The HmiGraphics packages

The HmiGraphics and HmiAnimation packages constitute a framework for 3D graphics and animation, based upon Java and OpenGL. This document describes the global structure of the various packages inside HmiGraphics and HmiAnimation. The subdivision into HmiGraphics and HmiAnimation is largely based on the project structure used by build tools like ant, Eclipse, and Netbeans. For an overview of the Hmi project structure, see [**?**]

See the javadoc.

1.1 Package overview

The current contents of HmiGraphics and HmiAnimation is as follows:

- hmi.animation: Deals with virtual objects, in particular animation of such objects. It defines a simple hierarchy of virtual objects, by means of VJoints, primarily used for skeleton based animation of human avatars. a virtual environment. Of course it is intended to be used in combination with the hmi.graphics packages, but in principle it can be build and used independently.

- hmi.graphics.collada: Deals with reading (and to some extent writing) graphics files according to the Collada standard. Apart from reading, it includes a sub package hmi.graphics.collada.scenegraph, for translating Collada descriptions into hmi.graphics.scenegraph descriptions.

- hmi.graphics.geometry: Utility class, dealing with some purely geometric algorithms.

- hmi.graphics.opengl: Deals with actual rendering OpenGL based 3D graphics. Although hmi.graphics.opengl is OpenGL based, it is still independent of the particular Java OpenGL binding or implementation. Currently there are two such OpenGL bindings: Jogl and LWJGL. Each of these has its own package (hmi.graphics.jogl, hmi.graphics.lwjgl), and one of them must be used in combination with hmi.graphics.opengl. hmi.graphics.opengl itself has a few sub packages:

  - hmi.graphics.opengl.state: Contains classes for dealing with the OpenGL state management.
  - hmi.graphics.opengl.geometry: Defines a few utility classes for rendering simple geometry, like spheres, boxes, and lines.
  - hmi.graphics.opengl.scenegraph: Defines the translation from hmi.graphics.scengraph structures to hmi.graphics.opengl structures.

- hmi.graphics.jogl and hmi.graphics.lwjgl are two packages that provide actual OpenGL bindings, relying on native code in the form of windows dll files or Linux so files. Both packages implement the GLRenderContext interface from hmi.graphics.opengl, and define a simple Renderer. Most of the JOGLContext and LWJGLContext code is generated, by means of a utility package hmi.graphics.gen.

- hmi.graphics.scenegraph: Defines scenegraph like structures for representing graphics data in a format independent of the external file format, but also independent of the actual render technology being used.

- hmi.graphics.util: The usual package of our favorite "utilities".

Currently, there are a few more "deprecated" and/or temporary packages:

- hmi.graphics.colladatest: Just for testing packages like collada, opengl, etcetera.

- hmi.graphics.gen: used for generating files within jogl and lwjgl implementations.

- hmi.graphics.render An attempt to create a platform neutral 3D renderer.

1.2 Skeleton Structures for avatars

An avatar that is to be used for (bodily) animation must posses a *skeleton* or *bones* structure. By this we do not mean a physical model or a visualization of a real skeleton, but rather a structure consisting *joints* and *segments* or *bones* that is to be used for the purpose of animation. See the picture in Figure 1.1 for an example skeleton. A skeleton consists of *joints*, like the Pelvis or Spine nodes in the picture, connected by means of *segments* (or "*bones*"), shown here by means of arrows connecting the joints. It will be clear that a skeleton is a rooted tree. (For the example skeleton, the root node is the Pelvis node)

1.2.1 Affine and linear transforms

Within this chapter by *linear* transform we mean an ordinary linear mapping for 3D space, represented by a 3×3 matrix M. A *translation* T is defined by a 3D translation vector t. When we want to make this vector it explicit, we write T_t for a translation operation. As is well known, a translation operation

