Visualizing Pose Estimation Data with Blender

Amaan Rahman
Albert Nerken School of Engineering
The Cooper Union
New York, United States
amaan.rahman@cooper.edu

Prerequisites

This paper assumes that the reader has general knowledge of how to navigate and take advantage of the various tools of Blender. If the reader has no prior knowledge of utilizing the Blender software, then follow this procedure as follows:

1) Blender Installation
2) *Recommended Basic Tutorials*:
    a) **Interface Tutorials**:
        i) Interface Tutorial #1
        ii) Interface Tutorial #2 (Highly Recommended)
    b) **Armature Tutorials**:
        i) Armature Tutorial #1
        ii) Armature Tutorial #2
        iii) Armature Tutorial #3
3) *Blender Documents*:
    a) Python API
    b) Math Types API
    c) Armature API
    d) Bones (EDIT mode) API
    e) Bones (POSE mode) API
    f) Blender Interface Manual

***Abstract*—This paper discusses about how to setup a Blender model to be syncronized with 3D coordinates of selected joints for syncrhonization in real-time.**

***Index Terms*—Blender, pose estimation, real-time, syncrhonization, joint-angle esimation**

I. Introduction

Pose estimation is a hot topic in the computer vision world. Primarily, it is used for tracking a desired subject (e.g. humans, animals, objects). Pose estimation can be used to estimate the 2D pose or the 3D pose; the latter case is more useful for 3D animations, and without the reliance of expensive motion capture systems, a low-cost camera setup with proper computer hardware can be utilized with 3D pose estimation algorithms to generate 3D pose data of the subject, or subjects. The 3D pose data can then be used in various applications such as Virtual Reality, Augmented Reality, and/or computer animations. This paper focuses on using 3D pose data for animating an arbitrary 3D model in Blender in real-time.

The method used to genereate the 3D pose data in this paper revolves around the 2D pose estimation algorithm OpenPose [1]. A stereo camera setup with low-cost RGB cameras were synchronized and calibrated to estimate the 2D pose of an individual from two different perspectives in a given frame. The estimated 2D poses were then triangulated to generate the estimated 3D pose of an individual at the given frame. The 3D pose data consists of the 3D coordinates of each joint of the individual based on Fig. 1.

The assumption in this paper is that the pose estimation data being imported into Blender contains 3D coordinates of selected joints of an individual.

II. Configurations

A. Importing Pose Estimation Data into Blender

There are various methods of importing raw 3D pose estimation data into Blender. This paper focuses on packaging the 3D pose estimation data into JSON format for human readability and ease in portability from the pose estimation model to Blender's python script. In this case, joints can be selected for Blender synchronization by including the specific keypoint

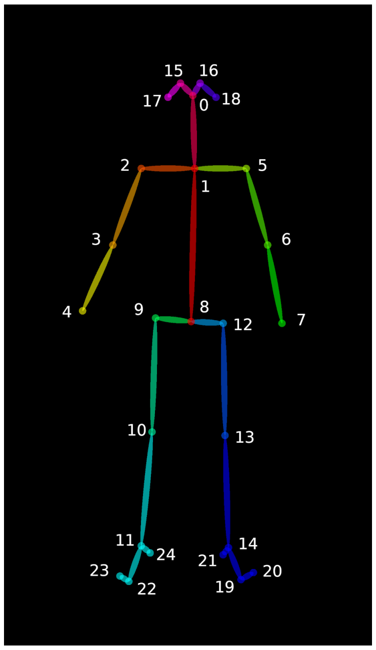


Fig. 1: Body25 joint keypoints format [1]

numbers. Additional infromation such as frame number can also be included so it can be processed in the Blender script. The JSON format follows the convention observed in Fig. 2a.

B. Coordinate Space for Pose Data

Based on Fig. 2b The origin is the principle camera of the stereo camera setup (this is configured when calibrating a stereo camera system utilizing either MATLAB or OpenCV). The z coordinate would be the estimated depth coordinate via a triangulation function. When the pose estimation algorithm fails to detect a joint, the estimated depth coordinate becomes a large negative float or as NULL as an indication of error.

C. Setting up the Blender World

The first thing that is to be done is to set up the basic armature skeleton. Delete the default cube, and using the shortcut **SHIFT+A**, click the armature option; the parent bone will be placed wherever the 3D cursor is located (at origin when starting up Blender). A bone has two parts to it: head, and tail according to Fig. 2c.

