Argoverse Stereo Competition

To support our Stereo Competition on [EvalAI (https://eval.ai/web/challenges/challenge-page/1704/overview)](https://eval.ai/web/challenges/challenge-page/1704/overview), we have made available ground-truth depth for [Argoverse v1.1 (https://www.argoverse.org/av1.html#stereo-link)](https://www.argoverse.org/av1.html#stereo-link), derived from lidar point cloud accumulation. We used our [scene flow method (https://arxiv.org/pdf/2011.00320.pdf)](https://arxiv.org/pdf/2011.00320.pdf) to accumulate lidar points from 11 frames and adopted evaluation metrics from the great [KITTI stereo challenge (http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo)](http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo). However, in comparison to KITTI, our stereo images have 10 times the resolution and 16 times as many training frames — making it a much larger and more challenging dataset.

Argoverse Stereo consists of rectified stereo images and ground truth disparity maps for 74 out of the 113 Argoverse 3D Tracking Sequences. The stereo images are (2056 x 2464 pixels) and sampled at 5 Hz. The dataset contains a total of 6,624 stereo pairs with ground truth depth, although we withhold the ground truth depth for the 15 sequence test set.

So here is a notebook to get you started with our stereo dataset and stereo competition. Have fun!

NOTE: Please carefully check the section `Updates for the Stereo Competition 2022` for specific details about out latest competition.

Data setup

The Argoverse Stereo dataset can be download from [here (https://www.argoverse.org/av1.html#stereo-link)](https://www.argoverse.org/av1.html#stereo-link). There are 2 packages from the **Argoverse Stereo v1.1** section you will need to download to get started with this tutorial:

- Rectified stereo images (train / val / test)
- Disparity maps (train / val)

This tutorial assumes that you have already downloaded and extracted all necessary data into a specific folder and that you have the [Argoverse API (https://github.com/argoai/argoverse-api)](https://github.com/argoai/argoverse-api) up and running. For example, this is the directory structure you should have:

```
argoverse_stereo_v1.1
└───disparity_maps_v1.1
|   └───test
|   └───train
|   |    └───273c1883-673a-36bf-b124-88311b1a80be
|   |        └───stereo_front_left_rect_disparity
|   |        └───stereo_front_left_rect_objects_disparity
|   └───val
└───rectified_stereo_images_v1.1
    └───test
    └───train
    |    └───273c1883-673a-36bf-b124-88311b1a80be
    |        └───stereo_front_left_rect
    |        └───stereo_front_right_rect
    |            vehicle_calibration_stereo_info.json
    └───val
```

Installing the dependencies

You will need to install three dependencies to run this tutorial:

- **Open3D**: See instructions on how to install [here (https://github.com/intel-isl/Open3D)](https://github.com/intel-isl/Open3D).
- **OpenCV contrib**: See instructions on how to install [here (https://pypi.org/project/opencv-contrib-python)](https://pypi.org/project/opencv-contrib-python).
- **Plotly**: See instructions on how to install [here (https://github.com/plotly/plotly.py)](https://github.com/plotly/plotly.py).

Once you get ready with the dataset and the dependencies you can run the cells below. Please make sure to change the path to the dataset accordingly.

In [1]:
```
%matplotlib notebook

import copy
import json
import shutil
from pathlib import Path

import cv2
import matplotlib.pyplot as plt
import numpy as np
import open3d as o3d
import plotly.graph_objects as go

from argoverse.data_loading.stereo_dataloader import ArgoverseStereoDataLoader
from argoverse.evaluation.stereo.eval import StereoEvaluator
from argoverse.utils.calibration import get_calibration_config
from argoverse.utils.camera_stats import RECTIFIED_STEREO_CAMERA_LIST

STEREO_FRONT_LEFT_RECT = RECTIFIED_STEREO_CAMERA_LIST[0]
STEREO_FRONT_RIGHT_RECT = RECTIFIED_STEREO_CAMERA_LIST[1]


# Path to the dataset (please change accordingly).
data_dir = "/data/datasets/stereo/argoverse1/"

# Choosing the data split: train, val, or test (note that we do not provide ground truth for the test set).
split_name = "train"

# Choosing a specific log id. For example, 273c1883-673a-36bf-b124-88311b1a80be.
log_id = "273c1883-673a-36bf-b124-88311b1a80be"

# Choosing an index to select a specific stereo image pair. You can always modify this to loop over all data.
idx = 34

# Creating the Argoverse Stereo data loader.
stereo_data_loader = ArgoverseStereoDataLoader(data_dir, split_name)

# Loading the left rectified stereo image paths for the chosen log.
left_stereo_img_fpaths = stereo_data_loader.get_ordered_log_stereo_image_fpaths(
    log_id=log_id,
    camera_name=STEREO_FRONT_LEFT_RECT,
)

# Loading the right rectified stereo image paths for the chosen log.
right_stereo_img_fpaths = stereo_data_loader.get_ordered_log_stereo_image_fpaths(
    log_id=log_id,
    camera_name=STEREO_FRONT_RIGHT_RECT,
)

# Loading the disparity map paths for the chosen log.
disparity_map_fpaths = stereo_data_loader.get_ordered_log_disparity_map_fpaths(
    log_id=log_id,
    disparity_name="stereo_front_left_rect_disparity",
)

# Loading the disparity map paths for foreground objects for the chosen log.
disparity_obj_map_fpaths = stereo_data_loader.get_ordered_log_disparity_map_fpaths(
    log_id=log_id,
    disparity_name="stereo_front_left_rect_objects_disparity",
)
```

```
Jupyter environment detected. Enabling Open3D WebVisualizer.
[Open3D INFO] WebRTC GUI backend enabled.
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.
```

Stereo images and ground-truth disparity loading

We provide rectified stereo image pairs, disparity maps for the left stereo images, and also disparity maps for foreground objects only.

