Department of Computer Science and Engineering
IIT Bombay

CS691: R&D Project

Character Animation from Video in Blender

Author:
Mithilesh Vaidya (17D070011)

Guide:
Prof. Parag Chaudhuri

Autumn 2021/2022

Abstract

We present a Blender plug-in which accepts RGB videos as input and transfers the sequence of human pose present in the video to any target armature. The pipeline consists of two distinct stages: human pose extraction from videos and transferring the animation from source to target bones. VIBE and MediaPipe, two popular pose extraction backends, have been incorporated for pose extraction. For the second stage, we use graph edit distance for computing the best alignment of source and target bones. Further fine-tuning can be done using a manual mapping mode. After demonstrating the results, we explore the limitations of the model and propose a Graph Neural Network which can be trained in a self-supervised fashion to alleviate some of the issues which plague the naive graph-matching implementation.

1 Introduction

Animating a character in softwares such as Blender is a laborious time-consuming task. However, a large diverse set of motions carried out by humans exists in the form of RGB videos. An end-to-end pipeline for transferring such motion to any given target armature would be a useful tool in an animator's toolbox.

Since there is no universal standard for character skeletons, the pipeline needs to be robust to diversity in the target armatures. For example, some characters might have a fine-grained spine represented using 3 bones while some may have only a single spine bone. In such cases, an end-to-end pipeline which handles such variation is important in making the solution scalable.

The problem can be broken down into two stages:

1. Pose extraction: Given an RGB video, the first stage involves extracting human pose from every frame.[1] Although the number of predicted joints varies from model to model, they usually output angle rotations for each joint.

2. Mapping joints: Characters are represented as a directed acyclic graph (DAG) in which each node represents a bone in the skeleton. For transferring the pose, source bones need to be mapped to target bones. The joint angles are then copied over to the target armature[2].

We develop a plug-in specifically for Blender since it is open-source and very popular in the animation community. Moreover, its functionality can be easily extended with Python scripts. For pose extraction, we found an existing plug-in [1] for converting MediaPipe keypoint output into a .fbx file. On the other hand, the official repository for VIBE contained a script which converted VIBE output into .fbx. Additionally, we found two plug-ins which do a decent job at transferring animations from one armature to the other [11, 9]. Due to its superior functionality, we choose the former plug-in as our base code, strip away the unnecessary parts and extend it with various features. The plug-in is developed in a way such that the two stages are independent of each other. As a result, each stage can be tweaked without affecting the other. For example, a new state-of-the-art pose extraction model can be incorporated into the backend without disturbing the second stage of transferring animations from source to target. This modularity is an important feature for making the plug-in future-proof.

Over the course of our experiments, we found some limitations to our approach. More specifically, some animations cannot be transferred to the target armature, irrespective of the accuracy of the bone

[1]If the video contains multiple people, a person tracking algorithm must be incorporated in the pose extraction backend to track a single subject.

[2]In this work, the words armature and skeleton are used interchangeably. While skeleton refers to the generic English term, armature refers to it's representation in a software such as Blender.

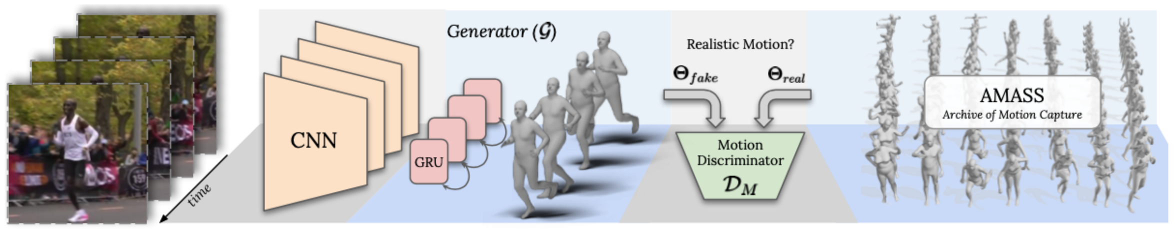


Figure 2: **VIBE architecture.** VIBE estimates SMPL body model parameters for each frame in a video sequence using a temporal generation network, which is trained together with a motion discriminator. The discriminator has access to a large corpus of human motions in SMPL format.

Figure 1: VIBE pipeline for extracting pose from an input RGB video. Image reproduced from [6].

mapping. Such issues arise when the target armature has a fundamentally different hierarchy. With the advent of deep learning, we plan to tackle this using a graph neural network.

The report is organised as follows. Section 2 introduces the pose recovery models which constitute the first stage of the pipeline. In Section 3, an algorithm for mapping source bones to target bones is discussed, along with its limitations. A video walkthrough and performance comparison of the backends is discussed in Section 4 while the graph neural network approach is introduced in Section 5. Concluding remarks are made in Section 6, along with a straightforward list of instructions for running the plug-in. Installation issues and other miscellaneous ideas are discussed in Appendix A and B.

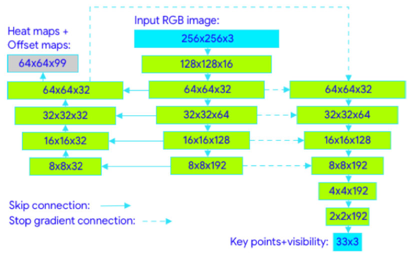
2 Pose Recovery models

For extracting human pose from a given RGB video, we implemented two popular existing frameworks[3]: MediaPipe [8] and VIBE [6]. Both are based on deep neural networks and trained on large datasets for accurate prediction of joint angles from a given RGB video. A brief description of both models is given below.

