul 2015. https://developer.nvidia.com/content/depth-precision-visualized.

[59] Stephan R Richter, Zeeshan Hayder, and Vladlen Koltun. Playing for benchmarks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2213–2222, 2017.

[60] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

[61] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3234–3243, 2016.

[62] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[63] Aitor Ruano. DeepGTAV. https://github.com/aitorzip/DeepGTAV.

[64] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[65] Christos Sakaridis, Dengxin Dai, and Luc Van Gool. Semantic nighttime image segmentation with synthetic stylized data, gradual adaptation and uncertainty-aware evaluation. *arXiv preprint arXiv:1901.05946*, 2019.

[66] Andrew Sanders. *An Introduction to Unreal Engine 4*. AK Peters/CRC Press, 2016.

[67] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[68] E. Shelhamer, J. Long, and T. Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, April 2017.

[69] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[70] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. In *Computer Graphics Forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[71] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In *Proceedings of IFIP Congress 71*, 1971.

[72] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[73] Dean Takahashi. Nvidia ceo bets big on deep learning and vr, Apr 2016. https://venturebeat.com/2016/04/05/nvidia-ceo-bets-big-on-deep-learning-and-vr/.

[74] Autonomoose Team. autonomoose, 2016. https://www.autonomoose.net/.

[75] Martin Thoma. A survey of semantic segmentation. *arXiv preprint arXiv:1602.06541*, 2016.

[76] Vincent Torre and Tomaso Poggio. On edge detection. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1984.

[77] Frederick Tung, Jianhui Chen, Lili Meng, and James J Little. The raincouver scene parsing benchmark for self-driving in adverse weather and at night. *IEEE Robotics and Automation Letters*, 2(4):2188–2193, 2017.

[78] Raquel Urtasun. Describing the scene as a whole: Joint object detection, scene classification and semantic segmentation. In *Proceedings of the 2012 IEEE Conference on*

Computer Vision and Pattern Recognition (CVPR), pages 702–709. IEEE Computer Society, 2012.

[79] Abhinav Valada, Johan Vertens, Ankit Dhall, and Wolfram Burgard. Adapnet: Adaptive semantic segmentation in adverse environmental conditions. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4644–4651. IEEE, 2017.

[80] Joey de Vries. Pbr theory, 2014. https://learnopengl.com/PBR/Theory.

[81] City Of Waterloo. City of waterloo open data, 2012. http://data.waterloo.ca/.

[82] Michael Wegener. Operational urban models state of the art. *Journal of the American planning Association*, 60(1):17–29, 1994.

[83] GTA Network Wiki. Weather, Jan 2017. https://wiki.gtanet.work/index.php?title=Weather.

[84] Wikipedia. L-system, Jul 2019. https://en.wikipedia.org/wiki/L-system.

[85] Magnus Wrenninge and Jonas Unger. Synscapes: A photorealistic synthetic dataset for street scene parsing. *arXiv preprint arXiv:1810.08705*, 2018.

[86] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. Bdd100k: A diverse driving video database with scalable annotation tooling. *arXiv preprint arXiv:1805.04687*, 2018.

[87] Yang Zhang, Philip David, and Boqing Gong. Curriculum domain adaptation for semantic segmentation of urban scenes. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2020–2030, 2017.

[88] S. Zheng, Y. Song, T. Leung, and I. Goodfellow. Improving the robustness of deep neural networks via stability training. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4480–4488, 2016.