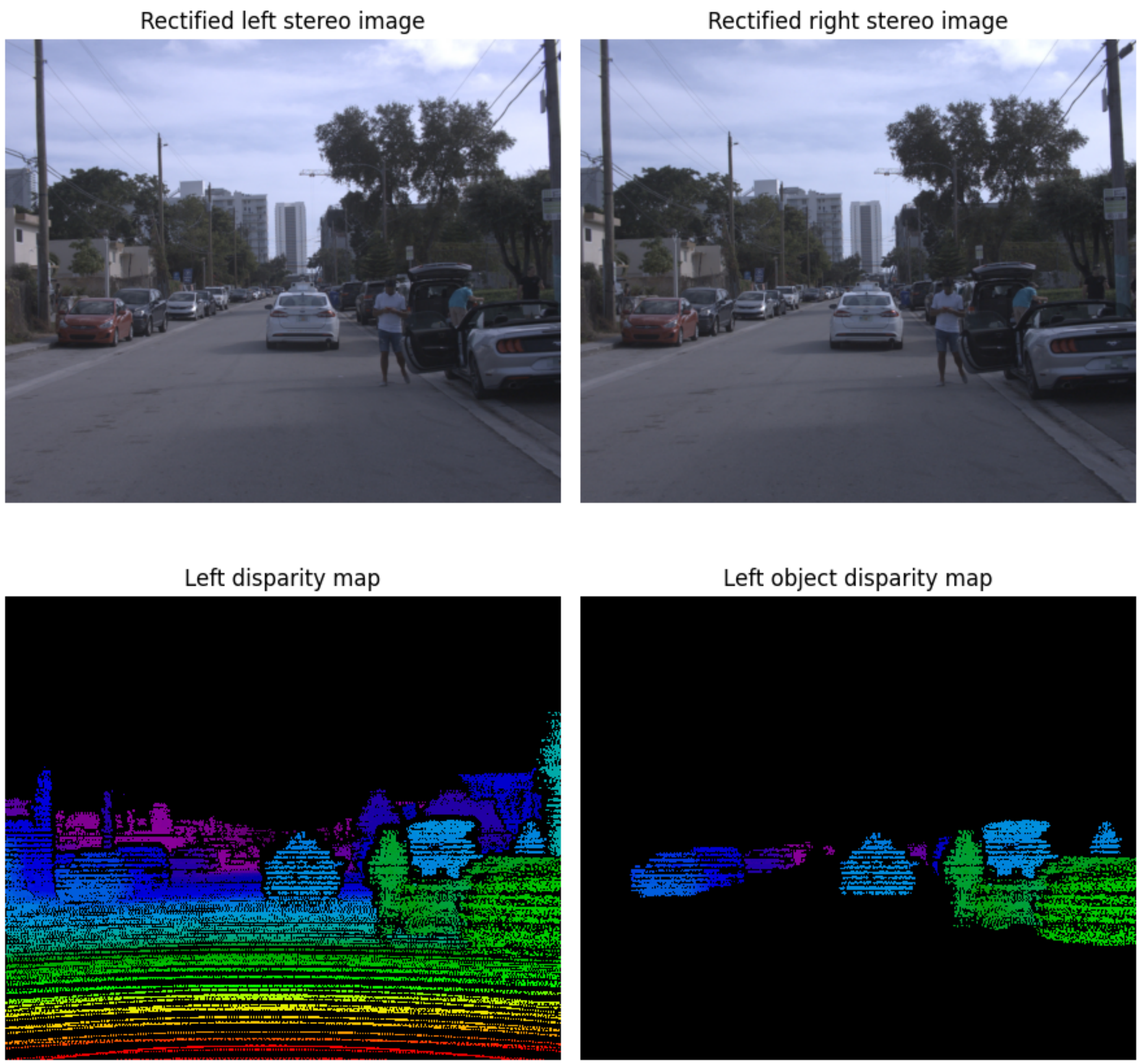
In [2]:
```
# Loading the rectified stereo images.
stereo_front_left_rect_image = stereo_data_loader.get_rectified_stereo_image(left_stereo_img_fpaths[idx])
stereo_front_right_rect_image = stereo_data_loader.get_rectified_stereo_image(right_stereo_img_fpaths[idx])

# Loading the ground-truth disparity maps.
stereo_front_left_rect_disparity = stereo_data_loader.get_disparity_map(disparity_map_fpaths[idx])

# Loading the ground-truth disparity maps for foreground objects only.
stereo_front_left_rect_objects_disparity = stereo_data_loader.get_disparity_map(disparity_obj_map_fpaths[idx])
```

Visualization

Let's visualize the stereo image pair and its ground-truth disparities.