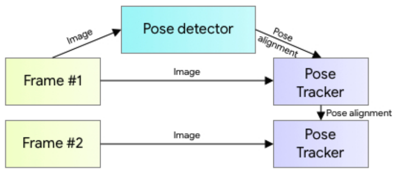
VIBE uses a pre-trained CNN for extracting pose information from each frame and embedding it in a high-dimensional space. For exploiting temporal information, the sequence of embeddings is fed to a GRU. This not only smoothens out the predictions over time but also helps the model infer parameters from frames containing joints which are occluded. 85 parameters of the popular SMPL [7] model are regressed from the output of the GRU. These include 72 joint angle parameters (24 per joint), 10 body parameters (for determining shape, height and body structure of the subject) and 3 camera translation parameters. A fixed transformation produces a mesh of 6890 vertices from these predicted parameters. To utilise unpaired 2D-3D examples, a GAN is trained to distinguish between a sequence of samples from the predicted model and a sequence of samples from a separate 3D database. Since size and quality of datasets is crucial for any deep learning model, this step enables it to exploit a large database of 3D motion capture dataset without the corresponding 2D RGB videos.

MediaPipe offers open source cross-platform, customizable ML solutions for live and streaming media. For this project, we use the Python implementation of BlazePose: On-device Real-time Body

[3]A crucial consideration for choosing the frameworks was availability of an open-source implementation. Since the main focus of the project was to use it as a black-box, any superior pose extraction method but without an available implementation could not be considered.



(a) The CNN pose extractor network.



(b) Overall block diagram.

Figure 2: The building blocks of BlazePose. Figure reproduced from [2].

Pose tracking [2]. It uses a light-weight CNN for pose extraction which is tailored for real-time inference. Unlike SMPL, which has 24 joints, MediaPipe estimates the locations of 33 joints.

For comparison purposes, both armatures are shown Figure 3.

Implementation: Plug-ins in Blender are written in Python. The pose extraction backend has a simple file selector frontend as shown in Figure 4. After installing the provided script, the file selector can be accessed by choosing the **Scripting** option in the Blender header. The UI in Figure 4 appears in the **Scene Properties** subsection. On choosing a .mp4 file, the corresponding pose extraction backend will run the deep learning model in inference mode and export a .fbx file which is subsequently imported into Blender.

In summary, the first stage of our pipeline consists of a pose extraction module which takes as input an RGB video and outputs joint angles (for a particular backend-dependent armature) for each frame. The resulting .fbx file is then imported into Blender.

3 Bone Mapping

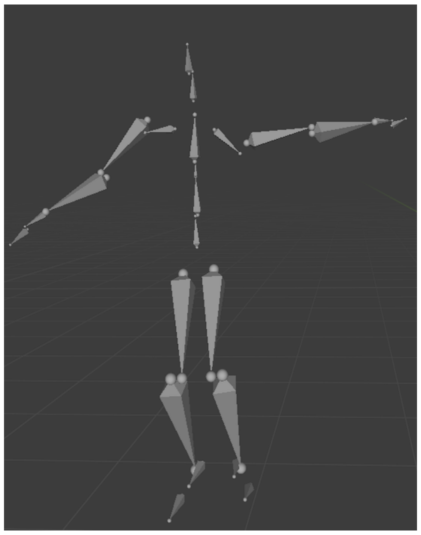
After extracting joint angles from the RGB video, the next stage involves transferring the animation from the source armature to the target armature. This can be achieved by mapping similar bones in the source and target armature and copying their animation parameters (joint angles). Note that due to differences in the hierarchy of the source and target skeleton, some bones in the source or target armature may remain unmapped e.g. the source may have 3 spine bones but the target has a single bone extending from the hip to the neck. Issues caused by such differences are discussed in Section 5.

We found an existing plug-in for Blender [11] which implements this transfer of animations. It has an elegant UI for mapping each source bone to a target bone. However, manually choosing the bones is tedious, especially for fine-grained armatures with a large number of bones. String matching algorithms operating on the bone names can be considered but the naming of the bones varies widely from armature to armature e.g. the thigh bone may be named as *L_thigh*, *thighLeft*, *hip_l* or *mixamo:upperleg_left*. Thus, simple string matching algorithms for mapping bones are inadequate.

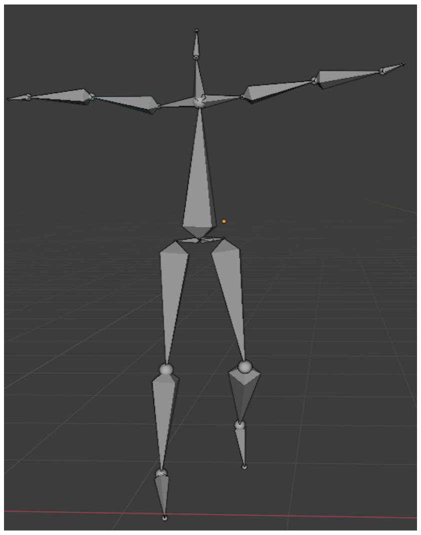
A screenshot of the UI can be found in Figure 5.

3.1 Graph Matching

To make the pipeline robust to bone name variations, we implemented a graph matching algorithm which generates an accurate first guess for the bone mapping.

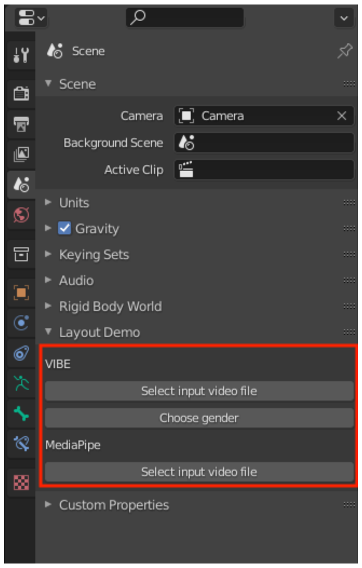


(a) Armature for VIBE



(b) Armature for MediaPipe

Figure 3: Armatures for the two pose extraction backends. Note that VIBE has 24 joints while MediaPipe has 33.



