CMPSC 458 Computer Graphics
Project 1, Fall 2023

version 1 of project description, released Sept 12

1 Motivation

In this project you will read in human body joints derived from Motion Capture (Mocap) data and visualize them as a 3D rendered stick figure with spheres at the joints and cylindrical "bones" between joints. The primary goal is to become familiar with the use of 3D coordinate transformations in computer graphics. To this end, there are three parts of the project where you will be generating and using coordinate transformations:

- You will do a rigid transformation of the joints to bring the human body into a canonical location and orientation in world coordinates;
- Given 3D object models of a unit sphere and a cylinder, you will generate multiple instances of each and transform them nonrigidly from object space into world space to form the joints and bones of a 3D stick figure model; and
- Given a camera location in world coordinates, a gaze direction pointing towards the body from that location, and an "up" vector, you will generate a world to camera rigid transformation using the procedure presented in class and the textbook.

As secondary goals, we will be gaining experience using python and numpy (for linear algebra and matrix transformations), pytorch3D (for rendering), Jupyter notebooks (for interactively developing and documenting code), and Google Colaboratory (a compute platform). You will be given a Jupyter notebook containing starter code that you will add to and modify to perform the project tasks.

Special note: In all cases of applying transformations to points, we want you to write the code yourself that generates and applies transformation matrices. There may be subroutines in pytorch3D or openCV or other libraries that apply geometric operations such as translation, rotation or scaling to sets of points represented as arrays or tensors, and that do so very efficiently. **Don't use those routines.** Use numpy arrays to represent homogeneous coordinate transformation matrices, computing and filling in the matrix elements yourself, and apply these transformations to vectors representing 3D points (or to an array representing a set of vectors as either the cols or rows) by directly doing matrix multiplication in numpy. In the case of 1D and 2D arrays, as we have here, either numpy.matmul or numpy.dot will do the multiplication. The goal is to understand the matrix math underlying geometric transformations.

2 The Basic Operations

Reading the Data

The starter code reads in an array of body poses, represented by a set of 17 “joint” locations estimated by a Mocap system during a Taiji performance. See left side of Figure 1 for a diagram showing the 17 joint locations. The shape of the pose data is 16876x17x4. The first dimension is a time index. The data is recorded at 50 frames per second, and 16876 frames / 50 fps is 337.52 seconds, or roughly 5.6 minutes of data. The second dimension indexes the 17 joints. The 17 joints are stored in the following order: Right Shoulder, Right Elbow, Right Wrist, Left Shoulder, Left Elbow, Left Wrist, Right Hip, Right Knee, Right Ankle, Left Hip, Left Knee, Left Ankle, Pelvis Center, Waist, Top of Neck, Clavicle, Thorax. The last dimension indexes into homogeneous world coordinates of each 3D point, in the order $[X,Y,Z,1]$. The Z axis of the world is vertical, pointing up opposite the direction of gravity. X and Y are horizontal coordinates. It is a right-handed coordinate system. The coordinates can be relatively large numbers, they are measured in millimeter units!

