DATASET AND EVALUATION OF SELF-SUPERVISED LEARNING FOR PANORAMIC DEPTH ESTIMATION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ryan Nett

December 2020
COMMITTEE MEMBERSHIP

TITLE: Dataset and Evaluation of Self-Supervised Learning for Panoramic Depth Estimation

AUTHOR: Ryan Nett

DATE SUBMITTED: December 2020

COMMITTEE CHAIR: Jonathan Ventura, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Zoë Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Franz Kurfess, Ph.D.
Professor of Computer Science
Dataset and Evaluation of Self-Supervised Learning for Panoramic Depth Estimation
Ryan Nett

Depth detection is a very common computer vision problem. It shows up primarily in robotics, automation, or 3D visualization domains, as it is essential for converting images to point clouds. One of the poster child applications is self driving cars [6]. Currently, the best methods for depth detection are either very expensive, like LIDAR, or require precise calibration, like stereo cameras. These costs have given rise to attempts to detect depth from a monocular camera (a single camera). While this is possible, it is harder than LIDAR or stereo methods since depth can’t be measured from monocular images, it has to be inferred. A good example is covering one eye: you still have some idea how far away things are, but it’s not exact [5]. Neural networks are a natural fit for this. Here, we build on previous neural network methods by applying a recent state of the art model to panoramic images in addition to pinhole ones and performing a comparative evaluation. First, we create a simulated depth detection dataset that lends itself to panoramic comparisons and contains pre-made cylindrical and spherical panoramas. We then modify monodepth2 [4] to support cylindrical and cubemap panoramas, incorporating current best practices for depth detection on those panorama types, and evaluate its performance for each type of image using our dataset. We also consider the resources used in training and other qualitative factors.
ACKNOWLEDGMENTS

Thanks to:

• My parents, for supporting me throughout my education.

• Andrew Guenther, for uploading this template
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation for Monocular Depth Perception</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Difficulties</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Use of Panoramas</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Contribution</td>
<td>3</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Depth Estimation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Panoramas for Depth Estimation</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Cylindrical</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Cubemap</td>
<td>9</td>
</tr>
<tr>
<td>2.3 CARLA</td>
<td>10</td>
</tr>
<tr>
<td>3 Dataset</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Simulations</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Stitching</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Hosting</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1 Raw Data</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Python Package</td>
<td>20</td>
</tr>
<tr>
<td>4 Baseline Experiment</td>
<td>22</td>
</tr>
<tr>
<td>5 Analysis</td>
<td>25</td>
</tr>
</tbody>
</table>
5.1 Pinhole ......................................................... 25
5.2 Cylindrical ........................................................ 27
5.3 Cubemap .......................................................... 27
6 Implementation ...................................................... 30
  6.1 Monodepth2 .................................................... 30
  6.2 Modifications .................................................. 32
    6.2.1 Cylindrical ................................................. 33
    6.2.2 Cubemap ................................................... 33
7 Results ............................................................ 36
  7.1 Evaluation ..................................................... 36
  7.2 Quantitative Results ......................................... 37
  7.3 Qualitative Results ........................................... 38
  7.4 Improvements .................................................. 40
  7.5 Resource Requirements ....................................... 41
8 Conclusion ........................................................ 42
  8.1 Future Work ................................................... 43
BIBLIOGRAPHY ....................................................... 44


LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>CARLA camera intrinsics. Note that cubemap panoramas use Pin-hole 90° images for each side. The horizontal and vertical coordinates are listed in the first row (in that order). The image size is width $\times$ height in pixels.</td>
</tr>
<tr>
<td>4.1</td>
<td>Results from Sharma, Nett, and Ventura [13] from using CylindricalSfmLearner on carla data, with and without wrap padding (with labels fixed). Units are in meters.</td>
</tr>
<tr>
<td>7.1</td>
<td>Quantitative results.</td>
</tr>
<tr>
<td>7.2</td>
<td>Improved pinhole results.</td>
</tr>
<tr>
<td>7.3</td>
<td>Training time with largest batch size that fit, and approximate GPU memory required for a single example, for each panorama type.</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>1.1</td>
<td>Panorama types, from Sharma and Ventura [15]. Spherical, cylindrical, and cubemap panoramas are shown, as well as how rectangular regions (like convolutional filters) are distorted on them.</td>
</tr>
<tr>
<td>2.1</td>
<td>Wrap padding from Sharma and Ventura [15]. This is done to enforce wrapping consistency in cylindrical panoramas.</td>
</tr>
<tr>
<td>2.2</td>
<td>Cube padding from Cheng et al. [2]. Image is from the paper’s associated GitHub repository. This is done to enforce edge consistency in cubemap panoramas.</td>
</tr>
<tr>
<td>3.1</td>
<td>Each of CARLA’s towns. Produced for and first used in Sharma, Nett, and Ventura [13].</td>
</tr>
<tr>
<td>3.2</td>
<td>A sample stitched cylindrical disparity and color image from CARLA.</td>
</tr>
<tr>
<td>3.3</td>
<td>A sample stitched spherical disparity and color image from CARLA.</td>
</tr>
<tr>
<td>3.4</td>
<td>Rendering of the exported mesh created from our spherical stitched data.</td>
</tr>
<tr>
<td>4.1</td>
<td>A sample of estimated depth from a CARLA image, produced for and first appearing in Sharma, Nett, and Ventura [13].</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of monodepth2, from Godard et al. [4].</td>
</tr>
<tr>
<td>5.2</td>
<td>Wrap padding from Sharma and Ventura [15]. This is done to enforce wrapping consistency in cylindrical panoramas.</td>
</tr>
<tr>
<td>5.3</td>
<td>Cube padding from Cheng et al. [2]. Image is from the paper’s associated GitHub repository. This is done to enforce consistency around the edges of cubemap panoramas.</td>
</tr>
<tr>
<td>7.1</td>
<td>Cylindrical ground truth and predicted disparity.</td>
</tr>
<tr>
<td>7.2</td>
<td>Pinhole ground truth and predicted disparity.</td>
</tr>
</tbody>
</table>
7.3 Cubemap predicted depth at epochs 4 and 6.
When we look at something, we automatically get a good idea of how far away it is. This is rather essential for everyday tasks, many of which we would like to automate. One of the best examples is self-driving cars [6]. They need to be able to create a 3D map of their surroundings, which is very necessary for driving. To be able to automate tasks like this, we need a way for computers to perceive depth, or estimate it from an image.

1.1 Motivation for Monocular Depth Perception

The current best technology for depth detection is LIDAR, which uses lasers to create a very accurate, if sparse, depth map. However, LIDAR is very expensive. An industry standard LIDAR sensor is $75,000, which is more than the cost of most of the cars they would be used on [7]. While there are companies claiming to sell much cheaper LIDAR sensors they say are just as good or better than the standard ones, none of them have been rigorously tested and some companies don’t publish their prices [8]. Another option is to use two cameras to triangulate the position of objects from images (or video). This is (mostly) what our brains do [5]. However, the cameras have to be precisely calibrated, and if there is any change in the position of either camera, the depth predictions will be incorrect. The easiest way to get depth maps would be to use a single camera and predict the depth from its images. We are somewhat capable of this: if you close one eye, your depth perception doesn’t entirely go away, although it does get noticeably worse [5]. This reflects the current
state of monocular depth perception: it is possible, but quite difficult. Recently, there has been a fair amount of success using neural networks to predict depth from monocular images. These methods try to replicate the same kind of inference that lets you perceive depth with one eye closed.

