CIS 580, Machine Perception, Fall 2022
Homework 5
Due: Mon Dec 12th 2022, 11:59pm ET

In this homework, you are going to implement two-view stereo and multi-view stereo algorithms to convert multiple 2D viewpoints into a 3D reconstruction of the scene.

The main code is implemented in two separate interactive jupyter notebooks, `two_view.ipynb` and `plane_sweep.ipynb` which import several functions from `two_view_stereo.py` and `plane_sweep_stereo.py` which you are responsible for.

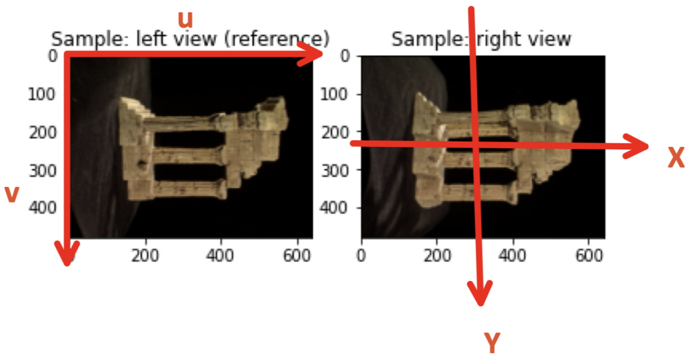
The prerequisite python libraries you will need to install to run the code are included in `requirements.txt` which if you are a pip user, you can install via `$ pip install -r requirements.txt`. Note that for the K3D library, which is used to visualize the 3D pointclouds in the jupyter notebook, there are some additional steps after pip installation (If you are using VS-Code, VS-Code will automatically help you to handle this). Namely, you may run the following lines after installation to explicitly enable the extension:

```
$ jupyter nbextension install --py --user k3d
$ jupyter nbextension enable --py --user k3d
```

You need to submit your code to our auto-grader in gradscope and also submit your final homework report answering the questions specified in the rest of this handout.

Note that in the readme file, we provide some common mistakes and hint to help you better debugging.

In `two_view.ipynb` notebook, we first would like to get an understanding of the dataset we are working with and visualize the images:



The row index of the image corresponding to pixel coordinate v and the horizontal direction Y of the camera frame. The column index of the image corresponding to pixel coordinate u and the vertical direction X.

To further help your understanding of the scene, you can uncomment the function `viz_camera_poses([view_i, view_j])` to interactively visualize the coordinate frames of the viewpoints. You can press `[i]` to show the world coordinate frame.

1. **Two View Stereo (65pts)**

   We use `view_i` as left view and `view_j` as right view.

   (a) (18pts) Rectify Two view.

   i. (2pts) Understand the camera configuration. We use the following convention: $p^i = R^i_w p^w + T^i_w$ to transform coordinates in world frame to camera frame. In the code, we use `R_wi` to denote R^i_w. You will need to compute the right to left transformation R^i_j, T^i_j and the baseline B. Complete the function `compute_right2left_transformation`

   ii. (8pts) Compute the rectification rotation matrix R^{rect}_i to transform the coordinates in the left view to the rectified coordinate frame of left view. Complete `compute_rectification_R`. Important note: You can find the derivation for rectification in the two-view stereo slides (Lec22-slides.pdf), but remember that the images in our dataset are rotated clockwise and thus the epipoles should not be placed at x-infinity but instead at y-infinity. **Hint**: move the epipole to the y-infinity using: $[0, 1, 0]^T = R^{rect}_i e_i$

   iii. (8pts) We implemented half of the two view calibration for you after getting R^{rect}_i. Complete the function `rectify_2view` by first computing the homography and then using `cv2.warpPerspective` to warp the image. When warping the image, use the target shape we computed for you as `dsize=(w_max, h_max)`. **Hint**:

   $$K^{-1}_{corr} \begin{bmatrix} u_{rect} \\ v_{rect} \\ 1 \end{bmatrix} = R^{rect}_i K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