The parent bone will actually represent the "pivot" or crux of the entire armature. That is, because extruding a bone from the head of the parent bone causes that respective child bone to not pivot with respect to the head of the parent bone regardless of being linked (test it out in **POSE MODE**). The solution would be to shrink the parent bone as small as possible and use the tail as keypoint 8 in Fig. 1. All bones that have been extruded from the tail of its previous bone should all be configured to have its roll to 0° (press 'N' in **EDIT MODE**); configure only one half of the armature for there is a symmetry tool to auto-complete the armature under \Armature\Symmetrize, and it only works when \Armature\Names\**"Autoname Left/Right"** is enabled. It is highly recommended that the individual bone names are changed afterwards because a bone's name is its ID that is utilized to access its metadata when scripting. The result should look like Fig. 3a.

The pivot bone in Fig. 3b, which is between the two hip bones, has been minimized so it's not visible. The armature is layed out to look more stiff because this will represent the rest state of the armature, making it easier to apply different poses in real time.A very helpful tool that will ease the process in updating the poses of each bone is enabling the local coordinate axes for each bone in the armature; this can be enabled by going to the **Armature** tab in the **Properties** section and check **Axes** under **Viewport Display** Fig. 3b. To see the axes through the bones go to **Wireframe** mode by pressing 'Z'. The armature will now look like Fig. 3c.

D. Local Spaces in Blender

Armature space: the origin is the tail of the parent bone and its coordinate axes is that of the parent bone, which can also be seen on the tail of the main body bone. This space is utilized in **EDIT MODE**.

Pose space: the local space of each bone, where the origin is the head of the bone and the tail is offset by 1 pose space unit from the head. The coordinate axes are the axes displayed at the tail of its respective bone Fig. 3c. This space is utilized in **POSE MODE**.

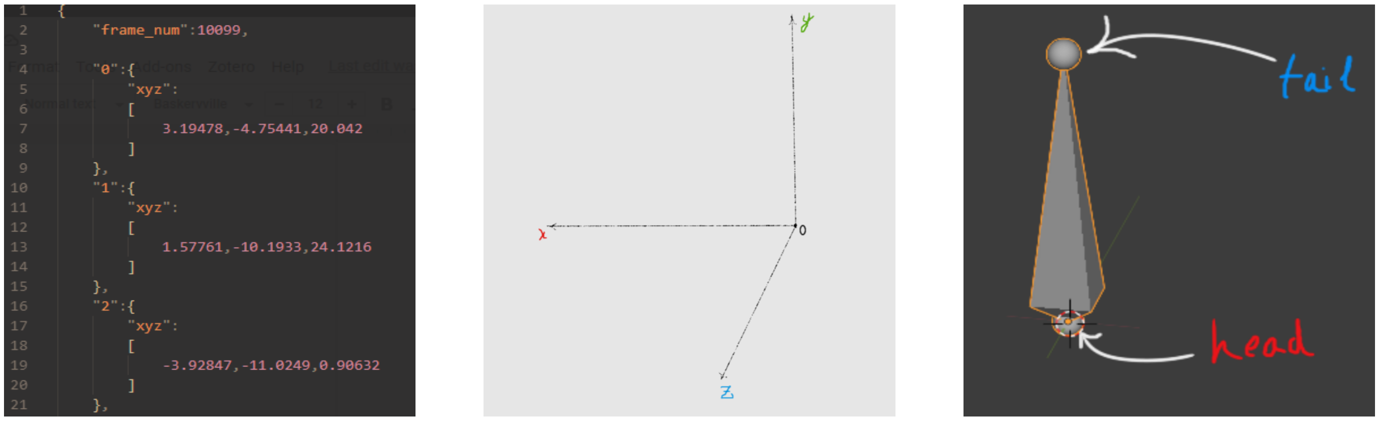
III. Real-Time Updating in Blender

A. Modal Operator Template

At this point, it is expected of the reader to have the JSON data file setup and the armature skeleton setup appropriately. Now it is time to set up the code that will update the armature's pose given the pose estimation data in real-time. It would seem wise to start coding the updating procedure, but the real-time aspect is what will cause a lot of trouble later on (trust me, I went through the painful realization). So how do we set up the framework for real-time updating of the armature?

Fortunately, there is a template for interacting with updating the Blender context window (Blender space and its interface, Fig. 4). As a consequence, the possibility of the armature being updated with the template as the framework is entirely possible.