1.2 Difficulties

A major difficulty with predicting depth is that gathering ground-truth data is hard. The best depth perception technology, LIDAR, only gets sparse depth maps. This makes supervised training of depth estimation models infeasible. To combat this, most modern depth estimation networks use self-supervised training using a pose network (an object’s pose is its position and orientation) [19]. While calculating depth from an image is hard, calculating pose is much easier, and it gives us hints as to what the depth should be. If we have two sequential frames of a video and have calculated the change in pose to be a move two meters forward, we know that the depth of most things in the second image should be two meters closer. This can be used to enforce “internal consistency” in our depth estimation network via a loss function and has proven successful at helping them train in an unsupervised manner. However, ground-truth depth maps are still needed for evaluation.

1.3 Use of Panoramas

A somewhat recent idea has been to use panoramic images instead of simple “pinhole” images (these are just normal images from a single camera). While this may help depth estimation some, the biggest improvement is likely to be in the pose prediction network. It is easier to tell how far you have moved when you are looking out the side window of a car, compared to looking out the front. Panoramas give the network
access to this “sideways” data, as well as data in all other directions. It stands to reason that neural networks would find pose prediction from a panorama easier, just like we would. It also makes sense to use panoramas independently of any accuracy benefits they may provide: a self driving car will need to know the locations of objects all around it, it might as well predict the depth all at once, rather than from separate cameras. This also allows for the entire depth estimation to take into account the entire image and prevents separate depth estimations from getting “out of sync”.

There are three main types of panoramas: spherical, cylindrical, and cubemap, shown in Figure 1.1.

1.4 Contribution

While panoramas seem promising, there has been no quantitative or qualitative comparison of pinhole images and different panorama types. This is likely in part because there is no existing dataset with ground truth depth and all of the data needed for all
three types of panoramas. This paper contributes a synthetic dataset with spherical and cylindrical panoramas, all 6 sides of pinhole images (for cubemap panoramas and regular pinhole training), and ground truth depth and pose. We validate our dataset using CylindricalSfmLeadner [13], and then use our dataset to compare monodepth2’s (a state of the art monocular depth estimation network [4]) performance on pinhole and panoramic images. We publish the code used to create this dataset, and instructions for creating custom new data, including weather, time, or traffic customization. We also publish a package for easily downloading and using the data, and we make our modifications to monodepth2 publicly available.
Our experiment draws heavily on existing depth estimation techniques, as well as existing methods for handling cylindrical and cubemap panoramas. We are comparing existing methods, not creating our own. We also use an existing simulator, CARLA, to generate our dataset [3].

2.1 Depth Estimation

Depth estimation has been a fairly popular area of research recently, with a major advance for monocular depth estimation coming from SfMLearner, which allowed for unsupervised training of depth and pose detection, using view synthesis as a loss [19]. It works by taking in several temporally near-by frames (typically a frame, the previous frame, and the next frame), using a depth estimation CNN independently on each frame and a relative pose detection CNN on all frames. The resulting depth map and pose information can be used to do view synthesis, which is warping a frame to see what it would look like from another frame’s pose. Given that these are indeed frames from the same video, the actual frame should match the synthesized view closely. This is used as the primary loss (L1 pixel loss summed across the entire image). Of course, not every object in the original image is present in the target one: sometimes the movement will cause objects that were in the original image to be occluded in the target, or vice versa. In addition, there will be new or missing objects around the borders of the image. To combat this, SfMLearner uses an “explainability mask”, which is effectively a per-pixel loss weight that represents how well a given
pixel can be modeled by view synthesis. A depthmap smoothness loss is also included (L2 of the depthmap’s gradient). Of some note here is the fact that this method relies the rest of the model using a differential image synthesis operation; in practice this is done using bilinear sampling.

Monodepth2 improves on SfMLearner in several ways, largely by improving edge cases of the loss function [4]. To compare two pixels (usually between an original frame and a synthesized one), monodepth2 uses a photometric error function $pe$ composed of $L_1$ and SSIM [18]:

$$pe(I_a, I_b) = \frac{\alpha}{2} (1 - SSIM(I_a, I_b)) + (1 - \alpha) || I_a - I_b ||$$

with an $\alpha$ value of 0.85. Monodepth2 also improves the smoothness loss, using an edge aware smoothness as done in the original monodepth:

$$L_{smoothness} = |\partial_x d_t^* e^{-|\partial_x I_t|} | + |\partial_y d_t^* e^{-|\partial_y I_t|} |$$

where $d_t^* = d_t / \bar{d}_t$ is the mean-normalized inverse depth [4].

A more robust approach to dealing with occlusions is also used: the original frame will be synthesized and compared against multiple target frames, and the photometric loss for each pixel will be the minimum across all of the targets (the average was used previously). This works because an object is not likely to be occluded in all frames: if it is occluded in one, the loss will be higher and thus will be ignored. This was shown to significantly reduce border artifacts and improve the sharpness of occlusion boundaries. A per-pixel mask is also used, but for different reasons. Instead of the weight from SfMLearner, this mask is a 0 or 1 value (per pixel) that is meant to filter out objects that are stationary relative to the camera. The view synthesis train method assumes a moving camera and a static scene, which is problematic when
things like the hood of your car end up in the image, or when the camera stops moving. The mask Monodepth 2 uses filters out regions that don’t change between adjacent frames, by applying a mask $\mu$ that filters out any pixels that have a higher photometric loss between the original frame and the target than between the warped frame and the target:

$$
\mu = [pe(I_t, I_p) < pe(I_t, I_o)]
$$

where $I_t$ is the target frame, $I_o$ is the original frame, and $I_p$ is the original frame warped to the target frame’s pose ([] is the Iverson bracket).

Using bilinear sampling for view synthesis causes gradient locality issues. Previous works combated this by predicting the depth and calculating the loss at multiple scales, using the intermediate layers. This however causes issues with low-res images and artifacts. Monodepth2 improves on this method by predicting scaled depth maps in the same way as normal, but then upscaling them to the input resolution before doing the view synthesis and loss calculation. This forces each stage of the network to work towards the same objects: full-scale depth estimation [4]. It also resolves many of the issues with low-res images giving ambiguous photometric loss values.

The photometric and smoothness losses are combined linearly and averaged over each pixel, scale, and batch. More details of monodepth2 will be covered in chapter 5 and 6.

### 2.2 Panoramas for Depth Estimation

Shum et al. [16] and Pelg et al. [10] are well known early papers on panoramic depth estimation. While it is still an active field, most of the work done on depth estimation focuses on pinhole images. However, for reasons outlined in chapter 1, panoramas
may help depth and pose detection. There are three types of panoramas commonly used: cylindrical, spherical, and cubemap (see Figure 1.1). Spherical panoramas are significantly harder to convolve over, so our evaluation is limited to cylindrical and cubemap panoramas (although the dataset contains spherical panoramas).