Scene

Scene

Camera: Camera

Background Scene

Active Clip

Units

Gravity ✓

Keying Sets

Audio

Rigid Body World

Layout Demo

VIBE

Select input video file

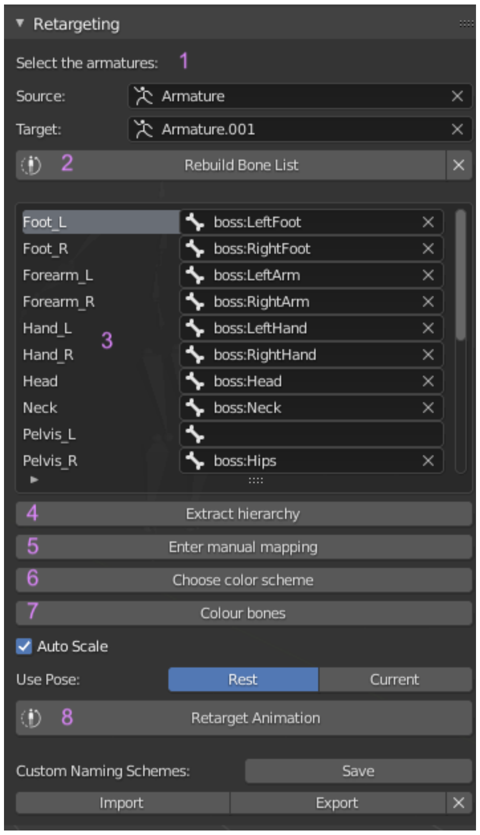
Choose gender

MediaPipe

Select input video file

Custom Properties

Figure 4: Video file selector. Since VIBE offers two separate SMPL models for males and females, the user can accordingly choose the gender. The resulting SMPL mesh will vary but the armature hierarchy is identical for both genders.



Select the armatures:

Source: Armature

Target: Armature.001

Foot_L	boss:LeftFoot
Foot_R	boss:RightFoot
Forearm_L	boss:LeftArm
Forearm_R	boss:RightArm
Hand_L	boss:LeftHand
Hand_R	boss:RightHand
Head	boss:Head
Neck	boss:Neck
Pelvis_L	
Pelvis_R	boss:Hips

✓ Auto Scale

Use Pose: [✓ Rest] Current

Custom Naming Schemes:

Figure 5: The retargeting UI, with annotations for each section in the plug-in. In brief, 1 is used for choosing the source and target armature. Naive string-based bone matching can be done by pressing 2. Mapped bones can be seen as a list of row entries in 3. Graph matching is carried out on pressing 4. Manual mapping mode for fine-tuning can be activated by 5. 6 and 7 are used for choosing the colour scheme and activating the colouring mode. After the bone mapping is ready, 8 transfers the actual animation parameters. A detailed explanation for each can be found in individual sections.

Graph edit distance is a measure of similarity between two graphs. In fact, popular metrics such as string edit distance may be interpreted as graph edit distances between suitably constrained graphs. At a high level, every deletion, insertion and substitution of a node or an edge is penalised until the graphs are isomorphic to each other. For our task, we use the *optimal_edit_paths* function in the NetworkX library [10] for transforming the source armature into the target armature with minimum cost.[4]

The cost for insertion, deletion and substitution is set to 1. However, this may not be optimal since in certain scenarios, dropping a leaf bone (instead of a bone in the centre) might give a more natural mapping even when the edit distances for both mappings are identical. Consider two arm skeletons:

[4]Computing the optimal mapping is an NP-hard problem [5]. Hence, the timeout parameter in the library is restricted to 120 seconds since time taken grows exponentially with the number of nodes in the graphs.

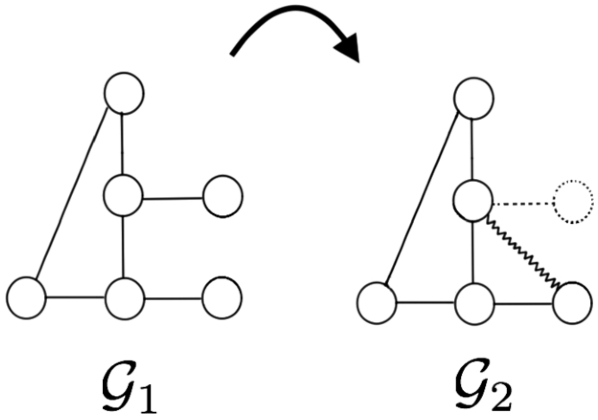
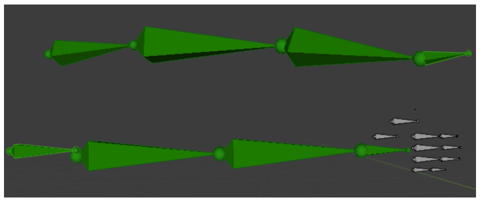


Figure 6: An illustration of Graph Edit Distance for two simple graphs. In the above figure, which is reproduced from [3], G_1 is transformed into G_2 and the edit distance is 3 units: 1 edge insertion, 1 edge deletion and 1 node deletion.

arm1 has 6 bones (name them a_i for i = 1,2,3,4,5,6) from the shoulder (a_1) to the fingertip (a_6) while arm2 has only 4 bones (name them b_i for i = 1,2,3,4) from the shoulder (b_1) to the the wrist (b_4) (no finger bones). Ideally, the first 4 bones from the shoulder of arm1 should be mapped to the 4 bones of arm2 ($a_1 - b_1, a_2 - b_2, a_3 - b_3, a_4 - b_4$). The graph edit distance for transforming arm1 into arm2 using this mapping will be 2 (2 deletions: a_5, a_6). However, the mapping ($a_1 - b_1, a_2 - b_2, a_3 - b_3, a_6 - b_4$) also has a cost of 2 (2 deletions: a_4, a_5). This shortcoming is depicted in Figure 7b. Any fixed rule for assigning different penalties to different bones is vulnerable to such ambiguities.

