Homework 2

Instructions

This homework contains a set of writing assignments (Problem 1) and programming assignments (Problem 2). It has a total of 100 points.

For Problem 1, compile your answers in PDF and title it `hw2.pdf`. For Problem 2, you will directly be editing the Python file `tsdf.py` provided by us.

Submission: Files to submit 1) `hw2.pdf` 2) `tsdf.py`. Please do not introduce additional dependencies or packages that are not already included in this homework. When you are done, zip the following files. Name the zip to `SUID_hw2.zip`, and upload it to Canvas before the homework deadline.

Problem 1 *(40 points)*

1.1 Homogeneous and Cartesian coordinate. – 4 points

Given a homogeneous coordinate $P = \{34, 56, 10\}$, what is its corresponding Cartesian coordinate?

1.2 3D to 2D projection. – 5 points

Given a 3D point P in the camera frame ${}^{C}P_{3d} = \{-2, 3, 5\}$. The camera's focal length $f = 100$. Image size width $\times$ height $= 400 \times 300$. If we assume the image frame A is aligned with the camera (as shown in the following Figure), what is the 2D coordinate of this point in the image frame A ${}^{A}P_{2d}$?

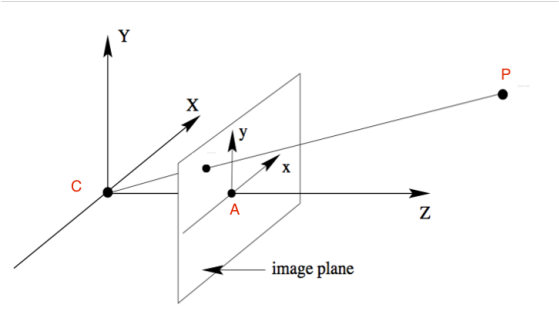


Figure 1: 3D to 2D projection

1.3 Change coordinate frame from World to Camera. – 5 points

Given a 3D point ${}^{w}P = [3,1,5]$ in the world frame. Camera pose in the same world frame as $T_{cam-pose} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 3 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

What is the P's position in camera frame ${}^{c}P$?

1.4 Signed distance function (simple 2D example) – 6 points

In the Figure below, the green line denotes a 2D surface S in this 2D space, grid size = 1cm. What is the signed distance value at locations A, B, and C, without normalization? Note: we define the sign as negative for location inside the surface and positive outside the surface.

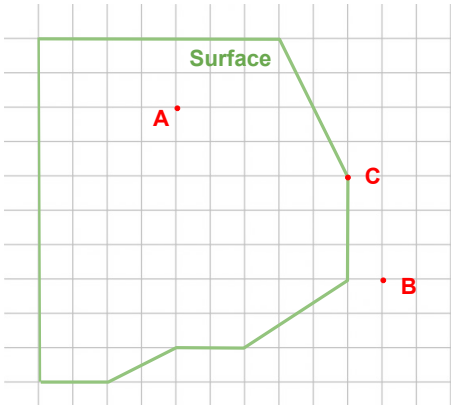


Figure 2: Signed distance function

1.5 Projective signed distance function – 10 points

Now we will add a camera to the scene `(cam_0)` and observe the surface from the right side, as shown in Figure 3, where the camera only observes part of the object's surface. Point A and C are sitting on the same projection ray, i.e., they both get projected on the same pixel (u, v) on image I.

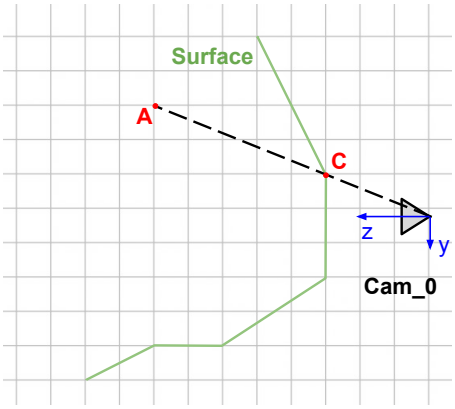


Figure 3: Projective signed distance function

1. What is the depth of point C with respect to the camera frame `depth`(C)?
2. What is the depth of point A with respect to the camera frame `depth`(A)?

3. What is the depth reading on the depth image $I_{\texttt{depth}}(u, v)$?

4. What is the projective SDF value for point A with respect to this camera? Based on the following definition: $\texttt{proj\_sdf}(A) = I_{\texttt{depth}}(u, v) - \texttt{depth}(A)$.