2.2.1 Cylindrical

The method we use for handling cylindrical panoramas is drawn from Sharma and Ventura [15]. To enforce the fact that the right and left edges should match, wrap padding is used, which pads each horizontal edge with a small section from the other edge. This is shown in Figure 2.1 and is done in every (non-1x1) convolutional layer, as well as in the smoothness loss and view synthesis. Note that this is standard convolutional padding: the amount of padding done is enough that the initial kernel doesn’t go outside the bounds of the padded image, which is why there is no padding done for 1x1 convolutions.
Figure 2.2: Cube padding from Cheng et al. [2]. Image is from the paper’s associated GitHub repository. This is done to enforce edge consistency in cubemap panoramas.

2.2.2 Cubemap

The methods for working with cubemaps panoramas use a somewhat similar padding method, except on all four sides. Proposed in Cheng et al. [2] and used for depth estimation in Wang et al. [17], the padding method is shown in Figure 2.2.

Wang et al. [17] proposed two additional methods for use with a self-supervised pose and depth estimation network like monodepth2. Both are used by us. The first is to do view synthesis on the whole 360° image, so that no pixels will be projected outside of the image boundary. This allows reprojected sides to draw pixels from other sides, as if the cubemap was a spherical panorama. The second is to use a pose consistency loss to train the network to estimate the same poses for each side of the cube. They use the standard deviation of the losses from each side for this, we use the variance for implementation reasons.
2.3 CARLA

To create our synthetic dataset, we need a simulator that would allow for many automatically moving vehicles and pedestrians, traffic obstacles (street lights, stop signs, etc.), multiple locations, and preferably weather control. The two best simulators we found were CARLA [3] and AirSim [12]. While AirSim has better image quality, we ended up choosing CARLA as it supports autonomous cars and pedestrians. CARLA allows us to gather dense depth maps and color images in a synchronous manner and has a fairly easy to use python API for controlling the simulation’s parameters. The details of CARLA are covered in section 3.1.
Chapter 3

DATASET

3.1 Simulations

To evaluate the use of different types of panoramas, we first had to create a dataset with enough data for all three types of panoramas, as well as having dense ground truth depth and pose. We also want to be able to simulate different environments with different conditions. To collect this data, we used CARLA [3], and as such we call our dataset the “CARLA Panoramic Depth Detection Dataset” or CPDD dataset, and informally refer to it as our carla dataset in this work. Carla includes cars, pedestrians, bicycles, and motorcycles, and all moving objects follow traffic rules. The cities are reasonable representations of a generic city, including things like tunnels, large intersections, freeways, residential areas, city centers, and suburbs (not all in the same city, though).

Carla also allows for changing the numbers of (other) cars or pedestrians, changing the weather, and changing the time. In our simulation code, we limited the weather to 7 presets, and the time to noon or sunset, giving us the following options:

- Number of cars
- Number of pedestrians
- City: 5 options
- Weather: Clear, Cloudy, Wet, WetCloudy, Soft, Mid, or Hard. Soft, Mid, and Hard are rain settings.
• Time: sunset or noon

The 5 cities have good variation, with the extremes being a small mountainous town and a large city. City 2 is the most generic, and we recommend using it as the test set. We use the first two runs of city 1 as the validation set. Images of each city are shown in Figure 3.1.

We found that the best number of cars was dependent on the city, as they are different sizes and have different street structures. Adding too few cars meant you wouldn’t see any, while adding too many meant that you would cause traffic jams and the entire simulation run would be stationary. Additionally, due to the random spawns, there is a small but non-zero chance for erroneous spawns, such as cars spawning on top of each other, that are less likely to occur if the cars have space to spread out. We found that 30 cars worked well for cities 1 and 4, 40 for 3 and 5, and only 10 for 2 (it’s the smallest). The number of pedestrians is not so important, at most it causes some crowded sidewalks. We used 200 pedestrians for each city.

We also used an index parameter to allow for multiple simulations with the same settings. The viewpoint car was randomized, as were the spawn points and behavior of other cars and pedestrians. The simulator will save the random seed used to allow for reproducible simulations.

The number of different options resulted in far too many possible simulations for us to actually take, especially considering we would want multiple runs from a single set of settings (to allow for randomness). To combat this, we decided to limit the dataset to only clear weather and noon, and take 5 runs per city. This still gives us 25 runs (each run has 1000 frames), which is more than enough training data for our needs. However, as our simulation code supports all of the listed options, it should not be that hard to generate more if needed.
Figure 3.1: Each of CARLA’s towns. Produced for and first used in Sharma, Nett, and Ventura [13].
The actual simulations were done using CARLA’s synchronous mode, ensuring that all data was taken at exactly the same time. Images were taken as 5 fps (in simulation time), and each simulation run took 1000 frames. This means that our 25 runs is the equivalent of 1.4 hours of video. Depth is collected using CARLA’s depth sensors, which is then manipulated to fit nicely into uint16. To achieve this, the depth values are stored in decimeters (10th of a meter). An example of the (cylindrical, stitched) depth is shown in Figure 3.2. Pose is calculated using CARLA’s intrinsic pose knowledge. Both color and depth images are 768 x 768, with a FOV of 100°, and the cameras are positioned along all 6 axes. 100° is used instead of 90° to provide a little overlap when stitching the sides together into panoramas. 90° resulted in some artifacts along the side edges. Note that when using cubemap panoramas the side images should be cropped to 90°, ideally with a sub-pixel method such as cv2’s getRectSubPix.

3.2 Stitching

While making cubemap panoramas from the side images is easy, making cylindrical or spherical panoramas is not. We do this and provide cylindrical and spherical panoramas as part of the dataset. In broad terms, this is done by projecting all 6 side images into 3D space as points, and then using bilinear sampling to construct the target panorama from those points.

Note that for the rest of this document, axes are oriented as followed with respect to the car/camera: X is right, Y is down, and Z is forward. This allows 3D axes to match up with the image’s X and Y coordinates and it is a right handed coordinate system.
We do the stitching by constructing lookup tables that, for each pixel in the panorama, point to which pinhole pixel to sample from. This also takes advantage of the fact that the pinhole images will be fed to the sampler concatenated together like \([\text{back}]||\text{left}||\text{front}||\text{right}\) for cylindrical stitching or \([\text{back}]||\text{left}||\text{front}||\text{right}||\text{top}||\text{bottom}\) for spherical stitching. The bilinear sampling method will take these source coordinates and sample resulting panoramas. Our sampling method uses Tensorflow [1] to let it run on the GPU and can process batches. Despite this, stitching is still quite time intensive.

To construct the needed lookup tables, we iterate over the panorama and calculate the source pixel locations. Instead of iterating over \(i, j\) pixel locations, we iterate over the coordinate system of the panorama. For a cylindrical panorama, this means we iterate over \(\theta\) and \(h\), while a spherical panorama uses \(\theta\) and \(\phi\). For both panorama types, we first project these coordinates into 3D space, then project them into pinhole space and figure out which pinhole image we should look at (this translates to an \(x\) offset because of the image concatenation).