The template is called **Operator Modal Timer**. The structure and layout of this python template will be the framework of generating a script that updates the pose of the armature with real-time pose estimation data. The class object *ModalTimerOperator* is what will allow for interacting with the Blender context window at real time. For this to happen, the class object has to be registered into Blender in order for the custom operator to be integrated into Blender. All custom classes must be registered appropriately in order to work alongside Blender's existing library and/or extend its functionality. The operator is called by its given ID name (*bl_idname*), which is *modal_timer_operator*, that is set to be part of the windows manager class under *bpy.op*.

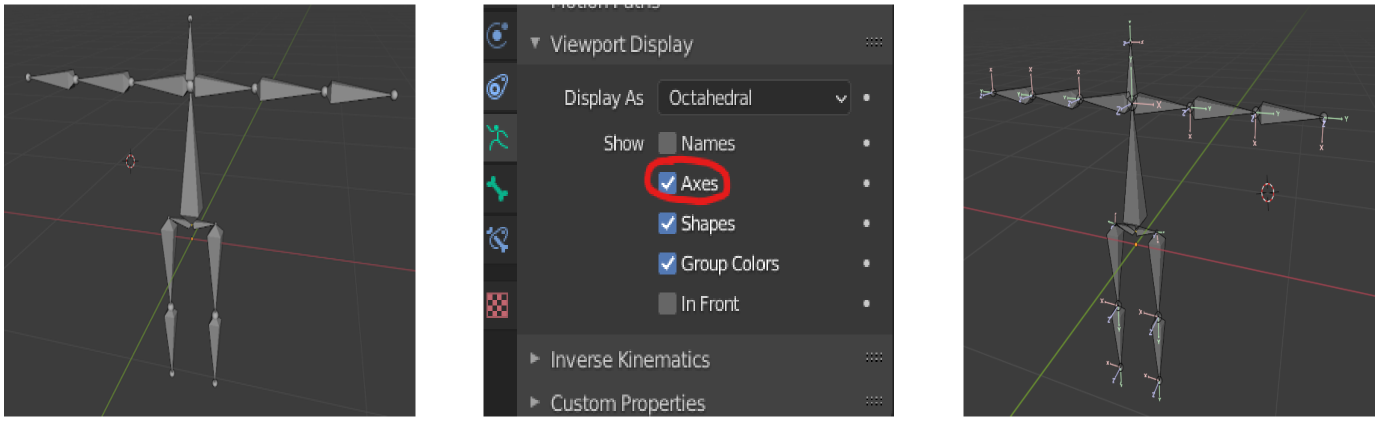


```
{
    "frame_num":10099,

    "0":{
        "xyz":
        [
            3.19478,-4.75441,20.042
        ]
    },
    "1":{
        "xyz":
        [
            1.57761,-10.1933,24.1216
        ]
    },
    "2":{
        "xyz":
        [
            -3.92847,-11.0249,0.90632
        ]
    },
```

(a) (b) (c)

Fig. 2: **(a)** JSON format indexed by keypoint number containing keypoint data. **(b)** Coordinate axes for pose data. **(c)** Blender armature bone.



Viewport Display

Display As Octahedral

Show
- [ ] Names
- [✓] Axes
- [✓] Shapes
- [✓] Group Colors
- [ ] In Front

(a) (b) (c)

Fig. 3: **(a)** Sample test armature model **(b)** Blender GUI for enabling visualizing bone axes **(c)** Armature with enabled bone axes.

Within the class in the template, it can be seen that there are a few functions named *modal*, *execute*, and *cancel*. The function *modal* handles the return cases of *'CANCELLED'*, *'PASS_THORUGH'*. *'CANCELLED'* return case is self-explanatory; it terminates the operator. *'PASS_THROUGH'* return case is to signal the operator to let the modal event complete and continue on. Within the *modal* function, there are two event cases: {*'RIGHTMOUSE'*, *'ESC'*} and *'TIMER'*. The former event is for terminating the operator given the appropriate keypress, and the latter event is to interact with the Blender context window (including the Blender rendering space) while the program is executing, rather than interact after execution (the usual case when scripting casually), at a given time interval (configured in the *execute* function); this is where the pose estimation data will be imported and updated to allow for real-time updates as the pose estimation algorithm is generating the data.