In [3]:
```
# Dilating the disparity maps for a better visualization.
stereo_front_left_rect_disparity_dil = cv2.dilate(
    stereo_front_left_rect_disparity,
    kernel=np.ones((2, 2), np.uint8),
    iterations=7,
)

stereo_front_left_rect_objects_disparity_dil = cv2.dilate(
    stereo_front_left_rect_objects_disparity,
    kernel=np.ones((2, 2), np.uint8),
    iterations=7,
)

plt.figure(figsize=(9, 9))
plt.subplot(2, 2, 1)
plt.title("Rectified left stereo image")
plt.imshow(stereo_front_left_rect_image)
plt.axis("off")
plt.subplot(2, 2, 2)
plt.title("Rectified right stereo image")
plt.imshow(stereo_front_right_rect_image)
plt.axis("off")
plt.subplot(2, 2, 3)
plt.title("Left disparity map")
plt.imshow(
    stereo_front_left_rect_disparity_dil,
    cmap="nipy_spectral",
    vmin=0,
    vmax=192,
    interpolation="none",
)
plt.axis("off")
plt.subplot(2, 2, 4)
plt.title("Left object disparity map")
plt.imshow(
    stereo_front_left_rect_objects_disparity_dil,
    cmap="nipy_spectral",
    vmin=0,
    vmax=192,
    interpolation="none",
)
plt.axis("off")
plt.tight_layout()
```

Figure 1

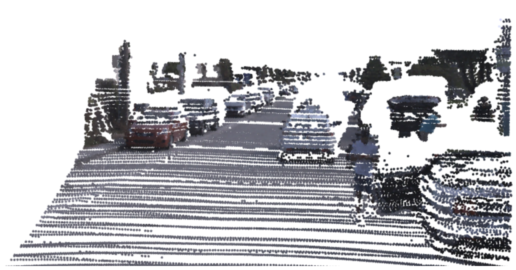


x=2410. y=1211. [0.0]

Recovering and visualizing the true depth from the disparity map

Here we use the following relationship to recover the depth from disparity: $z = \frac{fB}{d}$, where z is the depth in meters, f is the focal length in pixels, B is the baseline in meters, and d is the disparity in pixels.

In [4]:
```python
# First, we need to load the camera calibration. Specifically, we want the camera intrinsic parameters.
calib_data = stereo_data_loader.get_log_calibration_data(log_id=log_id)
camera_config = get_calibration_config(calib_data, camera_name=STEREO_FRONT_LEFT_RECT)

# Getting the focal lenght and baseline. Note that the baseline is constant for the Argoverse stereo rig setup.
focal_lenght = camera_config.intrinsic[0, 0]  # Focal length in pixels.
BASELINE = 0.2986  # Baseline in meters.

# We consider disparities greater than zero to be valid disparities.
# A zero disparity corresponds to an infinite distance.
valid_pixels = stereo_front_left_rect_disparity > 0

# Using the stereo relationship previsouly described, we can recover the depth map by:
stereo_front_left_rect_depth = \
    np.float32((focal_lenght * BASELINE) / (stereo_front_left_rect_disparity + (1.0 - valid_pixels)))

# Recovering the colorized point cloud using Open3D.
left_image_o3d = o3d.geometry.Image(stereo_front_left_rect_image)
depth_o3d = o3d.geometry.Image(stereo_front_left_rect_depth)
rgbd_image_o3d = o3d.geometry.RGBDImage.create_from_color_and_depth(
    left_image_o3d,
    depth_o3d,
    convert_rgb_to_intensity=False,
    depth_scale=1.0,
    depth_trunc=200,
)
pinhole_camera_intrinsic = o3d.camera.PinholeCameraIntrinsic()
pinhole_camera_intrinsic.intrinsic_matrix = camera_config.intrinsic[:3, :3]
pinhole_camera_intrinsic.height = camera_config.img_height
pinhole_camera_intrinsic.width = camera_config.img_width
pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_image_o3d, pinhole_camera_intrinsic)

# Showing the colorized point cloud using the interactive Plotly.
points = np.asarray(pcd.points)
colors = np.asarray(pcd.colors)

fig = go.Figure(
    data=[
        go.Scatter3d(
            x=points[:, 0],
            y=points[:, 1],
            z=points[:, 2],
            mode="markers",
            marker=dict(size=1, color=colors),
        )
    ],
    layout=dict(
        scene=dict(
            xaxis=dict(visible=False),
            yaxis=dict(visible=False),
            zaxis=dict(visible=False),
            aspectmode="data",
        )
    ),
)
fig.show()
```



Predicting the disparity map from a stereo pair image

Here we provide a baseline to predict a disparity map given the left and right rectified stereo images. We choose the classic **Semi-Global Matching (SGM)** algorithm and used its OpenCV implementation. Please check the [OpenCV documentation (https://docs.opencv.org/3.4/d2/d85/classcv_1_1StereoSGBM.html)](https://docs.opencv.org/3.4/d2/d85/classcv_1_1StereoSGBM.html) and the great [SGM paper (https://core.ac.uk/download/pdf/11134866.pdf)](https://core.ac.uk/download/pdf/11134866.pdf), if you are interested in more details.