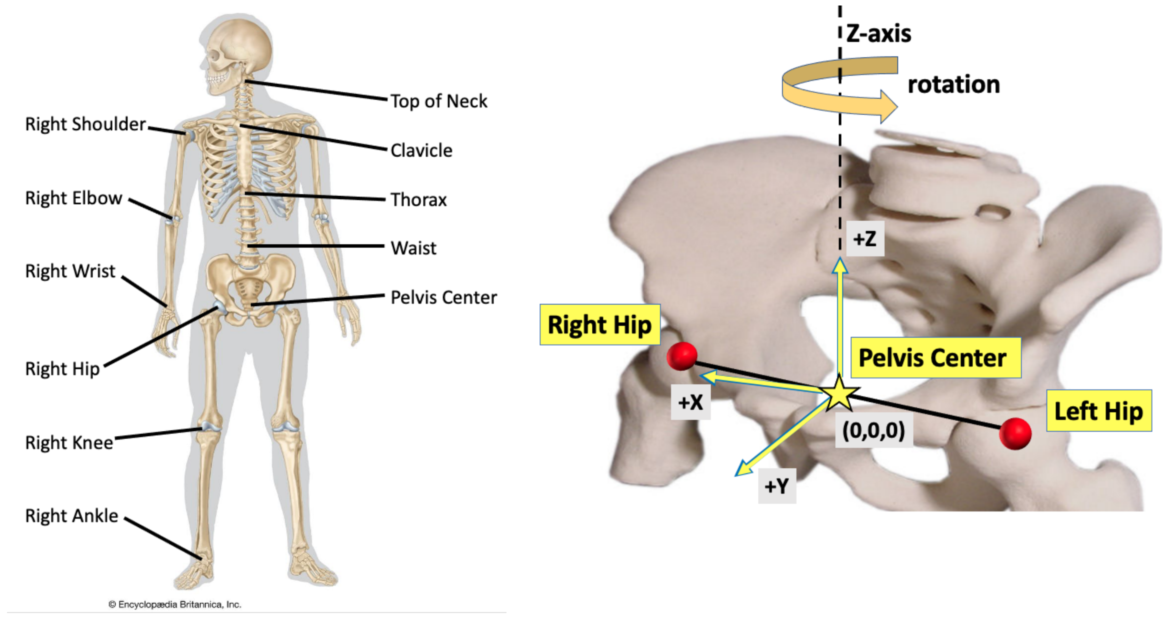

Figure 1: Left: Body pose is represented by 17 joint locations (to simplify the figure, the left arm and leg joint locations are not shown). Right: The body will be normalized into a canonical location and orientation by centering the Pelvis Center at the origin and rotating around the vertical Z axis to align left-to-right body direction with the +X axis and back-to-front body direction with the +Y axis.

Centering and Rotating to Normalize the Joint Data

Body pose is measured as a person moves around in the Mocap lab, and at any given time the body can be in an arbitrary location and facing in an arbitrary direction in the room. It is common to preprocess pose data to normalize it to a canonical location and orientation, because it is then easier to compare body poses and to apply learning-based pose recognition algorithms. We also will be doing it here, prior to visualization, because it makes it easier to define light sources and camera views in known relative poses to the body.

Body pose will be normalized to make the 3D point data invariant to the body's location and facing direction in the capture space. There are several possible ways to normalize pose data, but we will use a simple approach that uses points in the existing data. Refer to right side of Figure 1. Centering is performed by subtracting from each 3D point the location of the "Pelvis Center" point, thereby translating the Pelvis Center to the origin (0,0,0). An alternative, is to just subtract off the X and Y components of the Pelvis Center to keep the Z coordinate of each point (representing height above the floor) the same. After centering, orientation of the body is normalized by rotating points around the Mocap Z-axis so that the side-to-side vector pointing from "Left Hip" to "Right Hip" will be rotated to lie roughly along the positive Mocap X-axis and a vector pointing in the forwards direction from the body/pelvis will lie roughly along the positive Y-axis.

Hint: use the "Constructing a Basis from Two Vectors" procedure presented in class and in the textbook that defines a rotation by specifying one vector that becomes the Z axis (in this case it IS already the Z axis), and another vector that becomes approximately the X axis. As we know, the resulting orthonormal basis can also be interpreted/used as a 3x3 rotation matrix.

Building a Complex Model out of Simple Parts

Graphics is all about divide-and-conquer, both to make complex transformations by composing sequences of simple ones, and to form complex models out of arrangements of simple shape components. We will render human pose as a skeleton-like, 3D stick figure, with spheres representing the 3D pose points and with narrow cylinders connecting pairs of points.

You are given two simple shape primitives that you can use. A unit sphere (radius=1) centered at the origin, and a cylinder (circular radius = 0.5, height = 10) centered at the origin and with the height axis aligned with the Z axis. To generate a 3D stick figure model to visualize, you will want to make at least 17 spheres scaled to radius R (or R_i if you want to make spheres of different sizes for different body joints) and translated to be centered at the given 3D joint locations. To make "sticks" connecting pairs of spheres, you will make copies of the primitive cylinder and appropriately scale, rotate and translate it so that it has the desired endpoints. It is likely that the hardest part of this project will be writing a subroutine that makes a cylinder connecting two given 3D points by nonuniform scaling, rotation, and translation. You can choose which pairs of joints to connect together to make a connected body skeleton (some are obvious, like connecting shoulder to elbow, then elbow to wrist).