The *execute* function calls the operator itself. The *wm.event_timer_add(0.1, window=context.window)* configures the timer (initialized as *_timer = None*) to update the Blender context window at a specific timer interval (0.1 sec or 100 ms); the time interval is crucial because it represents the period of "frame" updates in Blender, which is supposed to be synchronized with the framerate of the pose estimation algorithm running on a stereo camera system (statistical tests can be done to determine the approximate framerate or seconds/frame). The return case *'RUNNING_MODAL'* is to signal Blender to continuously execute the operator.

B. Programming the Armature

Now that the template doesn't feel too alien, it can be utilized as the main framework for updating the armature's pose given pose estimation data in real-time. Firstly, the main class has to have the JSON file opened (never closed at any point while the operator is running or else the it will miss updates) and, most importantly, the armature has to be imported for accessing the bones (the goal is to update the

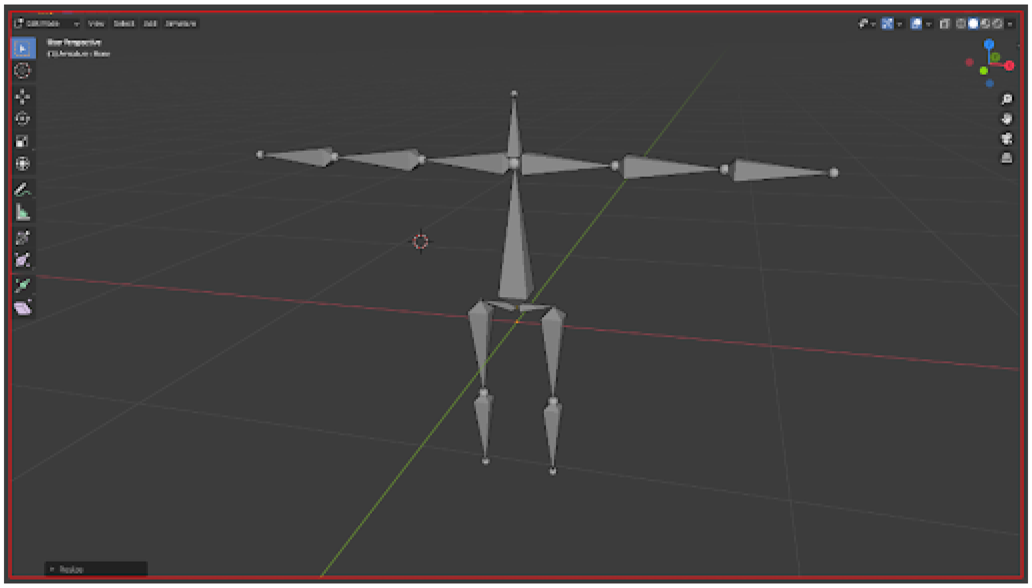


Fig. 4: Blender interactable interface window within the red box.

pose of each bone since the data is comprised of the position of the joints in stereo camera space).

The object interaction mode in Blender has to be configured to *'POSE'* because it decreases the complexity in updating the pose of a bone. If it was configured for **EDIT MODE** instead, then the complexity increases because in **EDIT MODE** a bone is in armature space; updating the position of the tail/head with its respective coordinate keypoint data directly will cause the physical shape of the bone to deform and may cause issues with other bones, thus this is unwanted behavior.

On the other hand, configuring the mode to **POSE MODE** will require rotations to be applied to the bone to update it. So why use this mode at all? It preserves the bones' shape, doesn't affect any other bone, smooth updates/transitions, and ease in interpolation. The problem is that this mode requires rotational values to update the pose of a bone, and the rotational value has to also account for how rotation in pose space is different compared to in world-coordinate space. Furthermore, the bones' local axes orinetations have to be accounted for when applying rotations.

Now that's settled, the initial step is to figure out a way to read the JSON file in python. Aforementioned, reading the JSON file will occur inside the *modal* function under *event.type == 'TIMER'*. It is suggested to use a *'try-except'* block when reading the file so the program can continuously read the JSON file, even if something unexepected corruption occurs. A quirky thing about python is that it has a lot of neat tools available for programmers. A helpful library is the *json* library, which can be used to load the JSON file as a python dictionary appropriately. As a result, when loading the JSON file the dictionary keys will be the keypoint numbers and the values of the keys will be dependent on what objects were associated with their keypoint numbers in the JSON file. In the case of Fig. 2a, the values will be array "xyz" of its corresponding keypoint number and the array contains the keypoint 3D coordinates (this could be handled more efficiently).