Another downside is that left-right ambiguities cannot be resolved by the graph matching algorithm. Currently, we use string matching to resolve these e.g. if leftHip is mapped to UpperLegR, we replace it with UpperLegL (and the mapping for rightHip is set to UpperLegR)


(a) Desired Mapping

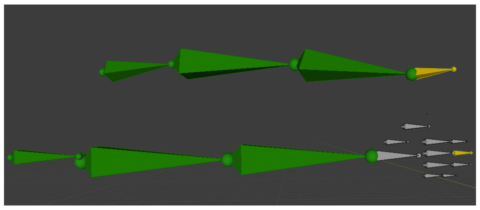

(b) Incorrect Mapping

Figure 7: Consider two different skeletons for the arm: the upper one terminates at the wrist while the lower one has a full set of fingers after the wrist. Note how the graph edit distances for transforming the upper arm into the lower arm are equal in both cases (2 deletions). In (b), the yellow fingertip in the lower arm gets mapped to the wrist bone in the upper arm. However, (a) is a more accurate mapping than (b) but such ambiguities cannot be resolved by a naive graph matching algorithm.

Implementation: Press the button labelled **Get Coarse Mapping**. The plug-in then expects 2 bones to be chosen by the user. To make the task simpler and reduce the chances of an incorrect mapping, we ask the user for the up bone (for both source and target armature) which is the root of the upper body of the skeleton. Both graphs are then split into two parts: the lower body rooted at the root of the armature and the upper body rooted at the user-chosen input. Graph matching is run separately on both halves and the UI table entries for mapping the bones are automatically populated with the generated best match.

3.2 Fine-tuning the mapping

After obtaining a coarse graph mapping using the algorithm mentioned in Section 3.1, we may need some further fine-tuning. This can be achieved by following the steps given below.

1. Press the **Enter Manual Mapping** button in the plug-in.
2. Choose one bone each from the source and target armatures which need to be mapped (ensure that both armatures are in pose mode).
3. Press **U** to complete the mapping. The row entry for the corresponding source bone will be reflected with the chosen target bone.

3.3 Visualisation of mapping

After generating a mapping using the graph matching algorithm, visualising the mapped bones can be helpful in understanding the errors in the predicted mapping. Colouring mapped bones in the source and target armature with the same colour can also assist the user in fine-tuning the mapping. The user is expected to give two inputs: the start and end bones in the source armature. The algorithm will pick all bones in the path from source to armature (there exists a unique path since it is a DAG) and pick the corresponding mapped bones in the target armature. Two colouring schemes have been implemented:

- Gradient: Colours of consecutive bones differ by a fixed value in each of the components, resulting in a clean gradient. (Figure 8)
- Random: Colours are picked randomly from the RGB spectrum. Helpful when number of bones is large, as a result of which distinguishing between consecutive bones becomes difficult when using the Gradient scheme.

Implementation:

1. Press the **Colour Mapping** button in the plug-in.
2. Choose the colour scheme from the drop-down menu in the plug-in.
3. Choose start and end bone in the source armature.
4. Press **U**. The bones will be colour-coded according to the mapping.

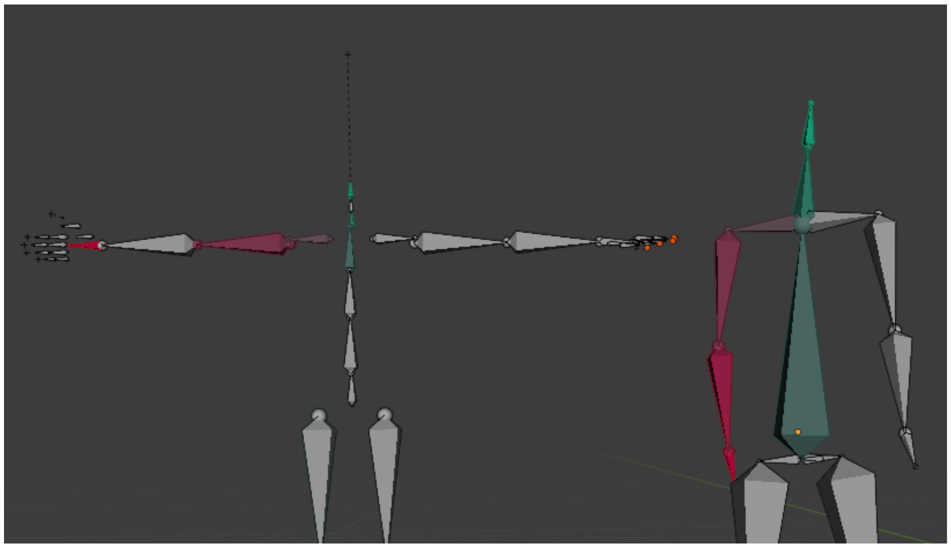


Figure 8: Gradient-based colouring of mapped bones. Notice that there is an error in the mapping: the bone between the wrist and the forearm in the armature on the left has been missed by the graph matching algorithm. Such errors can be quickly detected using colours.