For cylindrical panoramas, to translate \(\theta\) and \(h\) into 3D space, we use:

\[
\begin{align*}
X &= \sin(\theta) \\
Y &= h \\
Z &= \cos(\theta)
\end{align*}
\] (3.1)

Theta is adjusted beforehand so that the resulting 3D coords will be relative to the correct side, which is also calculated at that point. The pinhole coordinates are then:

\[
\begin{align*}
x &= f \frac{X}{Z} + C_x + \text{offset} \times \text{width} \\
y &= f \frac{Y}{Z} + C_y
\end{align*}
\] (3.2)
where \( f = \frac{\text{width}}{2 \tan(\frac{\text{fov}}{2})} \) is the focal length, and \( C_x = \frac{\text{width}}{2} \) and \( C_y = \frac{\text{height}}{2} \) are the center \( x \) and \( y \) coordinates, respectively. \( offset \) is the index of the pinhole image to use, from the concatenation order.

The formulas for spherical panoramas are very similar, except we iterate over \( \theta \) and \( \phi \):

\[
\begin{align*}
X &= \sin(\theta) \cos(\phi) \\
Y &= \sin(\phi) \\
Z &= \cos(\theta) \cos(\phi)
\end{align*}
\]

These coordinates are then adjusted to be relative to the correct side. Unlike for cylindrical, this is done after converting to 3D coordinates, as all that needs to be done to the 3D coordinates is reassignment and negating. The coordinates are converted into pinhole coordinates using the same formula as used for cylindrical stitching.

Because depth is measured as the distance from the camera’s plane, adjustments must be made to the depth values (the camera planes for pinhole images would make a cube, while the plane for a cylindrical panorama is a cylinder, etc.). For cylindrical panoramas this can be done by dividing depth by \( \cos(\theta) \), and for spherical panoramas it is done by dividing by \( \cos(\theta) \cos(\phi) \), where all values are the side-relative values.

For cylindrical stitching, these values are easily calculated at stitch-time, but for spherical stitching it is easier to save them with the look up table. An example of the stitched cylindrical color and disparity (converted from the stitched depth) is shown in Figure 3.2, and spherical versions of the same in Figure 3.3. Note that depths appear different because of normalization (more data, such as the ground immediately around our car, is included in the spherical panorama).
intrinsics are included in the data access package, but if you don’t have access to it, they are shown in Table 3.1.

To ensure our stitching algorithms were correct, we created and exported point clouds as mesh files (.ply format) and used a free mesh viewer to inspect them. A screenshot is shown in Figure 3.4. Of course, everything is seen from the car’s point of view, so as you can see objects are streaked around their edges, but all the angles are correct: roads and buildings make right angle turns, buildings go straight up, etc. This is not the case if our depth multipliers when stitching are incorrect, these angles would
Figure 3.3: A sample stitched spherical disparity and color image from CARLA.
Table 3.1: CARLA camera intrinsics. Note that cubemap panoramas use Pinhole 90° images for each side. The horizontal and vertical coordinates are listed in the first row (in that order). The image size is width \times height in pixels.

<table>
<thead>
<tr>
<th>Camera Coords</th>
<th>Pinhole (100°)</th>
<th>Pinhole (90°)</th>
<th>Cylindrical</th>
<th>Spherical</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_x )</td>
<td>322.214</td>
<td>384.000</td>
<td>325.949</td>
<td>325.949</td>
</tr>
<tr>
<td>( f_y )</td>
<td>322.214</td>
<td>384.000</td>
<td>1023.000</td>
<td>325.949</td>
</tr>
<tr>
<td>( c_x )</td>
<td>384.000</td>
<td>384.000</td>
<td>1024.000</td>
<td>1024.000</td>
</tr>
<tr>
<td>( c_y )</td>
<td>384.000</td>
<td>384.000</td>
<td>511.500</td>
<td>512.000</td>
</tr>
<tr>
<td>Image Size</td>
<td>768 \times 768</td>
<td>768 \times 768</td>
<td>2048 \times 1024</td>
<td>2048 \times 1024</td>
</tr>
</tbody>
</table>

be significantly off. For example, if we use \( \cos(\theta) \) instead of \( \cos(\theta) \cos(\phi) \), we would see cosine-like waves in vertical surfaces like buildings. These meshes allowed us to validate that our depths were actually correct.

Figure 3.4: Rendering of the exported mesh created from our spherical stitched data.

3.3 Hosting

The code used to run the simulations is available at https://github.com/rnett/CARLASim.
3.3.1 Raw Data

The dataset is stored in hdf5 files and hosted in an AWS S3 bucket at https://cscdatasets.s3-us-west-2.amazonaws.com/jventu09/carla_dataset/. The directory structure is `{town}/{time}/cars_{num_cars}_peds_{num_peds}_index_{i}` with `cylindrical.hdf5`, `spherical.hdf5`, `pinhole.hdf5`, and `pose.hdf5` in that directory. `cylindrical.hdf5` and `spherical.hdf5` contain “rgb” and “depth” datasets. `pinhole.hdf5` contains a group for each side, with each group containing “rgb” and “depth” datasets. All image datasets have 1000 images, with the sizes shown in Table 3.1, and the format [batch, height, width, channels]. They are stored as `numpy.ndarray`s. Color images have three channels (RGB), while depth images have one. Depth images are also `uint16`, with values in decimeters, while color images are the standard `uint8`. For example, the cylindrical color dataset would have shape [1000, 1024, 2048, 3] and the depth dataset would have shape [1000, 1024, 2048, 1]. `pose.hdf5` contains ”absolute_pose”, ”relative_pose”, and ”start_relative_pose” datasets, which are all shape [1000, 6] and contain the absolute pose (relative to CARLA’s origin), relative pose (to the last frame), and the pose relative to the initial pose, respectively. All of these pose values are [X, Y, Z, x, y, z] where X, Y, and Z are the location in meters (possibly relative, depending on the dataset accessed) and x, y, and z are the components of the unit heading vector. A `seed.txt` file also contains the random seed used for the simulation, for reproduction purposes.

3.3.2 Python Package

Accessing the raw data is a pain, so a python package for downloading and accessing data is provided. It is `cpdd-dataset` on PiPy, and the source code is on GitHub.
at https://github.com/rnett/cpdd-dataset. The readme explains the function of the package, but an overview is provided here.

The central idea of the package is the Config class, which represents a single simulation. From a Config, the remote and local locations for data files can be found, and the cylindrical, spherical, pinhole, and pose DataFiles can be downloaded and accessed. Each DataFile can be checked to see if it is downloaded, downloaded, and the actual data can be accessed. The pinhole DataFile also provides side accesses to only get the data for a single side (although all sides are downloaded in the same file).