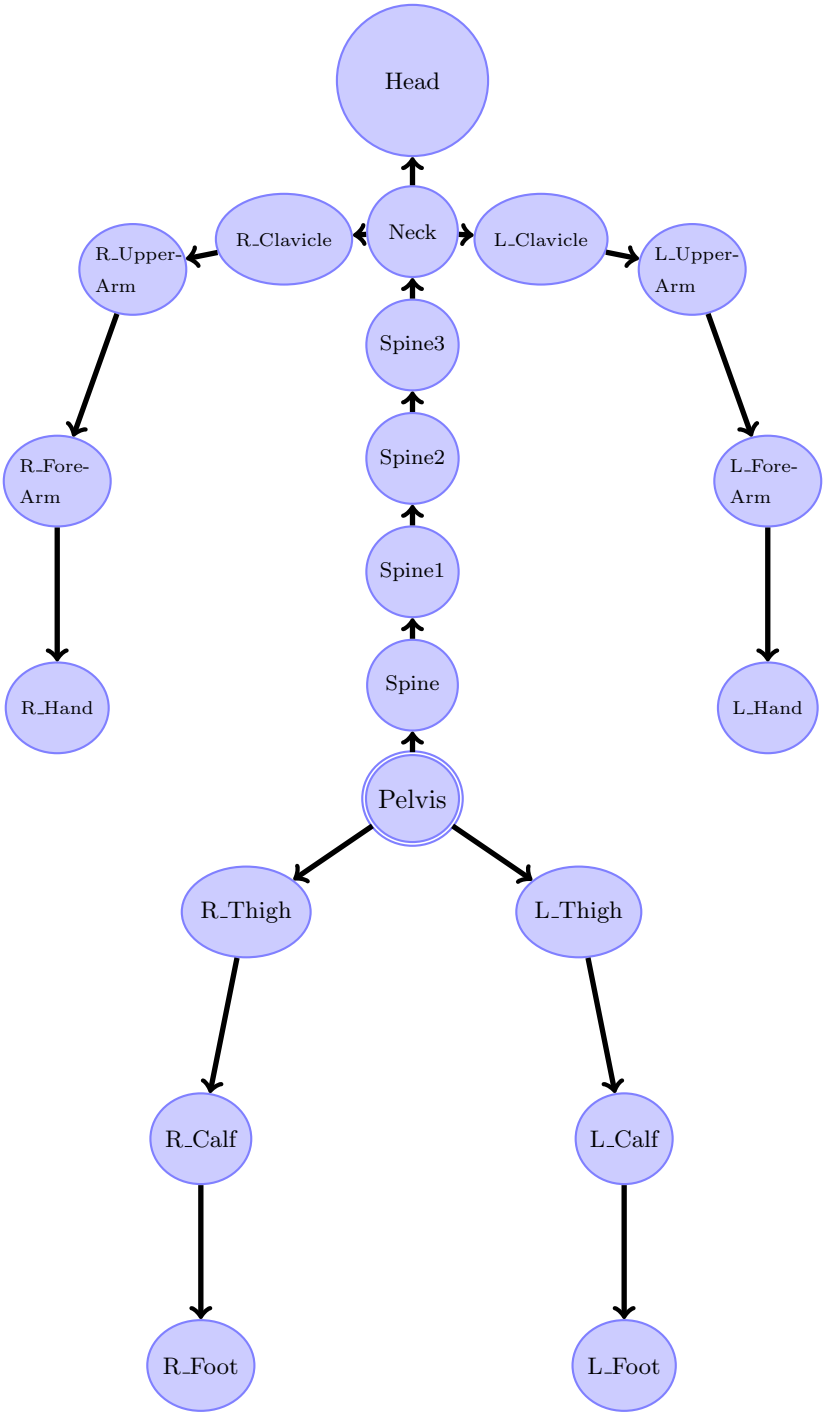

Figure 1.1: Example of a skeleton structure

T is not linear, so cannot be represented by a 3×3 matrix. We often need the combination A of a linear transform M followed by a translation T, so $A = T \circ M = T\,M$. Such a combination is called an *affine* transform. The combination in the reverse order, i.e. $M\,T = M\,T_t$ is also affine, since it is equivalent to the affine transform $A = T_{t'}\,M$, where $t' = M(t)$. As a consequence, a composition of affine transforms A_0 and A_1 is itself also affine:

$$A_0 A_1 = T_{t_0}\,M_0\,T_{t_1}\,M_1 = T_{t_0}\,T_{M_0(t_1)}\,M_0\,M_1 = T_{t'}\,M', \tag{1.1}$$
$$\text{where } t' = t_0 + M_0(t_1) \text{ and } M' = M_0\,M_1.$$

A (non-degenerate) linear transformation can be an *orthogonal* matrix Q which for 3D spaces is either a pure *rotation* R, or else a rotation combined with a *reflection*. (In the latter case it can represented, for instance, as $Q = R\,N$, where R is a pure rotation, and where N is a reflection. For instance, one may choose N to be $-I$. Another important class of linear transforms is that of *scaling*. In the simplest case, a scaling matrix S is just the (3×3) identity matrix multiplied with a *uniform scaling* factor s. A more complex case is when S is a diagonal matrix with three different scaling factors s_x, s_y, and s_z on the diagonal, which represents axis-aligned *non-uniform scaling*. In the most complicated case, a scaling matrix S is not even diagonal, but it would be diagonal in some rotated coordinate system. So it represents *non-unform scaling along rotated axes*, a situation sometimes called *skewing*. An important result here is that by means of so called *polar decomposition* any (non-degenerate) 3D matrix M can be decomposed as $M = Q\,S$, where Q is orthogonal, and where S is a scaling matrix. A nice property of polar decomposition is that Q is not just *any* orthogonal matrix, but that among all orthogonal matrices it is the one that is *closest* to M. We conclude that, for our purpose, we may assume that any linear 3D matrix M has the form $R\,S$ where R is a pure rotation and where S is a scaling matrix, possibly combined with a reflection operation. Our affine transforms A are therefore of the form $T\,R\,S$. Although A is not a 3D linear transform it *can* be represented by a 4×4 *homogeneous* transform matrix. So A has a matrix with the 3×3 matrix for $R\,S$ in the upper left part, the translation vector t for the translation T in the rightmost column, and a bottom row of the form (0, 0, 0, 1).