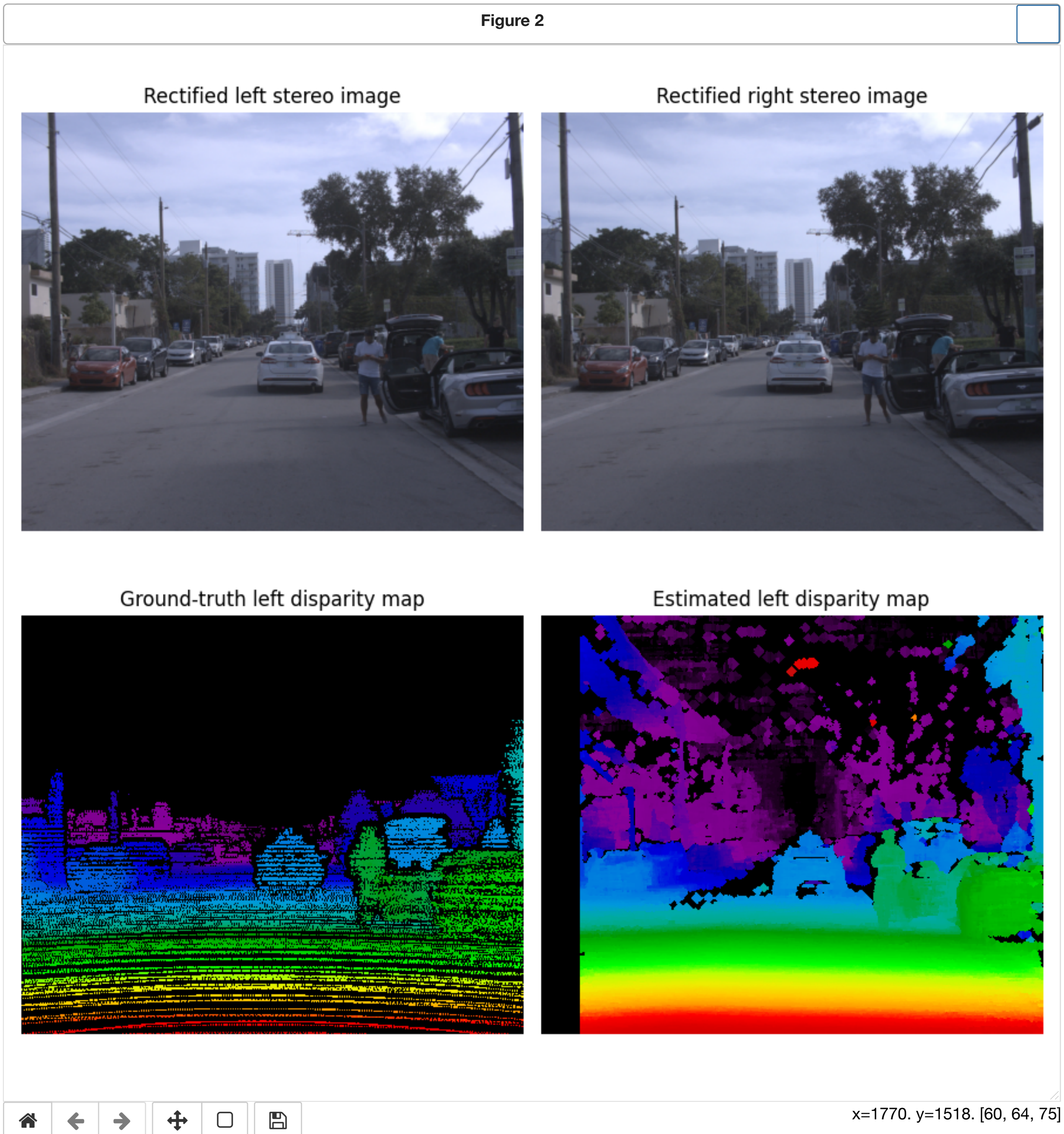
In [5]:

```
# Defining the SGM parameters (please check the OpenCV documentation for details).
# We found this parameters empirically and based on the Argoverse Stereo data.
max_disp = 192
win_size = 10
uniqueness_ratio = 15
speckle_window_size = 200
speckle_range = 2
block_size = 11
P1 = 8 * 3 * win_size ** 2
P2 = 32 * 3 * win_size ** 2

# Defining the Weighted Least Squares (WLS) filter parameters.
lmbda = 0.1
sigma = 1.0

# Defining the SGM left matcher.
left_matcher = cv2.StereoSGBM_create(
    minDisparity=0,
    numDisparities=max_disp,
    blockSize=block_size,
    P1=P1,
    P2=P2,
    disp12MaxDiff=max_disp,
    uniquenessRatio=uniqueness_ratio,
    speckleWindowSize=speckle_window_size,
    speckleRange=speckle_range,
    mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY,
)

# Defining the SGM right matcher needed for the left-right consistency check in the WLS filter.
right_matcher = cv2.ximgproc.createRightMatcher(left_matcher)

# Defining the WLS filter.
wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=left_matcher)
wls_filter.setLambda(lmbda)
wls_filter.setSigmaColor(sigma)

# Computing the disparity maps.
left_disparity = left_matcher.compute(stereo_front_left_rect_image, stereo_front_right_rect_image)
right_disparity = right_matcher.compute(stereo_front_right_rect_image, stereo_front_left_rect_image)

# Applying the WLS filter.
left_disparity_pred = wls_filter.filter(left_disparity, stereo_front_left_rect_image, None, right_disparity)

# Recovering the disparity map.
# OpenCV produces a disparity map as a signed short obtained by multiplying subpixel shifts with 16.
# To recover the true disparity values, we need to divide the output by 16 and convert to float.
left_disparity_pred = np.float32(left_disparity_pred) / 16.0

# OpenCV will also set negative values for invalid disparities where matches could not be found.
# Here we set all invalid disparities to zero.
left_disparity_pred[left_disparity_pred < 0] = 0
```

Visualizing the results

Here we plot the stereo image pair, the ground truth disparity, and the estimated disparity by SGM.