4 Results and Discussion

A video demonstration of both stages of the pipeline can be found in this shared folder. A summary of the files in the shared folder is given below.

- *dance.mp4/walk.mp4*: RGB videos containing a single person dancing/walking. We wish to extract human pose from these videos and retarget it to a given character in Blender.
- *dance_vibe_result.mp4/walk_vibe_result.mp4*: The SMPL model generated by VIBE is overlaid on the person in the original RGB video. By comparing the subject with the generated SMPL mesh, we get a rough idea of the correctness of VIBE.
- *stage_1.mov*: Video walkthrough of the first stage of the pipeline. VIBE is used to extract human pose from each frame of the walk video.
- *stage_2.mov*: After using VIBE for extracting human pose from the walk video in stage 1, the procedure for calling the graph matching algorithm in order to align the bones of the source and target armatures is demonstrated in this video. The animations are transferred and played. We observe that although it is not perfect, the character does follow the general trajectory of the source. Fine-tuning of the mapping can alleviate this issue.
- Extras:
  - *stage_1_vibe_dance.mp4*: VIBE is used for extracting pose from the dance video.
  - *stage_1_mediapipe_dance.mp4*: MediaPipe is used for extracting pose from the dance video.

- *limitations.mp4*: In this video, we discuss the limitations of the current model. Due to differences in the armature of MediaPipe and the target character armature, no mapping can produce the desired animation transfer. More specifically, we find that the angle between the root bone and the spine is different in both aramtures: in the target armature, a fine-grained spine is aligned along the root bone direction. In contrast, the root and spine bones in the MediaPipe armature are at right angles. As a result, transferring animation for the root and spine bones leads to an inconsistency in the generated character animation. This can be clearly seen in the video.
- *colors.mp4*: This is a demonstration of how we can use colours to visualise the mapped bones. This is especially helpful for fine-tuning the mapping.

A performance comparison of VIBE and MediaPipe in terms of speed is given in Table 1.

Sr. No.	Filename	Descrciption	Backend	# Frames	Time take (s)	FPS
1	walk.mp4	Person walking casually	VIBE	69	70	0.98
2	walk.mp4	Person walking casually	MediaPipe	69	7.66	8.86
3	dance.mp4	Complicated sequence of dance steps	VIBE	186	156	1.14
4	dance.mp4	Complicated sequence of dance steps	MediaPipe	186	26	7

Table 1: Performance comparison of the two backends. MediaPipe, which was originally designed for real-time inference, outperforms VIBE by a factor of around 7. Note that the backends do no currently support GPU acceleration; all experiments have been run on a CPU.

Refer to Figure 9 and Figure 10 for a qualitative discussion of the accuracy of the retargeted animations for the walk video.

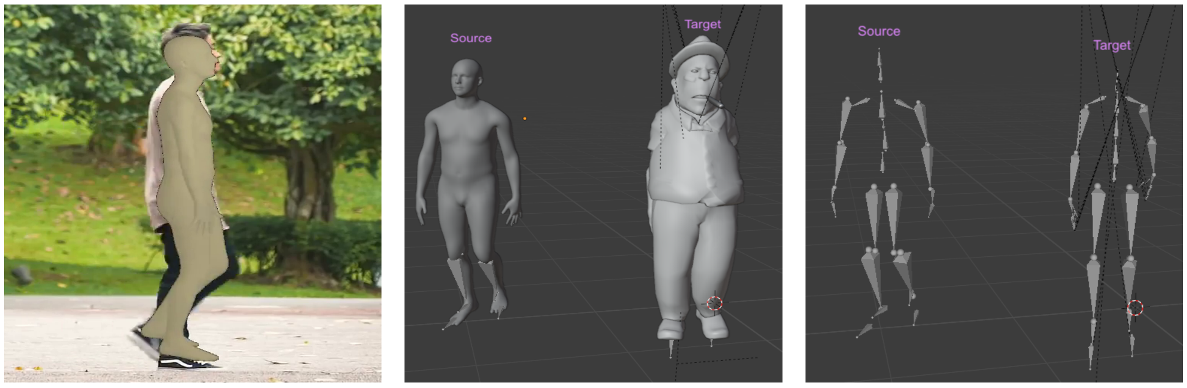


Figure 9: Results for frame 44 from the walk video are presented in this figure. We have the original pose in the RGB video on the left, both source and target characters after retargeting the animation in the middle and the armatures for source and target in the final image. The arms of the target character are occluded due to the t-shirt. However, we can see that the pose of both armatures are quite similar.

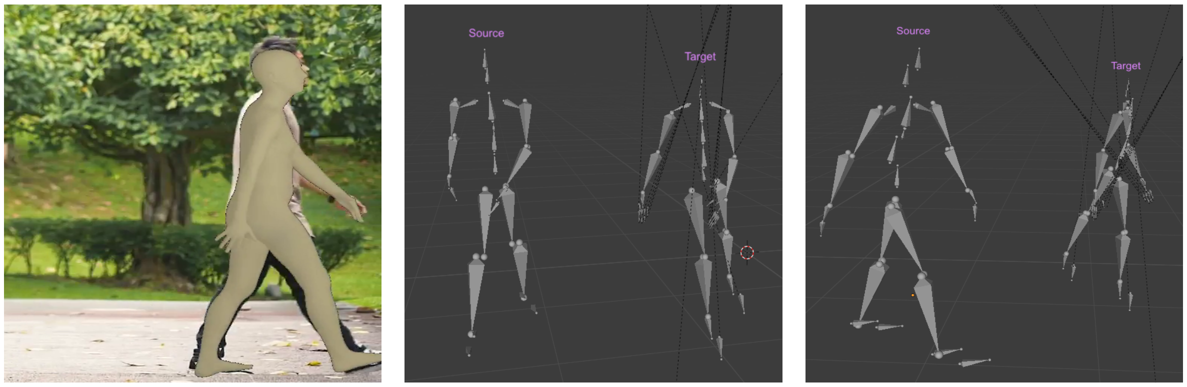


Figure 10: Results for frame 64 from the walk video are presented in this figure. We have the original pose in the RGB video on the left. The side view of the armatures (right) is more informative than the front view (middle) since the extent of the location of the arm along the direction of motion is more clearly visible in the figure on the right.