1.2.2 Skeleton transforms

The main use of skeletons is that they organize a collection of transforms, one transform for every skeleton joint J_i. Note: skeleton joints are sometimes called "bones". This terminology is slightly confusing since for others "bones" refer to the skeleton segments in between the joints.

For joint transformations we must distinguish between a *local transforms* D_i versus the *global transforms* A_i associated with each joint J_i. The local transform represents the transform caused by that single joint alone. The global transform A_i, is the combined effect of all joints on the path from the skeleton root up to and including the joint itself. For instance, for the joint called

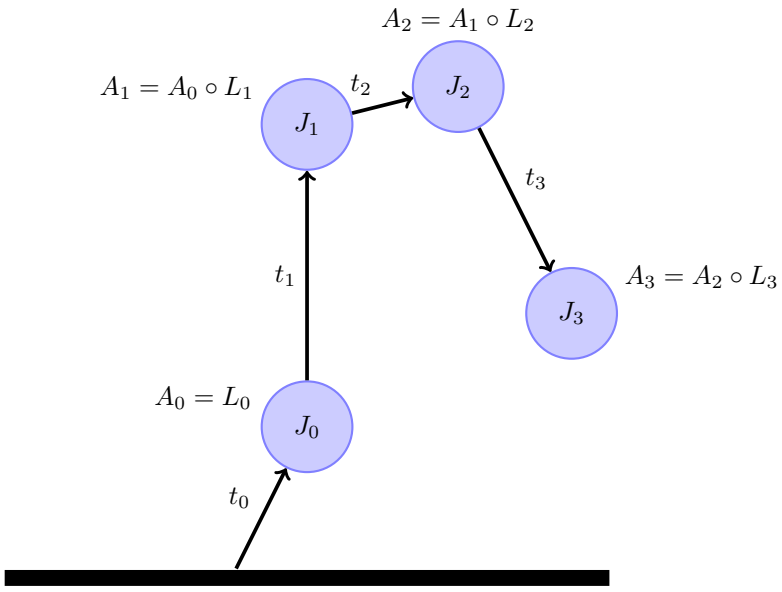

Figure 1.2: Part of a skeleton, with Translations t_i, Local Transforms D_i, and global transforms A_i

L_Forearm in the example skeleton, the global matrix $A_{\text{L\_Forearm}}$ represent the rotations and translations from the Pelvis joint, the various Spine joints, the Neck joint, the L_Clavicle joint, L_UpperArm joint, and finally the L_ForeArm joint itself.

The *local transform* D_i for a joint represents just the rotation R_i (and possibly scaling S_i) introduced by that joint alone, together with the translation that represents the vector t from the parent of that joint to the joint itself. For instance, for the L_ForeArm joint the local translation is represented by the vector from L_UpperArm to L_ForeArm. Note that each D_i is an affine transform.

The idea of a skeleton is that joint transformations are build up hierarchically, following the tree structure of the skeleton. So, when some joint J_i has joint J_p as its parent, then $A_i = A_p\, D_i$, where A_p is the transform associated with J_p, and where D_i is the *local* transform associated with joint J_i. For the special case of the root joint J_0, there is no parent, and we assume that in this case the transform A_0 is simply equal to the local transform D_0. For long chains of joint, we can thus factorize the transform A of the end of the chain into the local transforms of the joints on the chain. For instance, for the example skeleton we can calculate the affine matrix of, say, the neck joint as follows:

$$A_{Neck} = D_{Pelvis}\, D_{Spine}\, D_{Spine1}\, D_{Spine2}\, D_{Spine3}\, D_{Neck} \tag{1.2}$$

For the smaller scale example from Figure 1.3, we have:

$$A_3 = A_2\, D_3 = A_1\, D_2\, D_3 = A_0\, D_1\, D_2\, D_3 = D_0\, D_1\, D_2\, D_3. \tag{1.3}$$

We can decompose each local affine transform D_i into a linear transform L_i followed by a translation T_{t_i}, thus: $D_i = T_{t_i}\, L_i$. Using (1.1) we can rearrange equation (1.3).

$$\begin{aligned} &A_3 = T_t\, L \\ &\text{where } t = t_0 + L_0(t_1) + L_0\, L_1(t_2) + L_0\, L_1\, L_2(t_3) \\ &\text{and where } L = L_0\, L_1\, L_2\, L_3. \end{aligned} \tag{1.4}$$

So the net effect of a "chain" of local affine transforms, from skeleton root up to some skeleton joint, is equivalent to a single affine transform for that joint, which we have introduced before: it is the *global* transform for a joint.

