Pose mapping

This chapter describes how the 3D animation of Vere can be mapped onto the simplified kinematic model.

Objective

The complete ‘stick-figure’ that needs to be controlled is the simplified kinematic model of a human as is given in figure 7.

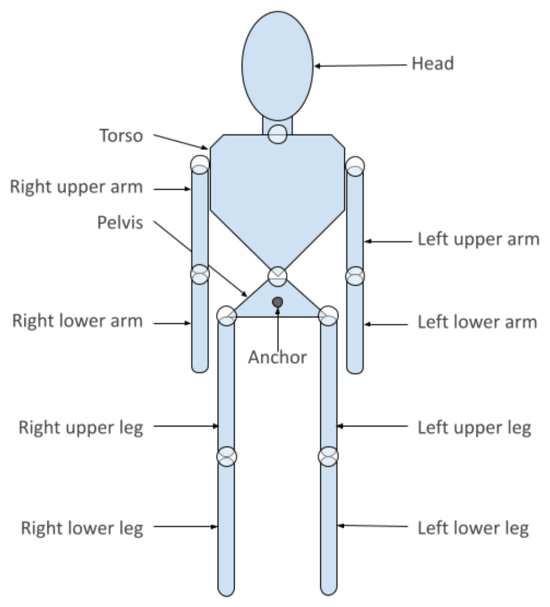


Figure 7: Kinematic model (i.e. ‘stick-figure’).

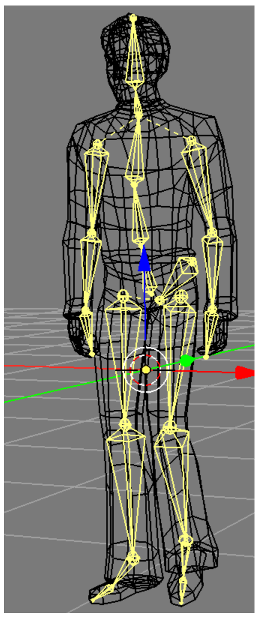
In the animation system the orientation of the segments in this kinematic model can be either controlled by the orientations from recorded data or those that are retrieved from the loaded file containing the animations as stored in a 3D model (see “3D animation file”). This chapter describes the latter, i.e. the ‘pose mapping’.

Segment mapping

In the kinematic model shown in figure 7 above, the segments are explicitly indicated (“Right upper arm”, “Right lower arm”, etc). From a practical measuring perspective, this makes sense because this corresponds to the fact that a sensor will be also attached to a specific *segment*. In the kinematic model the joints are not explicitly defined as identifiable nodes. A joint angle that might be required is calculated by the relative rotation of a segment in the coordinate system of its parent.

However, in the definition of the 3D model (stored in the glTF-file) this is done slightly differently and segments are not explicitly defined. Instead, the *joints* are defined and are connected in a hierarchical setup, i.e. the character is rigged as illustrated in the figure in an example in table 6.

Table 6: Rigged character where the joints define the model and mapped to segments.

Example rigged character	3D model	Kinematic model
	?	Head
	DEF-spine.003	Torso
	DEF-upper_arm.R	Right upper arm
	DEF-upper_arm.L	Left upper arm
	DEF-fore_arm.R	Right lower arm
	DEF-fore_arm.L	Left lower arm
	DEF-spine	Pelvis
	DEF-thigh.R	Right upper leg
	DEF-thigh.L	Left upper leg
	DEF-shin.R	Right lower leg
	DEF-shin.L	Left lower leg

Segment orientation

In the glTF-file, containing the 3D model, the animation is stored using a key-frame based description. This means that the actual orientation of a specific node at a certain timestamp is not explicitly available from the file itself, i.e. no frames are explicitly defined.

To retrieve the orientations (and to visualize the animation), the glTF-file is first loaded into a scene using the SceneKit SDK. After the scene is loaded, the animation can be played.

When playing the animation, we want the stick-figure to mimic the motions of the 3D model (as well as possible). For this we need to obtain the orientations of the nodes in the 3D model and map those on the segments of the stick-figure as given in table 6. To do this, we can create a `SCNSceneRendererDelegate` on which the `renderer(_:didRenderScene:atTime:)` method is called each time Scenekit finishes rendering the animation on the screen for the current timestep. In this method, the `worldOrientation` is retrieved for each relevant node as it currently appears onscreen. Because the animation is implicit (remember that only key-frames are defined), this information is available in the `presentation` property of the node.

For most segments this will work, however there is a problem with the shoulders because of the initial pose of the 3D character, which is defined as a T-pose (see figure 8) where the shoulders are *not* rotated.

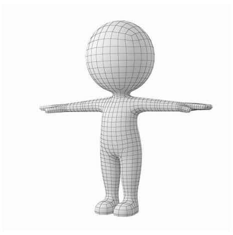


Figure 8: T-pose, i.e. all nodes are unrotated.

To compensate for this, a rotation of 45 degrees around the z-axis is first applied[2] to the segment before adding the rotation of the corresponding 3D node. The following code snippet shows how to do this given the orientation of the node in the 3D model.

```
let rotation = simd_quaternion( Double.pi/2, simd_double(0,0,1) )
let newOrientation = simd_mul( nodeOrientation, rotation )
```

[2] Scenekit uses a right-handed coordinate system with y-up and the z-axis pointing out of the screen.