In Figure 11, the results on transferring the dance animation to a character rig are depicted.

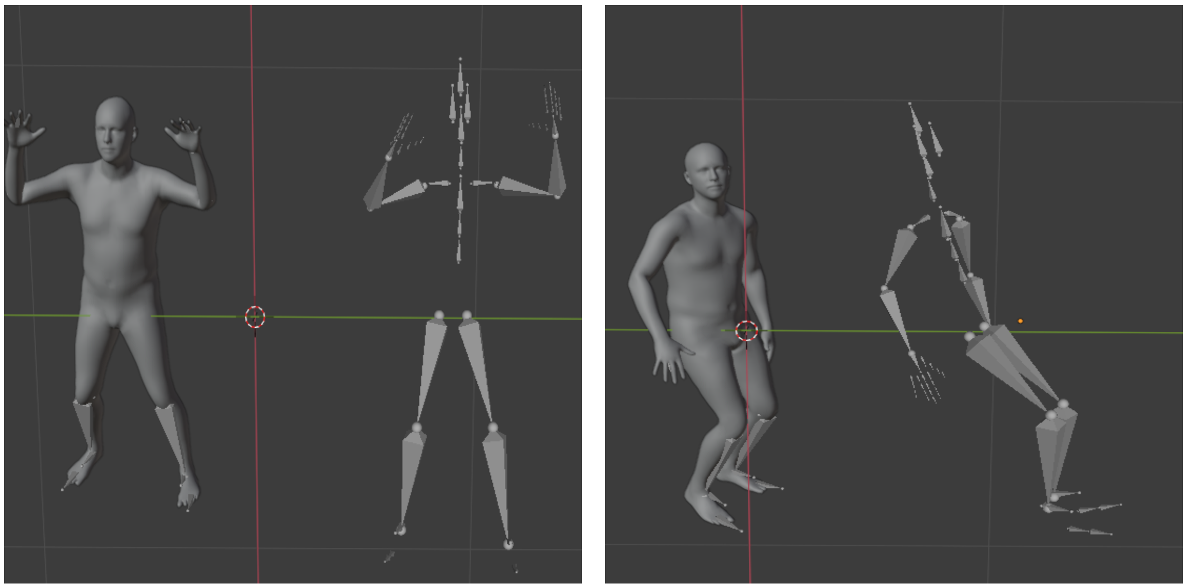


Figure 11: Comparison of the source and target armature pose for two randomly chosen frames from the dance animation. The two armatures are very similar in terms of the pose as desired. VIBE is used as the pose extraction backend. Manual fine-tuning was carried out due to errors in mapping of the wrists of the two armatures.

5 Graph Neural Network

The core limitation of the current methodology has been discussed in Section 4. When the source and target armatures have fundamentally different skeletons, no bone alignment will give accurate results. To overcome this, we propose a Graph Neural Network (GNN) based model which can retarget a given set of animation parameters to a significantly different target skeleton. The philosophy is as follows: the entire pose can be embedded in a fixed-dimensional vector (as demonstrated by architectures such as VIBE). Can we train a network to take this embedding as input and extract joint angles dynamically, depending on the hierarchy of the target armature?

Unlike other popular deep learning architectures such as CNNs, GNNs can operate on variable-sized inputs. GNN, as a framework, has been applied to a wide range of fields, from molecular biology to social network analysis [12]. We explore the possibility of using GNNs for our task of mapping animations from source armatures to structurally different target armatures.

5.1 Architecture Design

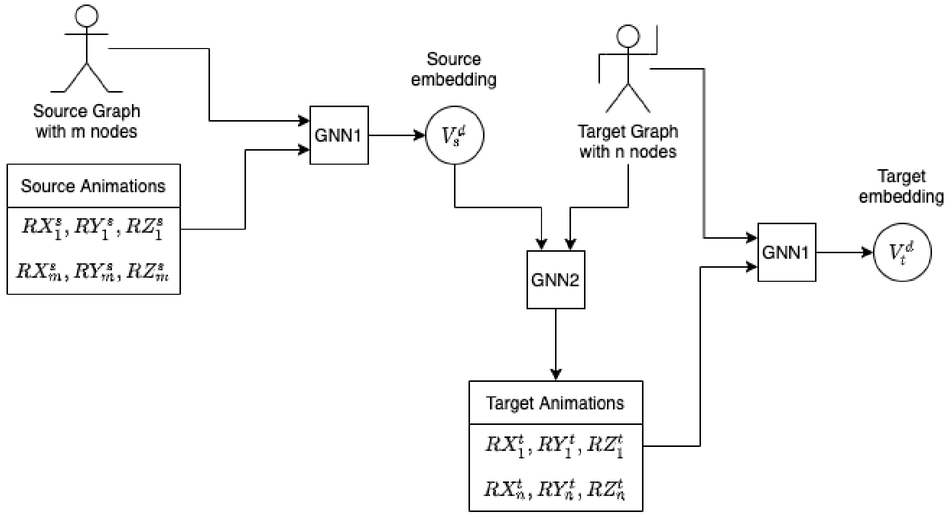


Figure 12: Proposed GNN architecture. RX_i^s (RX_i^t) refers to the rotation parameter of the i^{th} joint in the source (target) armature while V_s^d (V_t^d) refers to the d-dimensional source (target) embeddings. Note how the source and target graphs are structurally different. The nodes are represented by a feature vector described in Section 5.2.