1.2.3 Weight blending

Skeletons can be controlled and used by animation software; in those cases, we are only concerned with the local and global joint transforms. But skeletons are also used as an interface between animation engines and graphics rendering engines. In this case, the skeleton transforms are used to transform geometry that is used for rendering objects, in particular for rendering human avatars. The general idea is that there are one or more *meshes*, each consisting of many polygons, that model the geometric shape of body parts. There are two fundamentally different ways of doing this. One rather simple approach is to cut an animated object into parts, each of which is then animated independently, under the control of a single joint, dedicated to that part. For example, the "blue guy" avatar uses this approach, and uses separate meshes for limbs and other body part. The geometry associated with the elbow region of the body is transformed exclusively by the global affine transform for the elbow joint. One of the advantages of this simple approach is that all mesh vertices inside a single body part are transformed by the same affine matrix; a situation that suits the classical render pipeline of graphics hardware, as well as the OpenGL and DirectX interfaces for that hardware. Simple or not, an unavoidable problem with this approach is that an avatar body as a whole shows seams at places where different body parts connect.

An improved form of animating objects like human avatars is to use only *one seamless mesh*, and to use the skeleton transforms to deform by the following process: Each mesh vertex v is transformed under the influence of one or more joints $J_{i_0}, \ldots, J_{i_n}$, with *weights* $w_{i_0}, \ldots, w_{i_n}$ determining the relative influence of each joint. The exact set of joints and weights is unique for every individual vertex. Of course, one expects that most vertices will be controlled by a fairly low number of joints, typically one or two, and almost always less than four, all located in the neighborhood of the vertex. We would like to write down the transformation for some vertex v. We denote the global transform for joint J_i by A_i and, for simplicity, we assume that we have some vertex v that is influenced by joints $J_0, \ldots J_n$. The combined transform for v using weight blending is

defined as follows

$$v' = \sum_{i=0}^{n} w_i \, A_i(v) \tag{1.5}$$

1.2.4 Bind poses

Weight blending is an adequate animation method, but it requires both a suitable mesh as well as a skeleton that "fits" into the mesh. A problem here is that designers of meshes and designers of skeleton-based animations have slightly conflicting interests: An animation system would prefer a skeleton that is in some well defined neutral pose when all joint rotations are set to identity transforms. In that case all affine transforms reduce to mere translations. The HAnim standard for skeleton based animation, for instance, requires that in this situation the human avatar has a well defined pose where the body is upright, arms are pointing downwards, fingers are pointing downwards, the thumbs have a 45° degree orientation relative to the fingers, etcetera. The advantage of such a neutral pose is that an animation engine can put the avatar in some pose by setting well defined rotations within joints.

Designers of nice looking avatar meshes though, prefer a *different* pose, usually with the arms horizontally stretched, also known as the "T-pose". The main reason here is that graphics designers need to work on detailed graphic detail, and some areas like arm pits are difficult to reach and problematic when the avatar is in the HAnim neutral pose, rather than the T-pose. There are some variations of this "T-pose", for instance with the arms in a straight line but in a slightly lowered position.

The result is that meshes and skeleton structures usually do *not* automatically "match", and we need some process called "binding" the mesh to the skeleton. One of the steps in the binding process is that the skeleton, starting initially in its neutral pose, is put into "bind pose", by applying suitable rotations for its joints. For example, for our HAnim style skeleton, the bind pose for a T-shape avatar would include a 90° rotation for the shoulder joints, in order to get the arms into the T-pose. Other joints, for instance for arms and fingers, will likely have also non-identity rotations in the bind pose, although angles will not be as large as the 90° degree rotations for the shoulders. In other situations for instance in the case of exporting a virtual character from a tool like 3DSMax, the tool itself might use rather complicated transformations internally for binding a mesh to a skeleton. Unfortunately such more or less ad hoc bind poses show up when you export the mesh to some external format, for instance in the FBX format or the Collada format The message here is that,even if you are willing to design your character in HAnim pose, you still might have to deal with non-trivial bind poses.

After putting a skeleton in bind pose, the next step in the bind process is to assign blend weights to the vertices in the mesh. We won't discuss this (complicated) step here, and assume that you have used some tools to do this. For instance, most 3D modeling tools have a process called "weight painting"

that allow you to establish blend weights in a more or less intuitive way. In practice, assigning blend weight is a trial and error process.