In [6]:
```
plt.figure(figsize=(9, 9))
plt.subplot(2, 2, 1)
plt.title("Rectified left stereo image")
plt.imshow(stereo_front_left_rect_image)
plt.axis("off")
plt.subplot(2, 2, 2)
plt.title("Rectified right stereo image")
plt.imshow(stereo_front_right_rect_image)
plt.axis("off")
plt.subplot(2, 2, 3)
plt.title("Ground-truth left disparity map")
plt.imshow(
    stereo_front_left_rect_disparity_dil,
    cmap="nipy_spectral",
    vmin=0,
    vmax=192,
    interpolation="none",
)
plt.axis("off")
plt.subplot(2, 2, 4)
plt.title("Estimated left disparity map")
plt.imshow(
    left_disparity_pred,
    cmap="nipy_spectral",
    vmin=0,
    vmax=192,
    interpolation="none"
)
plt.axis("off")
plt.tight_layout()
```



Recovering and visualizing the predicted point cloud from the disparity map

In [7]:
```
# We consider disparities greater than zero to be valid disparities.
# A zero disparity corresponds to an infinite distance.
valid_pixels = left_disparity_pred > 0

# Using the stereo relationship previsouly described, we can recover the predicted depth map by:
left_depth_pred = \
    np.float32((focal_lenght * BASELINE) / (left_disparity_pred + (1.0 - valid_pixels)))

# Recovering the colorized point cloud using Open3D.
left_image_o3d = o3d.geometry.Image(stereo_front_left_rect_image)
depth_o3d = o3d.geometry.Image(left_depth_pred)
rgbd_image_o3d = o3d.geometry.RGBDImage.create_from_color_and_depth(
    left_image_o3d,
    depth_o3d,
    convert_rgb_to_intensity=False,
    depth_scale=1.0,
    depth_trunc=200,
)
pinhole_camera_intrinsic = o3d.camera.PinholeCameraIntrinsic()
pinhole_camera_intrinsic.intrinsic_matrix = camera_config.intrinsic[:3, :3]
pinhole_camera_intrinsic.height = camera_config.img_height
pinhole_camera_intrinsic.width = camera_config.img_width
pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_image_o3d, pinhole_camera_intrinsic)

# Showing the colorized point cloud using the interactive Plotly.
points = np.asarray(pcd.points)
# Randomly sampling indices for faster rendering.
indices = np.random.randint(len(points), size=100000)
points = points[indices]
colors = np.asarray(pcd.colors)[indices]

fig = go.Figure(
    data=[
        go.Scatter3d(
            x=points[:, 0],
            y=points[:, 1],
            z=points[:, 2],
            mode="markers",
            marker=dict(size=1, color=colors),
        )
    ],
    layout=dict(
        scene=dict(
            xaxis=dict(visible=False),
            yaxis=dict(visible=False),
            zaxis=dict(visible=False),
            aspectmode="data",
        ),
    ),
)
fig.show()
```

Saving the predicted disparity map to disk

We encode the disparity maps using the raster-graphics PNG file format for lossless data compression. The disparity images are saved as uint16 and its values range is [0, 256].

A zero "0" value indicates an invalid disparity/pixel. For the ground-truth disparity, zero means that no ground truth is available.

To recover the real disparity value, we first convert the uint16 value to a float and then divide it by 256.0.

In [8]:
```
# Encoding the real disparity values to an uint16 data format to save as an uint16 PNG file.
left_disparity_pred = np.uint16(left_disparity_pred * 256.0)

timestamp = int(Path(disparity_map_fpaths[idx]).stem.split("_")[-1])

# Change the path to the directory you would like to save the result.
# The log id must be consistent with the stereo images' log id.
save_dir_disp = f"/tmp/results/sgm/stereo_output/{log_id}/"
Path(save_dir_disp).mkdir(parents=True, exist_ok=True)

# The predicted disparity filename must have the format: 'disparity_[TIMESTAMP OF THE LEFT STEREO IMAGE].png'
filename = f"{save_dir_disp}/disparity_{timestamp}.png"

# Writing the PNG file to disk.
cv2.imwrite(filename, left_disparity_pred)
```

Out[8]:
```
True
```

Evaluating the results

Our evaluation algorithm computes the percentage of bad pixels averaged over all ground-truth pixels, similar to the [KITTI Stereo 2015 (http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo)](http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo) benchmark.

We consider the disparity of a pixel to be correctly estimated if the absolute disparity error is less than a threshold **or** its relative error is less than 10% of its true value. We define three disparity error thresholds: 3, 5, and 10 pixels.

Our [EvalAI leaderboard (https://eval.ai/web/challenges/challenge-page/1704/overview)](https://eval.ai/web/challenges/challenge-page/1704/overview) ranks all methods according to the number of bad pixels using a threshold of 10 pixels (i.e. **all:10**). Some stereo matching methods such as SGM might provide sparse disparity maps, meaning that some pixels will not have valid disparity values. In those cases, we interpolate the predicted disparity map using a simple nearest neighbor interpolation as in the [KITTI Stereo 2015 (http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo)](http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo) benchmark to assure we can compare it to our semi-dense ground-truth disparity map. Current deep stereo matching methods normally predict disparity maps with 100% density. Thus, an interpolation step is not needed for the evaluation.

The disparity errors metrics are the following:

- **all**: Percentage of stereo disparity errors averaged over all ground-truth pixels in the reference frame (left stereo image).
- **bg**: Percentage of stereo disparity errors averaged only over background regions.
- **fg**: Percentage of stereo disparity errors averaged only over foreground regions.

The $*$ (asterisk) means that the evaluation is performed using only the algorithm predicted disparities. Even though the disparities might be sparse, they are not interpolated.

We evaluate all metrics using three error thresholds: 3, 5, or 10 pixels. The notation is then: **all:3**, **all:5**, **all:10**, **fg:3**, **fg:5**, **fg:10**, and so on.