Additionally, methods to set the download location are provided, or an environment variable can be used (which is the recommended method). The camera intrinsics for the image types are also provided and are accessible from each data file. The package also provides methods to fill in wildcards with valid values, and methods to read configs (with wildcards) from CSV or text files. This makes using the dataset as simple as:

```python
import carla_dataset

train_configs = carla_dataset.config.load_csv("train_data.csv")
train_data = []

for config in train_configs:
    with config.cylindrical_data.download() as data:
        train_data.append({"color": data.color, "depth": data.depth})
```

The structure of the DataFiles also makes it easy to load the data later, or to use them with a PyTorch DataSet as we have done.
Chapter 4

BASELINE EXPERIMENT

After our creation of the dataset, but before our experiments on monodepth2, we wanted to get a baseline to hopefully improve on with monodepth2 and validate that our dataset was usable for depth estimation. To do that that, we adapted CylindricalSfmLearner from Sharma and Ventura [15] to support our dataset and ran experiments with it on our cylindrical data. CylindricalSfmLearner is a network for cylindrical panorama depth estimation, based on SfmLearner, which is covered in 2.1. As we mention in 2.2.1, it pioneers the wrap padding technique that we later add to monodepth2. In Sharma and Ventura [15], the authors were unable to find a real dataset with cylindrical images and ground truth depth or enough data to stitch them, so they used SYNTHIA-Seqs, a synthetic dataset containing images and ground truth depth from front, left, right, and backwards facing cameras [11]. This provides enough data to stitch cylindrical panoramas but is not enough to make spherical or cubemap panoramas. They also created and published a real-world cylindrical dataset called Headcam and evaluated CylindricalSfmLearner on it. Our experiments ended up finding several issues with our dataset that we fixed, and provided additional validation for CylindricalSfmLearner, which we used in Sharma, Nett, and Ventura [13].

Table 4.1: Results from Sharma, Nett, and Ventura [13] from using CylindricalSfmLearner on carla data, with and without wrap padding (with labels fixed). Units are in meters.

<table>
<thead>
<tr>
<th></th>
<th>Abs Rel</th>
<th>Sq Rel</th>
<th>RMSE</th>
<th>RMSE log</th>
<th>δ &lt; 1.25</th>
<th>δ &lt; 1.25²</th>
<th>δ &lt; 1.25³</th>
</tr>
</thead>
<tbody>
<tr>
<td>No wrapping</td>
<td>0.550</td>
<td>16.266</td>
<td>12.868</td>
<td>0.560</td>
<td>0.475</td>
<td>0.745</td>
<td>0.856</td>
</tr>
<tr>
<td>Wrapping</td>
<td>0.497</td>
<td>14.212</td>
<td>12.032</td>
<td>0.528</td>
<td>0.565</td>
<td>0.781</td>
<td>0.867</td>
</tr>
</tbody>
</table>
Our CARLA dataset is superior to SYNTHIA-Seqs in a couple of ways for cylindrical data: it has multiple traversals of the same city (SYNTHIA has multiple sequences from the same city, but they all follow the same path) and it has varied cities that are all somewhat similar. As a result of both of these, we could use a never-before seen city as the test set. We follow our advice in chapter 3 and use city 2 for the test set. Training and testing was done with stationary sequences removed, using the same evaluation methods as used on SYNTHIA and Headcam. The results from evaluating CylindricalSfmLearner on our CARLA dataset are shown in Table 4.1. Wrap padding is slightly but noticeably superior, and overall, the results are fairly good, especially for a never-before seen city. They also provide some context for judging our monodepth2 cylindrical results.

A sample of the estimated depth from using CylindricalSfmLearner on our CARLA dataset is shown in Figure 4.1. The depth maps were fairly good but have some severe holes in the street. A large part of the inaccuracy in the metrics comes from this. We suspect this is because the street and sky colors are so similar. We hoped that using monodepth2 for our base network instead of SfmLearner in this work would reduce those holes, or perhaps that they would be less prevalent in some panorama types. These results also validate the dataset itself, and help show that even if monodepth2 has issues with our dataset, it is most likely a problem with monodepth2, not our data.
Figure 4.1: A sample of estimated depth from a CARLA image, produced for and first appearing in Sharma, Nett, and Ventura [13]
To further validate our dataset and answer the original question of which type of panorama works best, we used monodepth2 [4] with additions to support cylindrical and cubemap data. We want to optimize its hyperparameters to give a decent level of performance on pinhole data, and then compare that to its performance on cylindrical and cubemap panoramas. We want the configuration to be as close as possible for each format, so that we are testing how well the same model applies to each format, rather than how much performance we can squeeze out of each. To be able to use it, we have to modify monodepth2 to support our panoramas. The modifications we made are described in this section (as well as an overview of how monodepth2 works), how they are implemented is described in the next. Similarly to how we are treating hyperparameters, we do not want to make major modifications to these methods, our objective is to test them mostly as-is.

### 5.1 Pinhole

For pinhole images, we use monodepth2 without any major alterations. A few implementation details were changed and are described in the next chapter. Monodepth2, as described in section 2.1, is a self-supervised model, which means that it trains without training labels. It does this by taking as input multiple sequential frames, predicting the pose differences between them, and using the poses and the predicted depth to reproject the target frame to the other frames’ points of view. If the predicted
depth is accurate, then the reprojections should closely match the actual images. The difference between the actual and reprojected images is used as the primary loss.

To predict both the pose differences and the depth, monodepth2 uses resnet encoders with custom decoders. The pose encoder and decoder take two frames and produce the difference in pose between them. The depth encoder and decoder take single images and predict the disparity for each, at each of the preset scales. Of course, both are batched. Monodepth also uses multi-scale depth estimation to help with training, as described in section 2.1. This is handled in the depth decoder and described in the Implementation chapter. Instead of comparing the lower resolution images to reprojections at that resolution, they upsample the lower resolution images to the original size and then do the comparison.

Monodepth2 also uses more complicated loss functions than simply comparing the images. It does do a per-pixel comparison of the images, called photometric loss, using SSIM and L1. It also adds an edge-aware smoothness loss on the depth map. To help deal with occlusions between the frames, it uses the minimum photometric loss for each pixel between the two comparison frames. These image losses are calculated for each scale as mentioned earlier. The multiple scales help prevent gradient issues with the reprojection sampling, and the upscaling helps prevent artifacts from the sub-scaled images. The formulas of the loss functions are shown in section 2.1. A good overview of monodepth2’s architecture is the image provided in the paper: Figure 5.1.
5.2 Cylindrical

To alter monodepth2 to work with cylindrical data, we had to modify the data loader and reprojection, and also added cylindrical padding as described in subsection 2.2.1. The data loader and reprojection both do the same thing as the pinhole versions but with cylindrical data, and thus our modifications are covered in the Implementation chapter. We do cylindrical padding by copying the left side of the panorama to the right, and vice versa, since they are of the same spatial location. This is shown in Figure 5.2 and is drawn from Sharma and Ventura [15]. We do this in all convolution layers, so the padding propagates and eventually includes representations of larger areas of the other side. The padding done here is the standard convolutional padding, the size of the images is not changed.