We continue with the problems for our animation engine, caused by the difference in the neutral skeleton pose and bind pose. Say we would like our avatar to be in our neutral pose when all joint rotations are set to identity. It is clear that we must adapt our equation 1.5 for weight blending. Somehow, we must take into account the joint transforms that were used to get the skeleton into the correct bind pose. Let's assume that we know the values for (global) affine joint matrices in the bind pose, and let's call these matrices the *bind matrices* B_i. The intuitive idea is that we can use the *inverse* bind matrices B_i^{-1} to bring our mesh back into the neutral pose, and from that neutral pose, we bring it into the desired pose for some animation specified by (global) joint matrices A_i. This suggest our improved weight blending equation:

$$v' = \sum_{i=0}^{n} w_i \, A_i \, B_i^{-1}(v) \tag{1.6}$$

One way of seeing that this must be the "correct" equation is to put the avatar in its bind pose again, by choosing joint transformations A_i equal to the bind matrices B_i: for in that case the A_i and B_i^{-1} matrices cancel, and we have that for all vertices $v' = v$. Which is correct, since the mesh without transforms applied is, by definition, in the bind pose.

Since bind matrices are *fixed*, one might think that you can get rid of the inverse bind matrices in Equation 1.6 by applying these inverse bind matrices just once to the mesh, and store the resulting transformed mesh, which would now be in the (by animators) desired neutral pose.
This idea *would* work if every vertex v would be associated with just a single joint, so for the simple model from section 1.2.3, one could transform the various body parts by applying the unique B_i^{-1} for that part. (Note that for the rather special case where the bind pose is actual the same as the neutral pose, the bind matrices would reduce to pure translations of the form $T(C_i)$, where C_i is the center position of joint J_i. So in this particular case, applying B_i^{-1} boils down to a "shifting back to the origin" operation for the various body parts.)
Unfortunately, the idea breaks down when more than one joint influences some vertex v. Why? let's try. So we store, in an offline process, the mesh transformed into its neutral pose. The result is that we have transformed vertex v into a vertex v' defined by: $v' = \sum_{i=0}^{n} w_i \, B_i^{-1}(v)$. If we now apply Equation 1.5, where we replace v by our "corrected" vertices v', then we get the following vertex v'' as the result of applying a pose defined by joint matrices A_i:

$$v'' = \sum_{i=0}^{n} w_i \, A_i(v') = \sum_{i=0}^{n} w_i \, A_i(\sum_{j=0}^{n} w_j \, B_j^{-1}(v)) \tag{1.7}$$

This last equation 1.7 clearly does not yield the same results as equation 1.6. Fortunately, we can modify and simplify bind matrices if we are willing to adapt transformation matrices A_i from animations. We discuss this below.

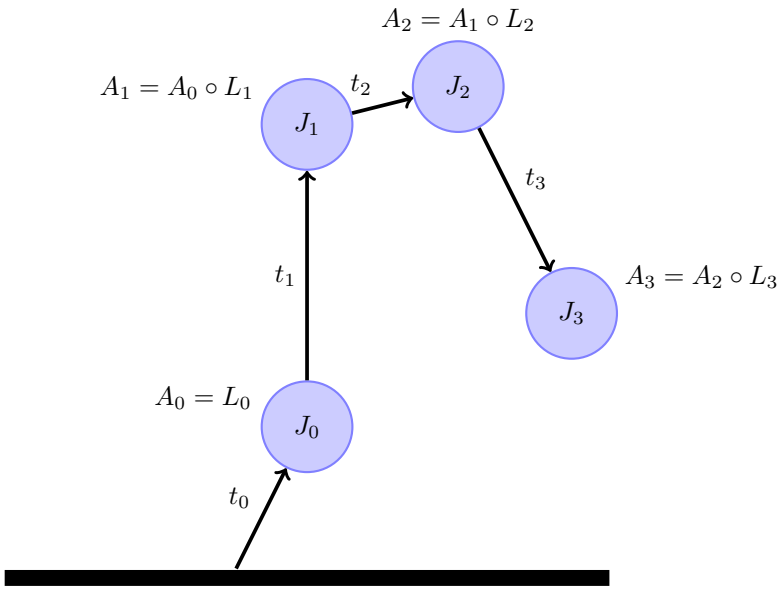

Figure 1.3: Part of a skeleton, with Translations t_i, Local Transforms D_i, and global transforms A_i

Adjusting bind matrices

We have seen the generic weight blending equation 1.6 above. There are various situations where we would like to modify the (inverse) bind matrices B_i^{-1}, thereby redefining the "neutral pose" for a character. One reason could be that the neutral pose as defined by some 3D modeling tool is inconvenient. For instance, the neutral position for a Collada export from 3DSMax defines a rather strange looking "neutral" position. Moreover, in such modeling tools the character mesh is often aligned with the Z-axis, (called the "up-axis") and for our animation engine we prefer a world where the Y-axis is the "up-axis". A final reason would be that we want to switch to a *new* neutral position like the one defined by the HAnim standard. In all such cases, three things have to happen:

1. The *(inverse) bind matrices* must be changed. This can be done by multiplying B_i^{-1} by matrices V_i that must be chosen in a suitable way for each of the situations mentioned above.