The sphere and cylinder shapes are given to you as "triangle mesh" structures. We will talk more about this surface representation in class in the coming weeks, but a triangle mesh consists of a set of 3D "vertices", which are just 3D points on the surface of the shape, and a set of "faces", each of which specifies three indices into points in the vertex list that connect together to form a triangular surface facet. For purpose of this project, you will be leaving the faces values alone and only manipulation the vertices as 3D points. For example, by scaling and translating the vertex points of a unit sphere mesh, while keeping the original face index values, you will be generating a new mesh representing a sphere with a new radius and center location.

The sample code shows an example of cloning primitive shapes, modifying their vertex locations, and incrementally joining them together into an increasingly more and more complex model.

Generating a World to Camera Transformation

To visualize the 3D body model, we will render it from a desired camera viewpoint, by filling in the values of a rigid transformation relating camera to world. This transformation task is likely to be the easiest, because it directly corresponds to a procedure we talked about in class and that also appears in the textbook in the chapter on Viewing. Choose an eye position (camera location in world coordinates), a gaze direction (aka look direction vector), and an up direction that in this case will be the positive Z axis of the world, just follow the procedure that generates the translation and rotation matrices that form the World to Camera (canonical to frame) change of basis transformation.

Due to our earlier normalization of the body into a predetermined location and orientation, we are more easily able to specify a camera viewpoint that is meaningful relative to the body. For example, if we place the camera on the positive Y axis of the world, looking back along the negative Y axis towards the origin, we should see a frontal view of the body. Similarly, if the camera is on the positive X axis of the world looking back along the negative X axis towards the origin, we should see the right side of the body. And so on.

This is a good time to remind you that we want you to compute the numerical entries of the rotation and translation matrices yourself, using numpy and linear algebra ideas that we have been going over in lectures. Pytorch3D has a function called "look_at_view_transform". **Do not use that function.** We may use such higher level, more efficient functions in future projects, but the whole point of this project is to understand how to specify the geometric transformations ourselves, at a fundamental level of detail.

For that matter, you are cautioned that Pytorch3D represents points as row vectors, and therefore when it makes a transformation matrix, it is the transpose of how we defined things.

Rendering an Image

After all the above is said and done, we are going to pass our 3D model and camera transformation information to Pytorch3D and let it do the final rendering. We can treat the rendering process as a black box, although later in the course we will be exploring the process in more detail (e.g. defining textures, shaders, types of lighting, etc).

This rendered image will be saved to a file, which you will be handing in along with your working notebook code and documentation.

3 Extra Credit

The main project just generates a single, static image. One idea for extra credit is to create an animation by generating multiple images, saving them out as numbered files, then using an external program such as ffmpeg to turn them into a movie or animated gif. For example, your animation could show a stationary body pose viewed by a camera circling around the body and turning to stay pointed at the body while it moves. Alternatively, the camera could remain stationary, but you could animate the body by using a sequence of frames of data from the Mocap data file. If you do that, keep in mind that the Mocap data was recorded at a fairly high frame rate of 50 frames per second. Because Taiji is a slow activity, it would be reasonable to save compute time by only rendering every *N*th frame of a temporal sequence of that pose data. For example, sampling every 10th frame would still yield a sequence of body poses captured at 5 frames per second.

If you want to attempt this, note that the renderer in Pytorch3D can take not only a single camera viewpoint, but a set of them (that is why the variable and named argument to the renderer function is the plural word “cameras”. For inspiration, look at the batched rendering section of the pytorch3D tutorial https://pytorch3d.org/tutorials/render_textured_meshes .

4 Grading

You will turn in your runnable Jupyter notebook, and one or more rendered images (and an animated movie if you did the extra credit).

Half of your grade will be based on documenting what you have done (using text, within the jupyter notebook). The other half will be based on your code running and doing what it is supposed to do. We will test your code on a set of known poses and camera viewpoints of our choosing to see how well the results match what our instructor code produces.