5.3 Cubemap

The way we handled cubic input data is described in detail in subsection 6.2.2, but from a design perspective all we need to know is that each side is handled indepen-
dently as far as pose and depth estimation. The pose results are merged to give a single pose change (and a consistency loss) and the results depth are concatenated together for reprojection. We also used the methods described in subsection 2.2.2, from Cheng et al. [2] and Wang et al. [17]: cube padding, pose averaging and consistency loss, and whole view reprojection. Cube padding is conceptually similar to cylindrical padding, but instead of padding with the opposite side, we pad with strips of the adjacent sides oriented so that the match the image being padded. This is shown in Figure 5.3. For example, the top side would be padded on the bottom with the top of the front side. Like cylindrical padding, this is done as normal convolutional padding, and doesn’t change the image size. Pose averaging is how the poses from each side are merged: by taking the average. We leave the top and bottom frames out of this average since there are few if any features to detect pose from. Pose consistency loss is closely related: it is the variance of the predicted poses, summed across the pose matrices. We use variance instead of the standard deviation used by Wang et al. [17] because the standard deviation caused gradient issues. It adds a $\lambda_{\text{pose}}L_{\text{pose}}$ factor to the loss, where $\lambda_{\text{pose}}$ is a hyperparameter and $L_{\text{pose}}$ is the pose consistency loss. Whole view reprojection means we use the entire view, i.e. all the sides, when doing reprojection. Each side’s reprojection is aware of the entire image and can sample pixels from the other sides as if we were working with spherical panoramas. How this is implemented is described in subsection 6.2.2.
Figure 5.3: Cube padding from Cheng et al. [2]. Image is from the paper’s associated GitHub repository. This is done to enforce consistency around the edges of cubemap panoramas.
6.1 Monodepth2

Here we give a brief overview of the implementation of monodepth2 [4]. The theory behind the model is described in section 2.1, and a high level overview is given in chapter 5. The model and our modifications are implemented in PyTorch [9].

The monodepth2 model takes as input 3-image sequences: a “before” frame, the target frame, and an “after” frame. It predicts the depth of the target frame and the poses between the target frame and the other frames. It then reprojects the target frame to the before and after frames’ (detected) locations and uses that to calculate the reprojection loss. As described earlier, Monodepth2 uses multiple scales of the images for depth estimation to help with training. This is only relevant to the depth decoder.

The monodepth2 model has 6 main parts:

- Pose encoder
- Depth encoder
- Pose decoder
- Depth decoder
- Reprojection
- Loss
Both depth and pose encoders are resnet18 models using ImageNet pretrained weights by default. The pose network is modified slightly to take two images as input by stacking their channels, in this way it is able to return features representing both images.

The pose decoder calculates the difference in pose between the two images (although there is no definition of the different images in the input, just the encoded data) as axis-angle and translation. The network uses several convolutions with ReLU activations and scales the output by 0.01.

The depth decoder is composed of 4 3x3 convolutions, each followed by an ELU and a 2x upsampling. Skip connections are also used. Monodepth2 also uses multi-scale output, so at each upsampling step that is wanted as output, the intermediate value is fed through an additional 3x3 convolution sigmoid activation and then saved as output (note that the intermediate value that is used by the next upsampling is unchanged). The disparity is calculated here and is converted to depth later.

The reprojection phase takes the depthmap, color image (of the target frame), and pose difference (as a translation matrix) and produces a color image that is what the scene would look like if it had been viewed from the given pose. Since the pose is the pose difference between two known frames, the produced image can be compared to the actual image to calculate the loss.

The actual reprojection is done in four phases: turn the depthmap into a point cloud, translate the point cloud according to the pose, turn the point cloud into a pixel mapping, and sample the color image according to the mapping. These are all fairly standard computer vision techniques, but we will give a summary here as it gives context to the modifications we made. Turning the depthmap into a point cloud is done by matrix multiplying a meshgrid of the image coordinates by the inverse of
the camera intrinsics to get the unit world coordinates, then multiplying those by the depth map. Translating the point could be by the difference in poses just the application of the translation matrix by matrix multiplication. Turning the point cloud into a pixel mapping is done by converting the world $X$ and $Y$ coordinates to camera $x$ and $y$ coordinates by dividing the them by the world $Z$, and then converting the camera coordinates to image coordinates by padding to 3x3 with ones and matrix multiplying by the camera intrinsics. The sampling is done using PyTorch’s bilinear sampler.

The loss functions are described in detail in the related work and analysis chapters, and in the monodepth2 paper. We are not modifying them except to add the cubemap pose consistency loss. All we need to know for the implementation is that the loss is calculated at each scale and averaged across them.

6.2 Modifications

To support the CARLA dataset, and to support different types of panoramas, a number of modifications had to be made to the base monodepth2 implementation. To work with the CARLA dataset, monodepth2’s dataset loader had to be adapted to CARLA. This was fairly easy to do using our cpdd-dataset python library, and most of the data loader was able to be copied from monodepth2’s existing loaders. Loading cubemap images is more complicated, and is detailed in its section. The machine we used to train these models has 4 Tesla V100s, and to take full advantage of them we parallelized the image processing sub-models (depth and pose encoders and decoders). This was done using PyTorch’s DataParallel and was did not require any large changes, although some modules had to be restructured to make their parameters and sub-modules detectable by PyTorch.
6.2.1 Cylindrical

Cylindrical images are input as large single images. This means that most of the networks are fine as-is. Reprojection support had to be added and we had to change the base convolution layer to one that supports the cylindrical padding we are using. As described earlier, we are using cylindrical wrapping padding. This was implemented as a custom convolutional layer extending `torch.nn.Conv2d`. We parameterized the modules that use convolutions with a parameter for the convolution layer to use, by passing the python class. Since most of the convolution heavy modules use pretrained resnets, we also had to copy and parameterize PyTorch’s resnet implementation. Our custom padding layer uses the same weight shapes as a normal `Conv2d` layer, so there were no issues with weight loading. The reprojection modules take the intrinsics as parameters, so we can simply pass them the cylindrical intrinsics. However, when using the cylindrical intrinsics in the depth to point cloud step, the point cloud is given in cylindrical coordinates, which don’t work with the pose translation matrix, so we have to convert them to Cartesian coordinates, do the translation, and then convert them back. We actually don’t convert them back to cylindrical world coordinates, instead we convert them directly to cylindrical image coordinates, which are then converted to normal image coordinates using the intrinsics.

6.2.2 Cubemap

Cubemap panoramas have several complicating factors that go beyond what was required for cylindrical panoramas. We can’t treat them as a single image, the padding depends on which side’s image you are padding (and on the other images), and they have their own loss from pose consistency. Other than calculating the pose consistency loss and the averaged pose, doing the padding, and doing reprojection, each side can
be treated as its own separate image. Because of this, we pack the different sides into
the batch dimension, in a defined order so that we can pull them out. For example, a
batch would look like [top$_1$, bottom$_1$, left$_1$, right$_1$, front$_1$, back$_1$, top$_2$, bottom$_2$, ...]. We
can then use helper functions to pull the sides out when needed. Because of the way
PyTorch’s data loaders return single examples and not batches, we don’t pack the
images in the loader, but instead return each side as a feature. These features are
batched automatically by PyTorch, giving us a batch for each side. When we get the
batches, we interleave the sides to get the format shown above. To pad a given side,
we need to take strips from adjacent sides, rotate them to the correct alignment, and
then concatenate them. We do this manually for each side in a custom convolution
layer like with cylindrical padding. This lets us use the same parameterization we
had set up for cylindrical padding. However, it requires pulling each side out of the
interleaved batch. Most of this is done using indexing and reshaping, so anecdotally
the performance cost isn’t very large.