2. The local *rotations* for every joint to be used for various poses in animations must be adapted accordingly. This step is necessary only when existing animation data that was created for the "old" bind matrices. When new animation data has to be produced it might be much more convenient to work immediately with the "new" bind matrices. For instance, if we switch bind matrices that are suitable for HAnim then new animation data should use the HAnim pose as "neutral" pose. On the

other hand, existing animation data, for instance, data exported from the 3D modeling tool, will be based upon the "original" bind matrices, and so we have to convert the rotations from that data.

3. The local *translations* for every joint must be adapted to the new bind matrix. This can be done once, since translations and (modified) bind matrices are not changed by animation data. The only exception here that we allow is the "humanoid root translation" t_0, for the root joint J_0: this translation is sometimes modified in animations.

We discuss here first the generic case, where we multiply B_i^{-1} by arbitrary V_i. We assume here that V_i is just a rotation, and contains no translation. That means that we replace an inverse bind matrix $B_i^{-1} = U_i^{-1}\, T_{-C_i}$ by $B'^{-1}_i = V_i\, U_i^{-1}\, T_{-C_i}$. We consider some (arbitrary) pose based upon the original bind matrices, specified by a series of rotations $R_0, R_1, \ldots, R_n$. For this pose, joint J_i has a global transformation M_i of the form:

$$M_i = T_{t_0}\, R_0\, T_{t_1}\, R_1 \cdots T_{t_i}\, R_i\, U_i^{-1}\, T_{-C_i}$$

When we multiply the U_i^{-1} matrices with V_i, we must switch to a new set of translation vectors t'_i and new rotation/scaling matrices R'_i, as follows:

$$M'_i = T_{t'_0}\, R'_0\, T_{t'_1}\, R'_1 \cdots T_{t'_i}\, R'_i\, V_i U_i^{-1}\, T_{-C_i}$$

We require that $M'_i = M_i$ for all i, something we can realize by making the following choice for t'_i and R'_i:

$$t'_i = V_{i-1}(t_i)$$
$$R'_i = V_{i-1}\, R_i\, V_i^{-1}$$

Here, we use the convention that $V_{-1} = Id$. Imagine some virtual joint J_{-1} with identity transformations, as parent for the humanoid root node J_0. Note that because of this, the humanoid root translation t_0 need *not* to be transformed, which is in particular convenient when this translation is modified in an animation.

$$\begin{aligned}
&M'_i = T_{t'_0}\, R'_0\, T_{t'_1}\, R'_1 \cdots T_{t'_i}\, R'_i\, V_i U_i^{-1} T_{-C_i} \\
&= T_{t_0}\, R_0\, V_0^{-1}\, T_{V_0(t_1)}\, V_0\, R_1\, V_1^{-1} \cdots \\
&= T_{t_0}\, R_0\, T_{V_0^{-1}(V_0(t_1))}\, V_0^{-1}\, V_0\, R_1\, V_1^{-1} \cdots \\
&= T_{t_0}\, R_0\, T_{t_1}\, R_1\, V_1^{-1}\, T_{V_1(t_2)}\, V_1\, R_2\, V_2^{-1} \cdots = \cdots \\
&= T_{t_0}\, R_0\, T_{t_1}\, R_1\, T_{t_2}\, R_2 \cdots T_{t_i}\, R_i\, V_i^{-1}\, V_i U_i^{-1} T_{-C_i} = \\
&= T_{t_0}\, R_0\, T_{t_1}\, R_1\, T_{t_2}\, R_2 \cdots T_{t_i}\, R_i\, U_i^{-1} T_{-C_i} = M_i
\end{aligned}$$

(This was to be shown.)

Simplifying bind poses

The first application of bind pose modification is to *simplify* the bind matrices. The main reason for this step would be to get rid of overly complicated bind matrices introduced by tools like the Collada exported for 3DSMax: the original "neutral" pose looks incomprehensible. Before we start simplifying we have a more detailed look at bind poses and bind matrices. Let's assume that, in order to put the skeleton in the bind pose, we must apply *local* joint transformations of the form $T_{t_i} L_i$, where t_i is a local translation, and L_i is a local rotation (and possibly scaling). We consider the concatenation of such transformations along a path within the skeleton, starting at the humanoid root, and ending in some joint J_i. For convenience we assume here that this chain consists of joints $J_0, J_1, J_2, \ldots, J_i$. The global transform for joint i for the bind pose must be equal to B_i. For in that case, it will be canceled by the inverse bind matrix B_i^{-1}, and effectively we have *no* transformation of the mesh. And the pose where there is no transformation is (by definition) the bind pose. Therefore, we see that

$$T_{t_0} L_0 T_{t_1} L_1 \cdots T_{t_i} L_i = B_i = T_{C_i} U_i.$$

The left hand side of this equation can be rewritten as $T_t L$, where:

$$\begin{aligned} &L = L_0 L_1 \cdots L_i = U_i. \\ &t = t_0 + L_0(t_1) + \cdots + L_0 L_1 \cdots L_{i-1}(t_i) \\ &= t_0 + U_0(t_1) + \cdots + U_{i-1}(t_i) = C_i \end{aligned}$$