In [9]:
```python
# Path to the predicted disparity maps.
pred_dir = Path(save_dir_disp)

# Path to the ground-truth disparity maps.
gt_dir = Path(f"{data_dir}/disparity_maps_v1.1/{split_name}/{log_id}")

# Path to save the disparity error image.
save_figures_dir = Path("/tmp/results/sgm/figures/")
save_figures_dir.mkdir(parents=True, exist_ok=True)

print(pred_dir)
print(gt_dir)

# Creating the stereo evaluator.
evaluator = StereoEvaluator(
    pred_dir,
    gt_dir,
    save_figures_dir,
    save_disparity_error_image=True,
    num_procs=-1,
)

# Running the stereo evaluation.
metrics = evaluator.evaluate()

# Printing the quantitative results (using json trick for organized printing).
print(f"{json.dumps(metrics, sort_keys=False, indent=4)}")
```

```
/tmp/results/sgm/stereo_output/273c1883-673a-36bf-b124-88311b1a80be
/data/datasets/stereo/argoverse1/disparity_maps_v1.1/train/273c1883-673a-36bf-b124-88311b1a80be
```

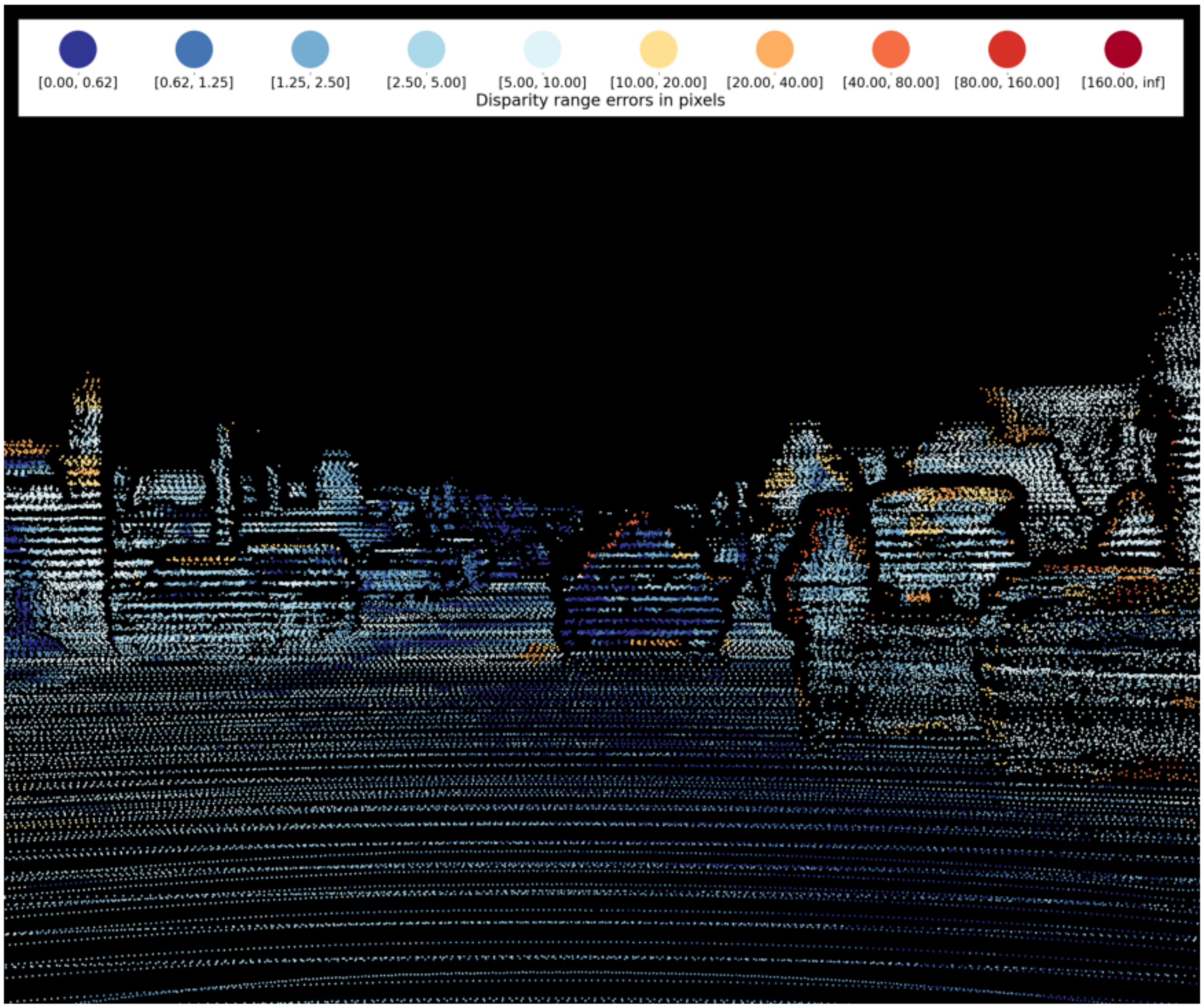

```
{
    "all:10": 5.446183637243827,
    "fg:10": 9.600825877494838,
    "bg:10": 3.556616323655998,
    "all*:10": 3.8116157374507393,
    "fg*:10": 7.697195243714259,
    "bg*:10": 1.664911488892232,
    "all:5": 21.389381959012425,
    "fg:5": 16.07019958706125,
    "bg:5": 23.808592221613583,
    "all*:5": 17.17165191549842,
    "fg*:5": 14.114550240537351,
    "bg*:5": 18.860638884709893,
    "all:3": 29.672960034425262,
    "fg:3": 18.530626290433585,
    "bg:3": 34.74059003051882,
    "all*:3": 24.646294980295885,
    "fg*:3": 16.61069256603431,
    "bg*:3": 29.085803119201646,
    "#parameters (M)": 0.001315,
    "#flops (T)": 8.64e-08,
    "#activations (G)": 2.156e-06,
    "Inference time (ms)": 1.0116778751214346,
    "Device name": "Tesla V100-SXM2-32GB"
}
```


Plotting the disparity error image

We compute the disparity error image as in the [KITTI Stereo 2015 (http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo)](http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo) benchmark. The disparity error map uses a log colormap depicting correct estimates in blue and wrong estimates in red color tones. We define correct disparity estimates when the absolute disparity error is less than 10 pixels and the relative error is less than 10% of its true value.