5. If we set the truncation value to be 8 cm, what is the projective TSDF $\texttt{proj\_tsdf}(A)$ value after normalization? What if the truncation value is 2 cm?

1.6 Integrate projective TSDF from multiple views – 10 points

Now we add another camera `cam_1`, that is observing the same surface from the top, as shown in Figure 4. This new camera sees a different part of the surface, labeled using yellow lines. This surface partially overlaps with the surface observed by cam_0. With this new observation, we can update all the projective TSDF values in the space.

1. What is the projective TSDF value for point A with respect to `cam_1`, before normalization and truncation?

2. Consider both camera observations, what should be the updated projective TSDF value, before normalization and truncation?

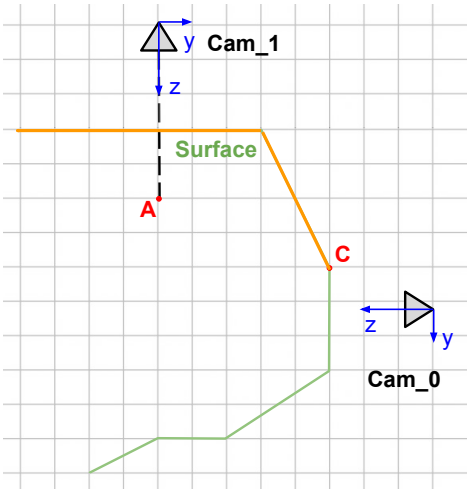


Figure 4: Integrate projective TSDF

Problem 2 (60 points)

Getting started. This part is the same as in HW1. We will install Python interpreter and dependencies using miniforge to avoid any version issues. Please follow the installation instructions for your system (Unix-like or Windows). You can also download the corresponding Mambaforge-OS-arch.sh/exe file at miniforge and execute the downloaded script. After installation and initialization, launch a **new** terminal and run the following command inside the unzipped homework zip file.

```
mamba env create -f environment.yaml
```

This will create a new environment named "hw2", which can be activated by running

```
mamba activate hw2
```

Implementations

You are now ready to implement a tsdf fusion loop! As discussed in the lecture, tsdfs are a way to integrate RGB-D images from different camera poses into a cohesive, implicit 3D reconstruction. We say the tsdf is implicit, because it does not directly contain 3D point locations or faces. Rather, the tsdf encodes the distance from voxels to the nearest surface, up to a truncation threshold. If a voxel is "behind" a surface, the tsdf value for this voxel will be negative. If the voxel is exactly on the surface, the value should be zero. If the voxel is in front of the surface, the value will be positive. Using this implicit representation, we can recover an explicit mesh by looking for zero crossings, where tsdf value switches between negative and positive.

In `tsdf.py` we define the `TSDFVolume` class. As part of the stencil code, we define many instance variables to be used.

```
self._voxel_size : float side length in meters of each 3D voxel cube.

self._truncation_margin : float tsdf truncation margin, the max
    ↪allowable distance away from a surface in meters.

self._volume_bounds : Numpy array [3, 2] of float32s, where rows
    ↪index [x, y, z] and cols index [min_bound, max_bound]. Note:
    ↪these bounds are rounded in such a way that dimension length
    ↪divided by self._voxel_size would be a whole number. Units are
    ↪in meters.

self._volume_origin : Origin of the voxel grid in world coordinate (
    ↪not voxel coordinates). Units are in meters.
```

```
self._tsdf_volume : Numpy array of float32s representing tsdf volume
    ↪where each voxel represents a volume self._voxel_size^3. Shape
    ↪of this volume is determined by (max_bound - min_bound)/ self.
    ↪_voxel_size. Each entry contains the distance to the nearest
    ↪surface in meters, truncated by self._truncation_margin.


self._color_volume : Numpy array of float32s with shape [self.
    ↪_tsdf_volume.shape, 3] in range [0.0, 255.0]. So each entry in
    ↪the volume contains the average r, g, b color.

self._voxel_coords : Numpy array [number of voxels, 3] of uint8s.
    ↪Each row indexes a different voxel [[0, 0, 0], [0, 0, 1], ...,
    ↪[0, 1, 0], [0, 1, 1], ..., [1, 0, 0], [1, 0, 1], ..., [x-1, y-1,
    ↪ z-2], [x-1, y-1, z-1]]. When a new observation is made, we need
    ↪ to determine which of these voxel coordinates is "valid" so we
    ↪can decide what voxels to update.
```