The easiest way to simplify bind matrices is to replace bind matricies of the form $B_i = T_{C_i} U_i$ by new bind matrices $B'_i = T_{C_i}$. So, basically, we want to drop the rotation and scaling parts U_i altogether, so that only a translations remains. Clearly, what must be done is to multiply B_i^{-1} by U_i, for in that case we have that $B'^{-1}_i = U_i U_i^{-1} T_{-C_i} = T_{-C_i}$, which is the desired inverse bind matrix. From the previous section we now see that we must adapt local translations t_i and rotation R_i as follows:

$$\begin{aligned} t'_i &= U_{i-1}(t_i) \\ R'_i &= U_{i-1}\, R_i\, U_i^{-1} \end{aligned}$$

From the equation above for $t = \cdots = C_i$, we see that $t'_i = C_i - C_{i-1}$ So the modified translation vector t'_i is simply the translation vector from the center position of joint J_{i-1} to the center position of joint J_i within the bind pose.

Reorienting your avatar

The next step that we want to discuss is how to re-orient mesh data. What we want is an avatar with its mesh and skeleton aligned with the Y-axis, looking into the positive Z-axis direction. Slightly more general, we want to apply some (linear) coordinate transform R to the mesh, skeleton, and animation poses.

For example, if we have some avatar with mesh and skeleton aligned with the Z-axis, and some pose where the shoulder joint rotates $-45°$ around the Y-axis, then after the coordinate transform we have a mesh and skeleton aligned with the Y-axis, and the same pose now has a shoulder joint rotation of $+45°$ around the Z-axis. In this case, the coordinate transform a is a rotation of $-90°$ around the X-axis.

Assume that we have some linear coordinate transform R. Usually, R will be a rotation, but it could include scaling. We want to transform mesh coordinates v into $v' = R(v)$, and then later on use our adapted skeleton to operate on this transformed mesh. The question now is: how to adapt the skeleton and animation poses.

Assume that some pose is described by local affine joint transforms D_i which describe local joint pose like the shoulder joint in the example above. Within the new coordinates, the effect of D_i will be achieved by $D'_i = R\,D_i\,R^{-1}$. (Just examine the effect of D'_i on some "new" coordinate of the form $v' = R(v)$. The result is that $D'_i(v') = R\,D_i\,R^{-1}(R(v)) = R\,D_i(v)$.) Of course this is to be expected: $R\,D_i\,R^{-1}$ is just the standard linear algebra result for how matrices change under coordinate transforms. Now we must take into account that the local transforms D_i that describe some pose are affine transforms of the form $D_i = T_{t_i}\,L_i$ where t_i is the (fixed) local skeleton translation from the parent of joint J_i to J_i itself, and where L_i is the (changing) rotation matrix for joint J_i. We can rewrite the D'_i:

$$D'_i = R\,D_i\,R^{-1} = R\,T_{t_i}\,L_i\,R^{-1} = R\,T_{t_i}\,R^{-1}\,R\,L_i\,R^{-1}. \tag{1.8}$$

The rightmost three factors, i.e. $R, L_i\,R^{-1}$, are just the transformed rotations L'_i, adapted to the new coordinate system. For instance, if L_i would be the $-45°$ around the Y-axis from the example above, and R would be the coordinate transform defined by a $-90°$ rotation around the X-axis, then $R\,L_i\,R^{-1}$ is actually the $+45°$ around the Z-axis, as expected.
The left most factors, i.e. $R\,T_{t_i}\,R^{-1}$ are the modified translations. We can simplify this considerably:

$$R\,T_{t_i}\,R^{-1} = T_{R(t_i)}\,R\,R^{-1} = T_{R(t_i)} \tag{1.9}$$

Finally, we can see how this all fits together, when we apply an linear transform R to a mesh that is being deformed by means of weight blending, specified by Equation 1.6. We assume that $A_i = T_{t_i}\,L_i$, that $B_i^{-1} = U_i\,T_{-C_i}$. We denote by L'_i the transformed L_i, that is, $R\,L_i\,R^{-1}$. Similarly we define ${U'}_i^{-1} = R\,U_i^{-1}\,R^{-1}$. Then the result of applying R yields the following transformed blend equation:

$$R\sum_{i=0}^{n} w_i\,A_i\,B_i^{-1} = \sum_{i=0}^{n} w_i\,R\,A_i\,B_i^{-1} \quad \text{, where}$$