In [16]:
```python
# Reading the PNG disparity error image and converting it to RGB.
disparity_error_image_path = f"{save_figures_dir}/{log_id}/disparity_error_{timestamp}.png"
disparity_error_image = cv2.cvtColor(cv2.imread(disparity_error_image_path), cv2.COLOR_BGR2RGB)

# Showing the disparity error image.
plt.figure(figsize=(9, 9))
plt.imshow(disparity_error_image)
plt.axis("off")
plt.tight_layout()
```

Figure 3



** Updates for the Stereo Competition 2022 **

For our 2022 stereo competition, we want to emphasize real-time performance. Although all entries will appear on the EvalAI leaderboard, your model must run faster than **200 ms** per depth prediction (i.e., during forward pass) on an Nvidia Tesla V100 GPU to be elegible for prizes.

For a fair competition, we highly encourage the use of the hardware platform [Amazon EC2 p3.2xlarge instance](https://aws.amazon.com/ec2/instance-types/p3/) which has a single Tesla V100 GPU.

To benchmark yor model, we provide a script to generate a model analysis report and the disparity predictions as demonstraded in the cells below.

For those who cannot use the deginated hardware platform, we will estimate the final inference latency of the model as if it was run on a Tesla V100 GPU using the [Deep Learning GPU Benchmark: A Latency-based Approach](https://mtli.github.io/gpubench/) tool.

Benchmarking your model

For our benchmark, we are only supporting PyTorch models.

To run our benchmark script you will need the following dependencies:

- **A cuda-enabled PyTorch installation with torchvision**
- **fvcore (https://github.com/facebookresearch/fvcore): See instructions on how to install here (https://github.com/facebookresearch/fvcore).**

NOTE: If you model is based on TensorFlow or cpu, please contact us so we can try to help you profiling the code.

To submit the results from your stereo matching method for evaluation in our EvalAI server, you will need to run your method on the entire test set (15 log sequences).

We provide a script ('generate_stereo_results.py') to generate the disparity predictions (PNG files) and benchmark you model over the entire test set. It evaluates latency, and number of flops, activations, and parameters of your model during the forward pass. Then, it generates the final submission zip file. The script checks whether the predicted dispartity maps have the correct shape and type, the number of logs are correct, and if the report file (model_analysis_report.txt) is available.

It will take a couple of minutes (less than 10 minutes) to loop over the test dataset, perform the predictions, benchmark the model, and generate the submission zip file. If the zip file is created succesfully and it will be ready for submission.

Please use the scrip to generate the benchmarking results and the submission file. This avoids submission errors in the EvalAI server.

The final directory structure you should have is shown below:

```
submission
└───
|    stereo_output
|    └───0f0d7759-fa6e-3296-b528-6c862d061bdd
|    |       disparity_315974292602180504.png
|    |       disparity_315974292801980264.png
|    |                     .
|    |                     .
|    |                     .
|    |
|    └───673e200e-944d-3b40-a447-f83353bd85ed
|    └───764abf69-c7a0-32c3-97f5-330de68e13af
|    └───model_analysis_report.txt
└─── stereo_output.zip
```