   (b) (32pts) Compute Disparity Map

   i. (3pts) Before you implement the full function, we provide an example for you to understand. Unlike the one in our slides, our image centers are different on Y direction (pixel v direction) across the different images. Given K_i, K_j with the same image center on X direction $c_{xj} = c_{xi}$ but different center on Y direction c_{yj}, c_{yi}, we denote $d_0 = c_{yj} - c_{yi}$, then our disparity is $d = d_0 + v_L - v_R$. Briefly explain $d = d_0 + v_L - v_R$ and derive the stereo equation with d_0, v_L, v_R, f_y, B, Z from page 42 in the two view stereo slides (Lec21-slides.pdf) or page 4 in Lec22-slides.pdf, assuming that the left and right view have the same focal length f_y. **Add the answer to your report.**

   ii. (5pts) We are going to compare the patch, complete the function `image2patch` using zero padding on the border (the padding means when you extract the patch of some pixel on or near the border of the image, you may find missing positions in the patch, then you can fill in the missing pixels as zeros.). This function should take an image with shape $[H, W, 3]$ and output the patch for each pixel with shape $[H, W, K \times K, 3]$. The function should work when `k_size=1`.

   iii. (9pts) Complete the three metrics in function `ssd_kernel`, `sad_kernel` and `zncc_kernel`. In `zncc_kernel`, you should return the negative zncc value, because we are going to use `argmin` to select the matching latter. You can find the definition of these three matching metrics below. The metrics treat each RGB value as one grayscale channel and finally you should sum the three (R,G,B) channels (sum across the channels at each pixel, i.e. [H,W,3] would go to [H,W]). The input of each kernel function is a src `[M,K*K,3]` that contains M left patches and a dst `[N,K*K,3]` that contains N right patches. You should output the metric with shape `[M,N]`. Each left patch should compare with each right patch. Try to use vectorized numpy operation in the kernel functions. You are going to get an example plot for pixel (400,200) of the left view and its matching score on the right view. **Note**: We define a small number EPS, please add the EPS to your denominator for safe division in zncc.

- SSD (Sum of Squared Differences) $\sum_{x,y} |W_1(x,y) - W_2(x,y)|^2$
- SAD (Sum of Absolute Differences) $\sum_{x,y} |W_1(x,y) - W_2(x,y)|$
- ZNCC (Zero-mean Normalized Cross Correlation), sometimes just "NCC"

$$\frac{\sum_{x,y}(W_1(x,y) - \overline{W_1})(W_2(x,y) - \overline{W_2})}{\sigma_{W_1}\sigma_{W_2}}$$

  - where $\overline{W_i} = \frac{1}{n}\sum_{x,y} W_i \qquad \sigma_{W_i} = \sqrt{\frac{1}{n}\sum_{x,y}(W_i - \overline{W_i})^2}$

iv. L-R consistency: When we find the best matched right patch for each left patch, i.e. argmin along the column of the returned [M,N] shape value matrix, this match must be consistent with the match found from the other direction: for each right patch find the best matched left patch, i.e. argmin along the row of returned [M,N] shape value matrix. We provide an example code of the LR consistency check, please understand this code.

v. (15pts) Implement the full function `compute_disparity_map` using what you understand from above examples (you can directly copy the example code and then expand upon it). From the consistency mask plotted in the notebook, what issue does the LR-consistency-check attempt to resolve? **Add the answer to your report.**
Hint: one call of `compute_disparity_map` might takes 1-2min, you can use `tqdm` to get a progress-bar.

(c) (6pts) Compute Depth Map and Point Cloud: given the disparity map computed above, complete the function `compute_dep_and_pcl` that return a depth map with shape [H,W] and also the back-projected point cloud in camera frame with shape [H,W,3] where each pixel store the xyz coordinates of the point cloud.

(d) (3pts) We implemented most of the post-processing for you to remove the background, crop the depth map and remove the point cloud out-liers. You need to complete the function `postprocess` to transform the extracted point cloud in camera frame to the world frame.

(e) (3pts) We implemented the visualization for you with K3D library; you can directly visualize the reconstructed point cloud in the jupyter notebook. **Include in your report** the screenshot of your Reconstruction using SSD, SAD and ZNCC kernel.

(f) (3pts) Multi-pair aggregation: We call your functions in the full pipeline function `two_view`. We use several view pairs for two view stereo and directly aggregate the reconstructed point cloud in the world frame. **Include in your report** the screen shot of your full reconstructed point cloud of the temple. Reconstruction may take around 10 min on a laptop.