$$
\begin{aligned}
&R\,A_i\,B_i^{-1} = \\
&R\,T_{t_0}\,L_0\,T_{t_1}\,L_1 \cdots T_{t_i}\,L_i, U_i^{-1}\,T_{-C_i} = \\
&R\,T_{t_0}\,R^{-1}\,R\,L_0\,R^{-1}\,R\,T_{t_1}\,L_1 \cdots T_{t_i}\,L_i\,U_i^{-1}\,T_{-C_i} = \\
&T_{R(t_0)}\,L_0'\,RT_{t_1}\,L_1 \cdots T_{t_i}\,L_i\,U_i\,T_{-C_i} = \cdots = \\
&T_{R(t_0)}\,L_0'\,T_{R(t_1)}\,L_1' \cdots T_{R(t_i)}\,L_i'\,U_i'\,T_{R(-C_i)}\,R
\end{aligned}
$$

The remaining R at the right end of this last formula is the transform to be applied on the mesh.

We conclude that, apart from transforming the mesh by means of applying R to the vertices, we need to adapt translation vectors t_i, rotation/scaling matrices L_i, and the as follows:

$$
\begin{aligned}
t_i' &= R(t_i) \\
L_i' &= R\,L_i\,R^{-1} \\
C_i' &= R(C_i) \\
U_i' &= R^{-1}U_iR
\end{aligned}
\tag{1.10}
$$

Since the translations, and the bind matrices are fixed, they can be calculated before we start rendering. This is very similar to the transformations for simplifying the bind matrix. What is different is that the translation part of the bind matrix is also modified

Redefining the avatar neutral pose

The result of the previous sections is an avatar where the original bind pose is also the neutral pose, that is, when all local rotation matrices are set to identity, the avatar will assume a pose equal to the bind pose. We would like to change this, and define some other pose, say the HAnim pose, to be the neutral pose. This problem can be split into two steps:

1. First, find out how to put the avatar in the HAnim pose,
2. Second, define that pose as the neutral pose, by adapting transformations and bind matrices.

We start with the second step, so, we assume that we have a a pose defined by local rotations (and possibly scaling) H_i that define a pose that we would like to set as the neutral pose. Using the existing bind matrices and translation vectors, the global transform for joint J_i for this new neutral pose is:

$$T(t_0)\,H_0\,T(t_1)\,H_1\,T(t_2)\,L_2 \cdots T(t_i)\,H_i\,B_i^{-1}$$

We would like to modify the bind matrices in such a way that we can represent the same pose using *identity* matrices replacing the H_i matrices. This

can be achieved by using new inverse bind matrices $B_i'^{-1}$ and new translation vectors t_i' of the form

$$\begin{aligned} &B_i'^{-1} = V_i B_i^{-1} \\ &t_i' = V_{i-1}(t_i), \text{ where} \\ &V_i = H_0 H_1 \cdots H_i \end{aligned} \tag{1.11}$$

Also, an existing (arbitrary) animation pose, defined by rotations/scalings $R_0, R_1, \ldots, R_n$, must be replaced by an adapted pose $R_0', R_1', \ldots, R_n'$ where:

$$R_i' = V_{i-1} R_i V_i^{-1}.$$

For the *first* step, i.e. putting some skeleton in the HAnim pose, one can use various techniques; in the end, all that count is that our VH is in the desired pose. For a pose like the HAnim standard, it is often useful to have a method that aligns specified segments of the human body with direction vectors *dir*. For example, the HAnim standard requires that upper and lower arm are "hanging downwards", i.e. are aligned with the direction of the (negative) Y-axis. In this case, we would align those segments with a direction vector $(0, -1, 0)$. Let us assume that we have determined that the current situation of the body is such that we have two joints, a parent and a child, and that the segment in between those two joints is currently aligned with some vector a. It is easy to establish a quaternion that rotates a into *dir*. (We assume that both a and *dir* are normalized vectors, i.e. $|a| = |dir| = 1$: Let $h = ((a + dir)/|a + dir|)$. (That is, h is the normalized half-vector in between a and *dir*.) Now q is simply $(a \cdot h, a \times h)$. We are not done yet, since we now know how to rotate our segment *after* that segment has been rotated by the skeleton transforms. Let us denote the latter rotation by a quaternion r. Then our construction delivers a quaternion q as above, with the property that $q\,r$ is a rotation that puts our segment in the desired orientation. Here, r itself is a product like $r = r_0 r_1, \cdots r_n$, where the quaternions r_i are the current (local) rotations of the skeleton joints, starting at the root, up to and including the parent joint rotation. What we want instead is a quaternion s with the property $q\,r = r\,s$, for in that case, we can replace the local quaternion r_n by $r_n\,s$, i.e. a post multiplication of the local parent joint rotation r_n with the s quaternion. From the equation it follows that $s = r^{-1}\,q\,r$. (Basically, this is is q rotated by r^{-1}.)