We provide the following useful functions in the starter code:

```
integrate(...) The main tsdf fusion method provides the steps and
    ↪examples of the function calls to complete the whole process.

get_valid_points(...): Compute a boolean array for indexing the voxel
    ↪ volume and other variables. Note that every time the method
    ↪integrate(...) is called, not every voxel in the volume will be
    ↪updated. This method returns a boolean matrix called
    ↪valid_points with dimension (n, ), where n = # of voxels. Index
    ↪i of valid_points will be true if this voxel will be updated,
    ↪false if the voxel needs not to be updated.

get_volume(...) retrieve volumetric data from your fusion

get_mesh(...) reconstructs a 3D mesh from a tsdf using the marching
    ↪cubes algorithm (https://en.wikipedia.org/wiki/Marching_cubes).
```

Your goal is to implement the following functions, using the instance variables described above and the functions you implemented in HW1 `transforms.py`. Update variables as needed to integrate new rgb-d observations into the tsdf volume. You can find the description for each of the methods below. The detailed documentation of each of these methods can be found in `tsdf.py`.

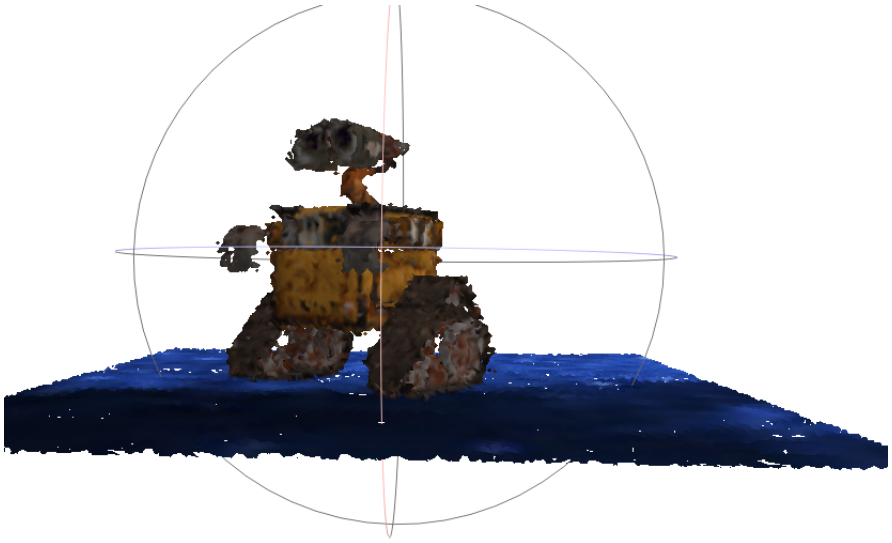
```
voxel_to_world(...): 10 pt. Convert from voxel coordinates to world
    ↪coordinates (in effect scaling voxel_coords by voxel_size and
    ↪shift by volume_origin).

transfrom_wrd2cam(...): 10 pt. Transform the points in the world
    ↪frame to the camera frame, using camera pose. Similar to the
    ↪problem 1.3.

compute_tsdf(...): 20 pt. Compute the new TSDF value for each valid
    ↪point. remember to apply truncation and normalization in the end
    ↪, so that tsdf value is in the range [-1,1]. Similar to the
    ↪problem 1.5

update_tsdf (...): 20 pt. Update the TSDF value and color for the
    ↪voxels that have new observations. Only update the tsdf and
    ↪color value when the new absolute value of tsdf_new[i] is
    ↪smaller than that of tsdf_old[i]. Similar to the problem 1.6
```

Testing and Debugging

To test your implementation, you can run the following, which should conduct fusion on 10 RGB-D frames in the `data` directory with some predefined volume settings:

```
python tsdf_run.py
```

Using your `tsdf.py` implementations, this script should create `ouput.ply`, which should show a Wall-E robot. When visualizing the mesh, open the file with MeshLab and select `Face` as the shading option. These tips will improve the visualization. Here are some example outputs:



Tips: To speed up the debugging process, can change the for loop in `tsdf_run.py` to consider only one frame, instead of 10. If everything is implemented correctly, you should be able to see Wall-E in the output mesh, however, with more missing surfaces.

Issues with numba: In the provided implementation, we use package `numba` and `prange` to take advantage of parallelism and speed up the compute. If your system has issues with `numba`, you can modify the code to bypass the problem. The code should be able to run properly, just a bit slow. Should still finish the 10-frame fusion within 10 minutes, but debug with one frame first to save time.

1. Change prange to range in both `transform.py` and `tsdf.py`
2. remove `@njit(parallel=True)` in both `transform.py` and `tsdf.py`
3. remove `from numba import njit, prange` in `transform.py`