At this point, at a given frame, the 3D coordinates of all the keypoints have been stored some way in python. The challenge now is to figure out how to translate the currently stored 3D coordinates into some form of rotational updating or joint-angle estimation. Before we go any further with joint-angle estimation, we must understand the given methods in Blender to perform such a feat.

C. Quaternions [6]

There are two popular methods of encoding rotations in Blender: Euler angles and quaternions. The former is popular for its ease in understanding compared to the latter. However, Euler angles introduce the Gimbal Lock problem, which "is the loss of one degree of freedom" in 3D space [5]. Quaternions are more computationally efficient compared to euler angles and introduce no Gimbal Lock, but they are infamous for their difficulty to understand.

The structure of a quaternion:

$$\mathbf{q} = q_0 + q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k} \tag{1}$$

The real part of the quaternion is q_0 and there are three imaginary parts: $\{q_1, q_2, q_3\}$

Rotations can be percieved as motions that revolve around an arbitrary axis at a given angle; this is known as axis-angle representation and can be seen in Fig. 5. The axis-angle representation of a rotational motion can be encoded in a quaternion, $\mathbf{q}$, given an axis, $\mathbf{v} = v_x\boldsymbol{i} + v_y\boldsymbol{j} + v_z\boldsymbol{k}$, and an angle θ:

$$\mathbf{q} = \cos\frac{\theta}{2} + \left(v_x \sin\frac{\theta}{2}\right)\boldsymbol{i} + \left(v_y \sin\frac{\theta}{2}\right)\boldsymbol{j} + \left(v_z \sin\frac{\theta}{2}\right)\boldsymbol{k} \tag{2}$$

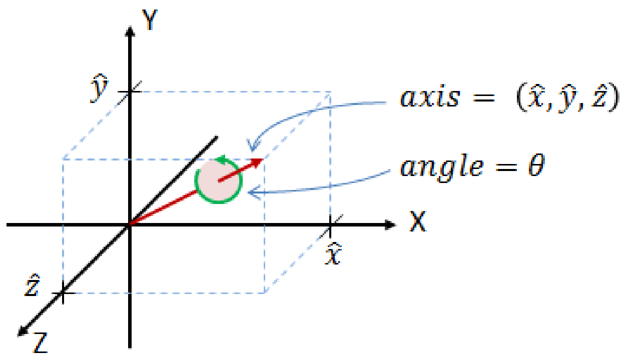


Fig. 5: Axis-angle representation of rotational motion. The rotational motion is θ degrees/radians about an arbitrary axis in 3D space with vector components $\langle \hat{x}, \hat{y}, \hat{z} \rangle$ [6].

In Blender, there are quaternion objects in the *mathutils* library [7]. A quaternion blender object has two forms:

- *quat = mathutils.Quaternion(*q_0, q_1, q_2, q_3*)*
- *quat = mathutils.Quaternion(*v_x, v_y, v_z, θ*)*

The first form is used when given the components of a quaternion like in Eq. 1. The second form is the one that this paper will focus on and uses axis-angle representation like in Eq. 2; θ must be in radians using *math.radians* function. In respect to the latter function, the reverse can also be done: extract the axis vector and the angle, in radians, from a quaternion; given an arbitrary quaternion, *quat*, this can be accomplished by:

quat.to_angle_axis()

It is always best practice to normalize vectors and quaternions, especially quaternions to perform valid rotations, in Blender to prevent unecessary complications. Vectors and quaternions share two normalization functions [7]:

- *normalize()*
- *normalized()*

The former returns the normalized result of the vector or quaternion the function was applied to. The latter modifies the vector or quaternion that the function is applied to and normalizes it.

Given a vector $\mathbf{u}$ and a normalized quaternion $\mathbf{q}$, $\mathbf{u}$ can be rotated by $\mathbf{q}$, being aware that 3D vectors are a special type of quaternion with no real part (pure quaternions):

$$\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^{-1} \tag{3}$$

$\mathbf{v}'$ is the resulting vector after performing a rotation encoded by $\mathbf{q}$. Due to $\mathbf{q}$ being normalized, $\mathbf{q}^{-1}$ is the same as $\mathbf{q}$ but the imaginary part of $\mathbf{q}$ negated.

In Blender, this is done through a simple function given a vector object *v*:

v.rotate()

Quaternion rotations can be combined as well. Given two quaternions, $\mathbf{q_1}$ and $\mathbf{q_2}$:

$$\mathbf{q_3} = \mathbf{q_1}\mathbf{q_2} \tag{4}$$

To accomplish this in Blender, the @ operator, or PEP 465 infix operator [8] in python, is used to perform matrix multiplication; also applies to quaternions as well.