The architecture of the proposed GNN framework is depicted in Figure 12.[5] There are two GNNs in our model:

- GNN1: Maps animation parameters of nodes to a high-dimensional pose embedding. This can be thought of as an encoder which aggregates local information into a global pose vector.

[5]The proposed model is trained to output the angles of the target joints only. The root translation is simply copied from the source to the target armature. This could lead to issues such as slipping. We plan to deal with it in future work.

- GNN2: Extracts angles from pose embedding and maps it to each node. It can be thought of as a decoder (or an inverse counterpart of GNN1).

The GNNs are composed of Graph Convolution, Fully-connected layers and an activation function. The number of layers is a hyperparameter which needs to be tuned.

5.2 Node features

To assist the GNN is identifying nodes, we generate a feature vector for each node. It has to be a fixed dimensional vector for processing it with Graph Convolutions e.g. in a social media graph, the node features could encoder age, gender, country, interests, likes/dislikes, etc. For our task, the feature vector should encode enough information that the location of the node, its relation to its neighbours and its role in the skeleton should be identifiable. We propose the following node features:

1. Distance from root (number of links)
2. Number of children
3. Max/Least distance to a leaf
4. 3-bit encoding for symmetric and non-symmetric bones: is_left, is_right, not_applicable e.g. left leg: 100, right arm: 010, spine: 001
5. Boolean indicating whether the bone is above the root or below the root
6. Offset from parent (x, y, z), normalised by torso length. This is especially important for cases where bones are missing and hence the model must be able to recover the appropriate pose angles from a coarse armature.

Attributes 1 and 3 are normalized by the number of nodes in the longest path from root to any leaf for making the attribute invariant to coarse and fine-grained armatures.

For a given pose, the 3-dimensional joint angle of each node is concatenated with the node features to get the final node representation which is subsequently fed to GNN1. For GNN2, the entire embedding is concatenated with the node feature vector of each node before feeding it to the GNN network.

5.3 Dataset

Deep learning methods are extremely data-hungry. As a result, any supervised learning paradigm is infeasible since there does not exist a dataset for our task. Self-supervised learning is a promising direction since it can operate with only an unlabelled dataset. The target labels are intelligently generated by masking some information from the samples. We propose the following methodology for generating the dataset for our task in a self-supervised fashion:

- Source armatures: Any large open-source MoCap dataset can be utilised for sampling the source armature and the corresponding animations. We use the CMU MoCap dataset [4] since it is free and contains a large variety of actions in 2,549 .bvh files.
- Target armatures: Take a few diverse skeletons which act as the base armatures. We can generate a huge target dataset by randomly dropping bones from these armatures. Care has to be taken to ensure that bones which exist in left-right pairs (e.g. arms) are dropped symmetrically.

To generate a batch of N training pairs:

- Sample N source frames from the CMU MoCap dataset. Each frame is converted into a source graph represented using node feature vectors (of size [n, f] where n is number of nodes in graph and f is number of node features) and the corresponding node angle parameters (of size [n, 3]).
- Sample N target armatures from the randomly generated target dataset (represented as a [m, f] feature matrix where m is number of nodes in the target armature).

We can form a 3-tuple of these to get an input of the form: (source graph, source animation, target graph). Currently, the GNN treats each frame of the video separately i.e. temporal information is not exploited. Thus, the RGB video problem is reduced to a frame-by-frame image pose transfer problem. We plan to use temporal information using sequence models such as GRUs in future work.

We can either generate tuples on-the-fly or create a fixed dataset offline and feed it to the GNN. The latter is preferred over the former to minimise convergence issues since in the former case, the model will potentially see new pairs in every batch and will lead to more noise in the gradient.

5.4 Loss

$$L = \sum_{s_i, t_i} d(s_i, t_j) - \sum_{s_i, s_j, i \neq j} d(s_i, s_j) \tag{1}$$

Here, s_i and t_i refer to the source and target embedding of the i^{th} example in the batch and $d(\cdot, \cdot)$ is any distance metric (discussed in more detail in Section 5.5).

By sharing GNN1 for generating the source and target embeddings and minimising the distance between them (term 1 in equation 1), we are constraining GNN1 to learn meaningful target angle parameters. However, there is a crucial flaw if we only minimise this distance: it could lead to embedding collapse. In other words, GNN1 can ignore the angles and simply output a constant embedding[6]. It will lead to a loss value of 0 but clearly, the model isn't learning any meaningful representation.

To overcome this, we need some *negative* samples. By maximising distance between embeddings which encode different poses, the model will be forced to learn meaningful representations in order to minimise the loss. This is accomplished by maximising the distance between the source embeddings in a given batch.[7] The second term in the equation 1 accounts for this distance.

5.5 Hyperparameters

The main hyperparameters of the architecture are:

- GNN1 and GNN2: Number of filters and number of layers of Graph Convolution (GraphConv). Currently, we have 2 layers with 96 filters in each GraphConv layer. ReLU activation is used.
- Embedding size: It forms a bottleneck in our model. Ideally, it should be as small as possible to prevent overfitting.