In [15]:

```python
from pathlib import Path

import torch
import torchvision

from argoverse.evaluation.stereo.generate_stereo_results import generate_stereo_results


# Define stereo model (example of dummy model below)
class YourStereoModel(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv_layer1 = torch.nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv_layer2 = torch.nn.Conv2d(in_channels=32, out_channels=1, kernel_size=3, padding=1)

    def forward(self, input1: torch.Tensor, input2: torch.Tensor) -> torch.Tensor:
        output1 = self.conv_layer1(input1)
        output2 = self.conv_layer1(input2)
        output = self.conv_layer2(torch.cat((output1, output2), axis=1))
        return output

your_model = YourStereoModel()

# Uncomment and load the model checkpoint to perform the final predictions
# state_dict = torch.load("PATH_TO_MODEL_CHECKPOINT")
# your_model.load_state_dict(model_dict)

# Change paths accordingly
# Path to the Argoverse Stereo dataset
data_dir = Path("/data/datasets/stereo/argoverse1/")
# Path to save the submission files (benchmark report, predictions, zip)
output_dir = Path("/home/ubuntu/argoverse-stereo-competition/submission/stereo_output")

# Example of transforms to be applied to the input data
# Change it accordingly. Transforms are ignored during latency computation.
transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.Resize(size=(528, 624)),
        torchvision.transforms.Normalize(
            mean=(0.485, 0.456, 0.406),  # ResNet mean image normalization
            std=(0.229, 0.224, 0.225),  # ResNet std image normalization
        ),
    ]
)

# Example of output report:
# Device name: Tesla V100-SXM2-16GB
# #Parameters: 737
# #Flops: 379551744
# #Activations: 10872576
# Mean forward time (ms): 1.0553814383049989
# Std forward time (ms): 0.05229927585160601
# Input size 0: torch.Size([1, 3, 528, 624])
# Input size 1: torch.Size([1, 3, 528, 624])
# | module                 | #parameters or shape   | #flops    | #activations   |
# |:-----------------------|:-----------------------|:----------|:---------------|
# | model                  | 0.737K                 | 0.38G     | 10.873M        |
# |  conv_layer1           |  0.448K                |  0.285G   |  10.543M       |
# |   conv_layer1.weight   |   (16, 3, 3, 3)        |           |                |
# |   conv_layer1.bias     |   (16,)                |           |                |
# |  conv_layer2           |  0.289K                |  94.888M  |  0.329M        |
# |   conv_layer2.weight   |   (1, 32, 3, 3)        |           |                |
# |   conv_layer2.bias     |   (1,)                 |           |                |

# NOTE: If your model outputs a disparity map with a different shape than the ground-truth disparity,
# the script will resize it to the original size (i.e., 2464 x 2056 pixels) using bilinear interpolation
# for the submission. Feel free to change the post-processing transformation according to your needs in the scrip
generate_stereo_results(your_model, data_dir, output_dir, transforms)

# Running the stereo evaluation again to take into account the model analysis report.
metrics = evaluator.evaluate()

# Printing the results (using json trick for organized printing).
print(f"{json.dumps(metrics, sort_keys=False, indent=4)}")
```

```
Forward passing the test dataset, computing latency, and saving predictions...

100%|████████████████████████████████████████| 1094/1094 [02:32<00:00,  7.18it/s]

Computing number of flops, activations, and parameters...
```

```
Device name: Tesla V100-SXM2-32GB
#Parameters: 737
#Flops: 379551744
#Activations: 10872576
Mean forward time (ms): 0.9865462810509383
Std forward time (ms): 0.09089147283422745
Input size 0: torch.Size([1, 3, 528, 624])
Input size 1: torch.Size([1, 3, 528, 624])
| module                 | #parameters or shape   | #flops   | #activations   |
|:-----------------------|:-----------------------|:---------|:---------------|
| model                  | 0.737K                 | 0.38G    | 10.873M        |
|  conv_layer1           |  0.448K                |  0.285G  |  10.543M       |
|   conv_layer1.weight   |   (16, 3, 3, 3)        |          |                |
|   conv_layer1.bias     |   (16,)                |          |                |
|  conv_layer2           |  0.289K                |  94.888M |  0.329M        |
|   conv_layer2.weight   |   (1, 32, 3, 3)        |          |                |
|   conv_layer2.bias     |   (1,)                 |          |                |
Saved report at /home/ubuntu/argoverse-stereo-competition/submission/stereo_output/model_analysis_report.txt.
Checking outputs and generating submission file. Please wait...

100%|████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████| 16/16 [01:07<00:00,  4.20s/it]

Creating zip file for submission...
Zip file (/home/ubuntu/argoverse-stereo-competition/submission/stereo_output.zip) created succesfully. Please s
ubmit it to EvalAI for evaluation.
```


```
{
    "all:10": 5.446183637243827,
    "fg:10": 9.600825877494838,
    "bg:10": 3.556616323655998,
    "all*:10": 3.8116157374507393,
    "fg*:10": 7.697195243714259,
    "bg*:10": 1.664911488892232,
    "all:5": 21.389381959012425,
    "fg:5": 16.07019958706125,
    "bg:5": 23.808592221613583,
    "all*:5": 17.17165191549842,
    "fg*:5": 14.114550240537351,
    "bg*:5": 18.860638884709893,
    "all:3": 29.672960034425262,
    "fg:3": 18.530626290433585,
    "bg:3": 34.74059003051882,
    "all*:3": 24.646294980295885,
    "fg*:3": 16.61069256603431,
    "bg*:3": 29.085803119201646,
    "#parameters (M)": 0.001315,
    "#flops (T)": 8.64e-08,
    "#activations (G)": 2.156e-06,
    "Inference time (ms)": 1.0116778751214346,
    "Device name": "Tesla V100-SXM2-32GB"
}
```


Submitting results to EvalAI

Here are some directions to submit your results (i.e., the generated `stereo_output.zip`) to the EvalAI server.

Your zip file will likely be large (e.g., ~3 GB). You will need to use the EvalAI command line interface (EvalAI-CLI) to submit such large files. Please follow the instructions described here (https://github.com/Cloud-CV/evalai-cli) for installing it. In addition, you can follow the submission instructions in our EvalAI Stereo Competition page (https://eval.ai/web/challenges/challenge-page/1704/submission).

Once you have the EvalAI-CLI up and running, you can submit your results using the following command. Please ensure to add as much details as possible about your method (e.g., a brief description, link to paper, link to code if publicly available, etc.).

Note that you can only submit to EvalAI once a day. Please do not submit multiple times using different accounts. We are actively monitoring it.

```
$ evalai challenge 1704 phase 3365 submit --file /tmp/results/your-model/stereo_output.zip --large
```

If everything goes well, you can check the status of your submission using the command:

```
$ evalai submission 'YOUR SUBMISSION ID'`
```

The evaluation normally takes about 2 minutes to complete. Once completed you can check the results in the My Submissions (https://eval.ai/web/challenges/challenge-page/1704/my-submission) session in the EvalAI web interface. Then, you can select to show your method in our leaderboard (https://eval.ai/web/challenges/challenge-page/1704/evaluation) and check how it compares against our baselines and others!

Well done completing this tutorial and good luck!