To implement the pose averaging and consistency loss, we had to add a module that
is applied to the calculated poses. The module extracts the sides from the batch,
changes their basis (using rotation matrices) to match the front side, calculates the
average (to use as the pose) and the variance (the loss), and then calculates the pose
for each side by changing the basis of the average pose back to the side. We leave
the top and bottom poses out of the average (but use the average as their poses)
since there are very few features that would enable pose to be estimated from those
images. This is particular to self driving cars (which is what our dataset is); if you
were dealing with a drone, you would at least want to include the bottom. Pose
calculations aren’t done per-scale, so we add this loss after the per scale averaging.
It is scaled by a hyperparameter $\lambda_{\text{pose}}$. 

34
We also had to adapt reprojection to treat each set of side images as a single image. It would be easy enough to reproject each side individually, but we wanted the sides to be able to draw from each other, as if they were a complete panorama. The key to the way we handle this is that each side is transformed identically from world to image coordinates, in the same manner as pinhole images. If we took some point value and moved it from the front side tensor to the right side tensor, and then processed it, the result would be exactly the same as if we hadn’t moved it (note that we moved the exact value, without any translation to account for the PoV change). We concatenate all of the sides horizontally to handle sampling all at once, so to target a particular side you just adjust the horizontal value of the pixel mapping. When processing a side, we can detect when a point would be out of bounds of that side's image by checking that the largest component of the world coordinate is in the same direction as the side’s normal vector, and find the side the point should be sampled from using the same method. Once we know the source side of an out of bounds point (the side we will need to sample from), we can find what the world coordinates of this point would be from the source side’s pov. Since all sides are processed identically, instead of processing this point as part of the source side, we can replace the coordinates in this side with the coordinates from the pov of the source side, and add an offset after processing to ensure it points to the correct side.

Actually doing this requires converting all of the side’s point clouds to the same coordinate system (we use the forward camera’s) so that we can tell from which side each point is visible from. Thankfully, both the coordinate system swaps and the visibility detection can be done in tensor operations, so we never have to iterate over the tensors.
Chapter 7

RESULTS

7.1 Evaluation

Each model was trained for 20 epochs on cities 1, 3, 4, and 5, with city 2 being the training set and runs 0 and 1 of town 1 being used for validation (and thus removed from the training set). Additionally, the $\lambda_{pose}$ hyperparameter was optimized for the cubemap models, with the best value being 0.01. We experimented with the pinhole model to find decent hyperparameters for the carla dataset, and then used those hyperparameters for the panoramic models. For cubemap, since each batch has six times as many images as pinhole or cylindrical (from the sides), we reduced the number of epochs to 4. While this gives us a bit more than 20 equivalent epochs, it worked best with the learning rate scheduler. We found that using the imagenet pretrained weights did not work well on our simulated images. To account for training the encoders from scratch and for having less training data (18,000 as opposed to KITTI’s 40,000), we doubled the learning rate to 0.0002. We did not remove stationary sequences from the training dataset as done in monodepth2 at first. Our reasoning for this is that monodepth2 has special handling for stationary objects within images, and we wanted to see if this applied to entire images, and if it applied differently to different formats. We ended up adding it back for pinhole and cubemap images in later evaluation runs. All of these runs were with the batch size that would roughly fill a single Nvidia V100 GPU, which has 32 GB of RAM. This was 10 for pinhole, 2 for cylindrical, and 1 for cubemap.
We evaluate the depth and pose predictions using the same method as monodepth2, which gives us absolute relative error, squared relative error, RMSE and log RMSE, and percentile errors (the % of errors less than 1.25, 1.25², and 1.25³). Percentile errors are best when they are high, but we want the other five to be as low as possible. All of our metrics are in meters. We will also consider factors such as training time and memory requirements.

### 7.2 Quantitative Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Abs Rel</th>
<th>Sq Rel</th>
<th>RMSE</th>
<th>RMSE log</th>
<th>δ &lt; 1.25</th>
<th>δ &lt; 1.25²</th>
<th>δ &lt; 1.25³</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monodepth2 (pinhole) [4]</td>
<td>0.115</td>
<td>0.882</td>
<td>4.701</td>
<td>0.190</td>
<td>0.879</td>
<td>0.961</td>
<td>0.982</td>
</tr>
<tr>
<td>CylindricalSfmLearner (cylindrical) [13]</td>
<td>0.497</td>
<td>14.212</td>
<td>12.032</td>
<td>0.528</td>
<td>0.565</td>
<td>0.781</td>
<td>0.867</td>
</tr>
<tr>
<td>Pinhole</td>
<td>1.208</td>
<td>14.903</td>
<td>10.761</td>
<td>0.954</td>
<td>0.964</td>
<td>0.196</td>
<td>0.430</td>
</tr>
<tr>
<td>Cylindrical</td>
<td>0.338</td>
<td>6.995</td>
<td>8.502</td>
<td>0.374</td>
<td>0.671</td>
<td>0.874</td>
<td>0.940</td>
</tr>
<tr>
<td>Cubemap</td>
<td>0.609</td>
<td>4.348</td>
<td>8.490</td>
<td>0.720</td>
<td>0.346</td>
<td>0.500</td>
<td>0.644</td>
</tr>
</tbody>
</table>

Our initial results are shown in Table 7.1. Cylindrical does quite well, cubemap and pinhole are decent, but poor compared to monodepth2’s KITTI results. The poor results for pinhole was surprising, as monodepth2 is originally a pinhole model. To ensure that we hadn’t made some change that was causing this, we went back to the original model and ran it with our data. This didn’t cause any improvements, and since both monodepth2 and CylindricalSfmLearner work with our data, we think this is a problem with monodepth2’s interactions with our dataset, not an issue with our data.
7.3 Qualitative Results

Cylindrical and pinhole (front only) ground truth and predicted disparity images are shown in Figures 7.1 and 7.2, respectively. The cylindrical prediction is fairly good. It has some artifacts, and some parts of the visual image (i.e. text on the sign) show up in the depthmap, but everything is there and at reasonable depths. There are blobs behind edges against the sky, which are most visible around the streetlights. The pinhole image also looks fairly good if you ignore the blobs. The depth of objects

\footnote{See Qualitative Results section for a discussion of cubemap results. While these results look decent, they are not valid.}
is clearly being detected, and fairly well at that. The blobs may be an artifact of
the CARLA dataset, as similar blobs were seen in Sharma, Nett, and Ventura [13].
However, they are mostly not present in the cylindrical predicted images. Overall,
the cylindrical images are clearly the best.