Quaternion interpolation permits smooth rotational motions. There are various interlpolation methods, but the primary ones are Linear Interpolation (Lerp) and Spherical Interpolation (Slerp) [9]. In Blender, there is a function for Slerp called *slerp()*, but there is none for Lerp; the formula for Lerp could instead be programmed in python [9]:

$$Lerp\left(\mathbf{q_1}, \mathbf{q_2}, h\right) = \mathbf{q_1}\left(1 - h\right) + \mathbf{q_2}h, \;\; h \in [0, 1] \tag{5}$$

The linear interpolation occurs between $\mathbf{q_1}$ and $\mathbf{q_2}$ and is controlled by an interpolation factor, h, that ranges from 0 to 1.

D. Joint-Angle Estimation

Now that the theory and Blender tools regarding quaternions are covered, we can now layout the framework for joint-angle estimation code in Blender. Aforementioned, the pose of each bones need to be updated by estimating the new joint-angle to rotate a bone to its new pose. Given the 3D pose estimation data stored in a python dicotionary, corresponding vectors to each bone can be computed using Blender's vector objects. For instance, according to Fig. 1, a vector can be generated from keypoint 3 to keypoint 4, which corresponds to the armature's right forearm bone; the goal is to get the right forearm bone to align itself with the vector from keypoint 3 to 4.

The difficult part in joint-angle estimation is accounting for bones' local axes orientations. The armature space axes in Fig. 3c (can be seen on the tail of the main body bone) has its x-axis flipped compared to the axes the pose data is based off from in Fig. 2b. This is a trivial case that can be easily accounted for, however the bones' local axes for the arms and legs are oriented differently especially the left and right sides. For instance, the bones in the right arm have their local axes rotated 90° counterclockwise about the armature space z-axis (clockwise for the left arm). To account for these axes orietnation offsets relative to the armature space axes, the computed vector has to be initially rotated based on its corresponding bone's local axis orientation offset.

Once the vector for a corresponding bone is initialied and setup appropriately, the jont-angle between the vector and the bone can be computed as a quaternion using a special Blender function: *to_track_quat(track, up)*. The *track* paramter corresponds to the axis that the bone is closely aligned with, and the *up* parameter corresponds to the axis that bone should rotate about. Again, one must be aware of a bone's local axes orientation with respect to the armature space axes.

At this point, the quaternion for the joint-angle has been computed. The new quaternion must be interpolated with respect to the current bone's quaternion state; a bone's quaternion state can be aquired using *(*rotation_quaternion) attribute of the bone. Through experimentation, Slerp yielded rotation glitches when the joint-angle is acute and other issues such as rotations that complement the true rotation. Thus, Lerp was used instead and a simple python function was developed for it based on the Eq. 5. After interpolation, the bone can now be rotated to its new pose given the normalized quaternion (make sure all quaternions and vectors throughout the script are normalized).

IV. Conclusion

There is still a lot of work to be done with implementing a proper model for using Blender to visualize pose estimation data in real-time. Issues still exist with the orientations of the legs, and the smoothness of the rotations could be improved further; maybe different interpolation methods could be utilized to make this happen. Nonetheless, there is high potential in animating Blender models with pose estimation data in real-time, but still more work to be done.

References

[1] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh, "Openpose: Realtime multi-person 2d pose estimation using part affinity fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[2] T. Simon, H. Joo, I. Matthews, and Y. Sheikh, "Hand keypoint detection in single images using multiview bootstrapping," in *CVPR*, 2017.

[3] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, "Realtime multi-person 2d pose estimation using part affinity fields," in *CVPR*, 2017.

[4] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, "Convolutional pose machines," in *CVPR*, 2016.

[5] "Gimbal lock," Jan. 2021, page Version ID: 1000885298. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Gimbal_lock&oldid=1000885298

[6] "Quaternions." [Online]. Available: http://danceswithcode.net/engineeringnotes/quaternions/quaternions.html

[7] "Math Types & Utilities (mathutils) — Blender Python API." [Online]. Available: https://docs.blender.org/api/current/mathutils.html#mathutils.Quaternion

[8] "PEP 465 – A dedicated infix operator for matrix multiplication." [Online]. Available: https://www.python.org/dev/peps/pep-0465/

[9] E. B. Dam, M. Koch, and M. Lillholm, "Quaternions, Interpolation and Animation," p. 103.