[6]To be more precise, it should be fed with negative source pairs of the same skeleton. Otherwise, it can maximise the distance between source pairs of different poses by simply encoding the node feature vectors into the pose embedding. Loss can still be pushed to 0 while ignoring joint angles.

[7]This is not entirely accurate because if two source embeddings do indeed represent the same pose, maximising the distance between their embeddings will be counterproductive. However, due to the sheer size of the dataset, the probability of such pairs will be negligible.

- Distance metric: ($\hat{s}, \hat{t}$ are the source and target embeddings respectively)
  - Cosine distance:
$$distance(\hat{s}, \hat{t}) = 1 - \frac{\hat{s} \cdot \hat{t}}{||\hat{s}|| \cdot ||\hat{t}||} \tag{2}$$
    In preliminary experiments, we found that normalisation of vectors by its norm to make it a unit vector is crucial; otherwise, the loss diverges.
  - Euclidean distance:
$$distance(\hat{s}, \hat{t}) = \sqrt{\sum_{i=1}^{n}(s_i - t_i)^2} \tag{3}$$
    where n is dimension of embedding.
- Batch size and learning rate: standard deep learning hyperparameters which need to be tuned for the task.

6 Conclusion

In this work, we presented a Blender plug-in for extracting human pose from videos and transferring it to any character armature in Blender. Two pose extraction backends have been incorporated into the pipeline (which can be further extended with better backends). Then, bones in the source and target armature are mapped using a graph matching algorithm. The optimal mapping is defined as the one which minimises the number of edit operations (insertion, deletion and substitution) required to transform the source armature into the target armature. The joint angles are then transferred to the corresponding bones for each frame.

On inspecting the results, we discovered the limitations of our models. In particular, the animations are way off when the target armature is different from the source armature. To tackle this, we proposed a graph neural network framework which can be trained in a self-supervised fashion.

Given below is a summary of the steps involved in converting a sequence of human poses in an RGB video to an animated character using the developed plug-in:

1. Choose the desired pose extraction backend and press **Select input video file** (Figure 4). Choose the .mp4 video file from the file navigator UI. The backend will process the RGB video and export the animated armature to Blender.

2. Choose your source and target armatures in section 1 of Figure 5.

3. Press **Extract Hierarchy** (button 4 in Figure 5) and choose the up bones in source and target armature (refer to Section 3 for an explanation). Then press **P**. On execution of the graph matching algorithm, the rows in section 3 of 5 will be populated.

4. Use buttons 5, 6 and 7 to fine-tune the mapping.

5. Press **Retarget Animation** (button 8 in Figure 5) to transfer the animation.

The entire code can be found on Github.

References

[1] "B3D MoCap Import". In: *Github* (). URL: `https://github.com/carlosedubarreto/b3d_mocap_import`.

[2] Valentin Bazarevsky et al. "BlazePose: On-device Real-time Body Pose tracking". In: *arXiv preprint arXiv:2006.10204* (2020).

[3] Utkan Onur Candogan and Venkat Chandrasekaran. "Convex graph invariant relaxations for graph edit distance". In: *Mathematical Programming* (2020), pp. 1–35.

[4] "CMU Graphics Lab Motion Capture Database". In: (). URL: `http://mocap.cs.cmu.edu`.

[5] "Graph Edit Distance". In: *Wikipedia* (). URL: `https://en.wikipedia.org/wiki/Graph_edit_distance`.

[6] Muhammed Kocabas, Nikos Athanasiou, and Michael J Black. "Vibe: Video inference for human body pose and shape estimation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 5253–5263.

[7] Matthew Loper et al. "SMPL: A skinned multi-person linear model". In: *ACM transactions on graphics (TOG)* 34.6 (2015), pp. 1–16.

[8] Camillo Lugaresi et al. "Mediapipe: A framework for building perception pipelines". In: *arXiv preprint arXiv:1906.08172* (2019).

[9] "Motion Capture Connector". In: *Github* (). URL: `https://github.com/pohjan/Motion-capture-connector`.

[10] "Optimal Edit Paths". In: *NetworkX* (). URL: `https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.similarity.optimal_edit_paths.html#networkx.algorithms.similarity.optimal_edit_paths`.

[11] "Rokoko Studio Live". In: *Github* (). URL: `https://github.com/Rokoko/rokoko-studio-live-blender`.

[12] Zonghan Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

A Installation

Blender comes with its own Python 3.9 installation. All scripts written as part of the plug-in are executed with this built-in version of Python. As a result, any additional packages required by the script must be installed via the pip package manager linked to Blender's Python. The steps given below for installation have been tested on macOS 11.0 and 12.0. Since Linux and macOS are based on UNIX, the steps will be similar with appropriate OS-dependent commands e.g. *brew* vs *apt* for installing packages.

1. Download the Blender installer from the official website and follow the official instructions to install Blender.
2. Set up an alias in Terminal for python and pip to Blender's built-in versions. For macOS, the paths are:
   - python: /Applications/Blender.app/Contents/Resources/2.93/python/bin/python3.9
   - pip: /Applications/Blender.app/Contents/Resources/2.93/python/bin/pip3.9
3. Install required packages using Blender's pip from requirements.txt.
4. If you encounter an error called *can't find OpenGL*, follow this link for help.

B Helpful Resources

This section is a list of miscellaneous resources which were helpful in developing the plug-in:

1. Getting started with plug-ins in Blender.
2. Fixing translation for VIBE backend.
3. Splitting the Layout interface into two or more windows. Can be used for displaying source and target armatures side-by-side.
4. Websites for free rigged characters:
   - https://www.mixamo.com//
   - https://free3d.com/3d-models/fbx-characters
5. Understanding the .bvh file format:
   - https://rdoc.info/gems/bvh
   - https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html