You might notice that the cubemap images aren’t shown. This is because the pre-
dictions are uniformly max depth. This isn’t due to NaNs or anything like that, the
network trains properly and the weights are reasonable values, it just gets stuck at a
local minima and learns to output max depth. This behavior was also seen, somewhat
randomly, on pinhole models trained with small batch sizes. In one case, we saw this
when training a model for 40 epochs, but not when training with the same hyperpa-
rameters but for 20 or 60 epochs. Note that even with this issue, the cubemap model
shows decent evaluation results because of the way monodepth2’s evaluation code
handles minimum and maximum depth values. To be clear, the evaluation results
are not valid. We suspect this issue is in part because a few cities have images with
lots of sky, where the sky is nearly the same color as the roads. It is conceivable that
the network would be confused by everything being mostly the same color. We also
think part of the issue (and it’s prevalence in cubemap models) is caused by cube-
map using top and bottom images, which means a third of the training data for the
depth and pose encoders and decoders is just sky or road. Unfortunately, processing
all sides is necessary for reprojection and cube padding, so modifying the network
to leave them out is outside of the scope of this work. It would also be application
dependant. Increasing the batch size seems to help with this, likely because you have
some confusing images and some normal images being averaged out before weights are
changed. We were able to slightly mitigate this issue, as covered in the next section.
7.4 Improvements

While the original premise for this work was to simply apply a near state of the art model to panoramic images, using the best practices for handling each format, the lack of good results for pinhole or cubemap models prompted us to try to make some small improvements. Our results are shown in Table 7.2. We had some success with pinhole models using a batch size of 1 and removing stationary frame sequences, shown in the table as ”Pinhole (best settings)”. This was also with the learning rate reset to monodepth2’s default of 1e-4. These results are significantly better than our previous results but are still very inferior to monodepth2’s original performance on KITTI. Additionally, when training with a batch size of one, the model is more prone to falling into the “everything is sky” local minima. Training also takes much longer.

<table>
<thead>
<tr>
<th>Model</th>
<th>Abs Rel</th>
<th>Sq Rel</th>
<th>RMSE</th>
<th>RMSE log</th>
<th>δ &lt; 1.25</th>
<th>δ &lt; 1.25²</th>
<th>δ &lt; 1.25³</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monodepth2 (pinhole) [4]</td>
<td>0.115</td>
<td>0.882</td>
<td>4.701</td>
<td>0.190</td>
<td>0.879</td>
<td>0.961</td>
<td>0.982</td>
</tr>
<tr>
<td>Pinhole (original)</td>
<td>1.208</td>
<td>14.903</td>
<td>10.761</td>
<td>0.954</td>
<td>0.064</td>
<td>0.196</td>
<td>0.430</td>
</tr>
<tr>
<td>Pinhole (best settings)</td>
<td>0.768</td>
<td>8.802</td>
<td>10.449</td>
<td>0.760</td>
<td>0.188</td>
<td>0.407</td>
<td>0.611</td>
</tr>
</tbody>
</table>

To attempt to solve cubemap’s issues, we removed stationary frame sequences and used a batch size of 2 (this doesn’t fit on one GPU but was doable using our parallelization modifications) and a learning rate of 1e-4. This didn’t solve the issue, but we were able to see the network gradually devolve throughout training. Predicted depth for the same image at epochs 4 and 6 is shown in Figure 7.3. Note how the depth gets further away and somewhat more uniform. Objects are clearly differentiated, and some, like the trees and the car, are predicted to be closer than the background. However, objects that are close in color to the sky, like the road and sidewalk, are predicted to be really far away. The overall image is also shifted further away. We attribute this mostly to the inclusion of top and bottom frames, since depth prediction is done on the same network for all sides.
7.5 Resource Requirements

Table 7.3: Training time with largest batch size that fit, and approximate GPU memory required for a single example, for each panorama type.

<table>
<thead>
<tr>
<th>Panorama Type</th>
<th>Training Time</th>
<th>GPU Memory Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pinhole</td>
<td>24h</td>
<td>2.9 GB</td>
</tr>
<tr>
<td>Cylindrical</td>
<td>74h</td>
<td>12 GB</td>
</tr>
<tr>
<td>Cubemap</td>
<td>175h</td>
<td>21 GB</td>
</tr>
</tbody>
</table>

As shown in Table 7.3, pinhole models are by far the cheapest to train. Cylindrical models take about 3 times as long and need about 4 times as much memory, while cubemap models take about 7 times as long and need a whopping 7 times as much memory. Keep in mind that the training time is with enough batches to fill a single V100 GPU (30 GB of memory), while the memory requirements are for a single batch. Even so, the time and memory both roughly line up with the difference in pixels: the cylindrical panoramas have 3.5 times as many pixels as a pinhole image, and cubemaps have 6 times as many.
In this thesis, we have created a synthetic dataset that lends itself well to panoramas and includes cylindrical and spherical panoramas. We ran initial experiments using CylindricalSfmLearner from Sharma and Ventura [15] to validate our data and provide additional validation for CylindricalSfmLearner, and published our results in Sharma, Nett, and Ventura [13]. We modified monodepth2 from Godard et al. [4] to support cylindrical and cubemap panoramas and used it to evaluate each type of panorama on our dataset. The results we got show that cylindrical panoramas perform significantly better than pinhole images, while cubemap panoramas had training issues.

Our dataset is publicly available in an AWS S3 bucket at https://cscdatasets.s3-us-west-2.amazonaws.com/jventu09/carla_dataset/, structured as described in the Dataset chapter. There is an associated python package cpdd-dataset on PiPy with methods to download and load it from Python. Our code is also available, as well. The model code is located at https://github.com/rnett/monodepth2 and is a fork of monodepth2, and the simulation code is located at https://github.com/rnett/CARLASim.

While the poor performance of our pinhole models in contrast to monodepth2’s original excellent performance on pinhole data does raise questions about whether our cylindrical model is truly better, we have conducted a thorough enough set of hyperparameter tests using the original monodepth2 model with our data that we are certain that this is an issue with monodepth2, not with our model. This doesn’t entirely validate our results: it is possible that when using a model that does not have these issues, pinhole images would perform better in comparison to cylindrical.
However, it doesn’t invalidate our results either, and cylindrical panoramas being so much easier to work with is a valuable result in and of itself. Our cubemap results were also rendered void by an issue with batch sizes and sky and road images, as described in the Qualitative Results section of the Results chapter. Overall, our main evaluation result is that cylindrical images are significantly more resilient to model quirks and perform better overall. With monodepth2, they are also able to be used without removing stationary frame sequences.

8.1 Future Work

There are several clear opportunities for future works to build on this one. The most obvious is to use a different base network, perhaps SfmLearner, in the hopes getting good pinhole performance on our carla data. It is possible that there is something with our dataset that makes pinhole depth perception much harder than other commonly used datasets, but this is very unlikely, considering cylindrical works for monodepth2 and for CylindricalSfmLearner. Another compelling option is to attempt to solve the cubemap training issues. There are several ways we would recommend to start. One is batching losses, even if the examples aren’t batched. This prevents the network from going down a hole to a local minima, so to speak. Randomizing the samples may also help. Another is to change the network architecture so that instead of applying the same encoders and decoders to each side, each side has its own. Using one for the top and bottom and one for the sides could also be done, or some other method to still include the top and bottom in padding and reprojection, but not in training.