Students need to complete in `two_view_stereo.py`:

`compute_right2left_transformation`

`compute_rectification_R`

`image2patch`

`ssd_kernel`

`sad_kernel`

`zncc_kernel`

`compute_disparity_map`

`compute_dep_and_pcl`

`postprocess`

2. **Plane-sweep stereo (35pts)**

   You will now compare your two-view results with a multi-view stereo algorithm known as plane-sweep stereo applied to the same temple dataset. This time, we will utilize 5 different views, choosing the middle-most view as the reference view.

   The main code for plane-sweep stereo is in `plane_sweep.ipynb` which imports functions from both the `two_view_stereo.py` and `plane_sweep_stereo.py` files.

   To give the high level steps, we will briefly describe the algorithm along with the functionality you will be responsible for:

   (a) **(25pts) Warping neighboring views**

   The key idea is that we sweep a series of imaginary depth planes (in this example, fronto-parallel with the reference view) across a range of candidate depths and project neighboring views onto the imaginary depth plane and back onto the reference view via a computed collineation.

   You will be responsible for implementing the functions

   i. (5pts) `backproject_corners`

   ii. (5pts) `project_points`

   in `plane_sweep_stereo.py`.

   (15pts) In the `warp_neighbor_to_ref` function in the same file, you will use the above two functions together with the built-in cv2 functions `cv2.findHomography` and `cv2.warpPerspective` to compute the homography and warp the neighboring view via the homography respectively.

   (b) **(5pts) Computing a cost map**

   i. Now, we will produce a cost/similarity map between the reference view and the warped neighboring view by taking patches centered around each pixel in the images and compute similarity for every pixel location. At patches with the correct candidate depth, the similarity should be high, and the similarity should be low for incorrect depths.
   You can use the same `image2patch` function you implemented in `two_view_stereo.py` to return patchified images.

   ii. (5pts) In this example, we will use the Zero-Normalized Cross Correlation (ZNCC) metric to measure similarity between the patches which you have implemented above in the two-view case. However, you need to make a slight modification in this example to return the cost map at each pixel over the entire 2D image. Also, remember that we compute the

ZNCC across each RGB channel separately but sum the results of each channel at every patch.
You will need to implement the `zncc_kernel_2D` in `plane_sweep_stereo.py` which should be similar to your implementation of the function in the two-view setting, but with the aforementioned modifications.

(c) **Constructing the cost volume** *All of this portion is implemented for you.*

Note: for faster debugging, you can control the number of depths to sweep through by decreasing the `num_depths` variable from 25 to a lower number. However, remember to increase it back for best results downstream.

  i. For each depth plane, we repeat the above steps of computing a cost map between the reference view and each of the 4 neighboring views and sum the results of each of the 4 pairs to aggregate into a single cost map per depth plane.

  ii. Next, we construct the cost volume by repeating the aforementioned steps across each of the depth planes to be swept over and stacking each resulting cost map along the depth axis.

  iii. In the notebook, you should get a gif of the cost maps sweeping across each depth and hopefully, be able to see a slice of high similarities sweeping across the 3D temple model.
  reference video here (we've grayscaled in your implementation)
  IMPORTANT NOTE: the markdown cell to show the written gif file has some mysterious caching that prevents automatically updating the displayed gif to match the locally written one. We recommend you manually open/see the locally written gif everytime you must rewrite the gif.

  iv. Finally, to extract a depth map from the cost volume, we choose a depth candidate by taking the argmax of the costs across the depths at each pixel.

(d) **(5pts) Colorized pointcloud from depth map**

  i. (5pts) We can obtain a pointcloud from our computed depth map, by backprojecting the reference image pixels to their corresponding depths, which produces a series of 3D coordinates with respect to the camera frame. You are responsible for implementing this in the `backproject` function in `plane_sweep_stereo.py`.

  ii. However, we must then transform these coordinates to be expressed with respect to the world frame instead of the camera frame. This has already been done as before in the `postprocess` function in `two_view_stereo.py`

  iii. Similarly, as in the two-view case, we apply some post-processing to get a nice looking colorized pointcloud from the depth map by filtering the black background and any potential noisy outlier points which we have implemented for you in the `postprocess` function.

To summarize, for this portion of the assignment, you will be responsible for:

(a) `backproject_corners` in `plane_sweep_stereo.py`

(b) `project_points` in `plane_sweep_stereo.py`

(c) `warp_neighbor_to_ref` in `plane_sweep_stereo.py`

(d) `zncc_kernel_2D` in `plane_sweep_stereo.py`

(e) `backproject` in `plane_sweep_stereo.py`