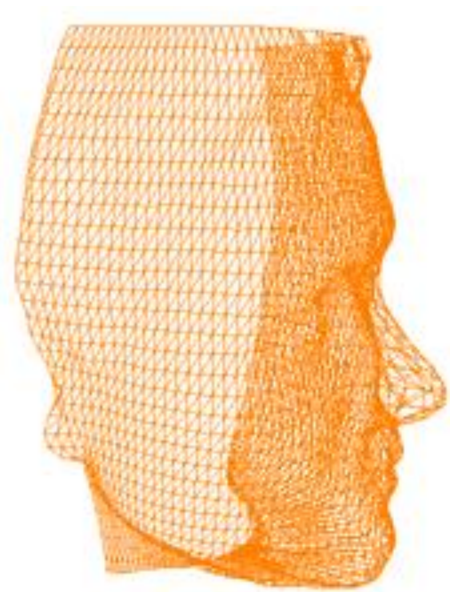
COSC422 Advanced Computer Graphics

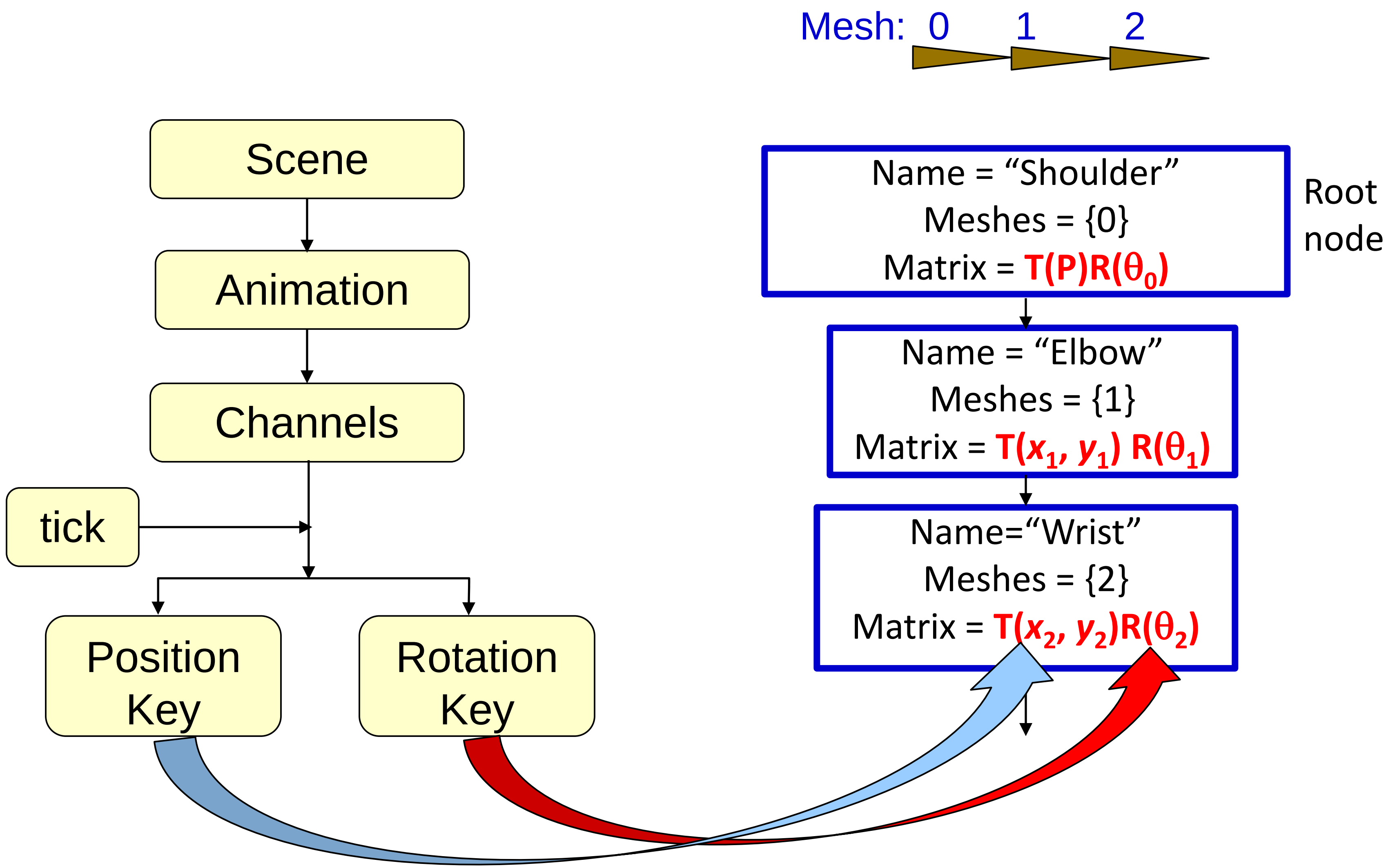


9 Character Animation

Semester 2
2019

R. Mukundan (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

Recap: Animating a Skeleton



Transformation Hierarchy

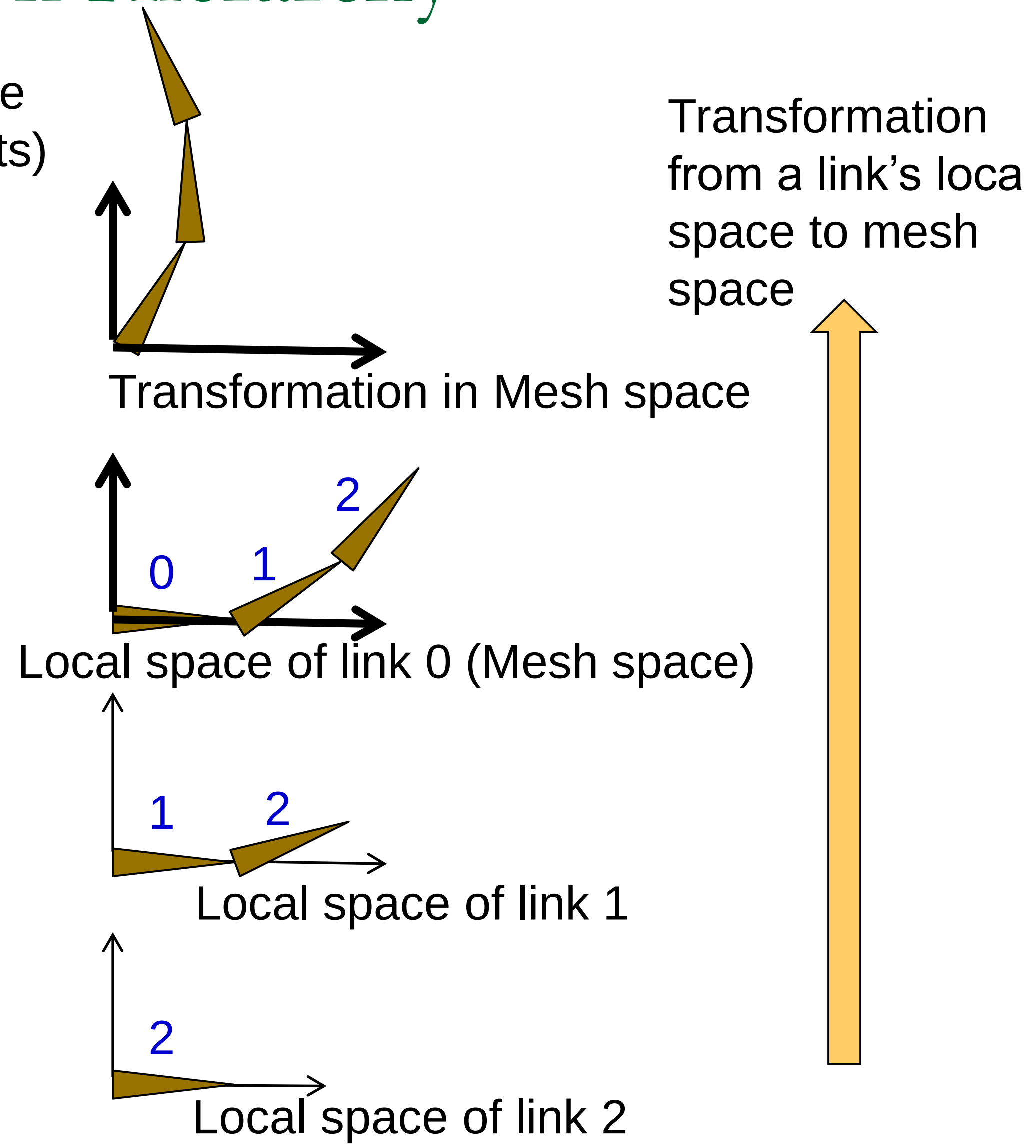
Each node in the tree is a mesh node
(It contains references to mesh objects)

Root node

Name = "Shoulder"
Meshes = {0}
Matrix = $\mathbf{T(P)R(\theta_0)}$

Name = "Elbow"
Meshes = {1}
Matrix = $\mathbf{T(\mathit{x}_1, \mathit{y}_1)\ R(\theta_1)}$

Name="Wrist"
Meshes = {2}
Matrix = $\mathbf{T(\mathit{x}_2, \mathit{y}_2)R(\theta_2)}$

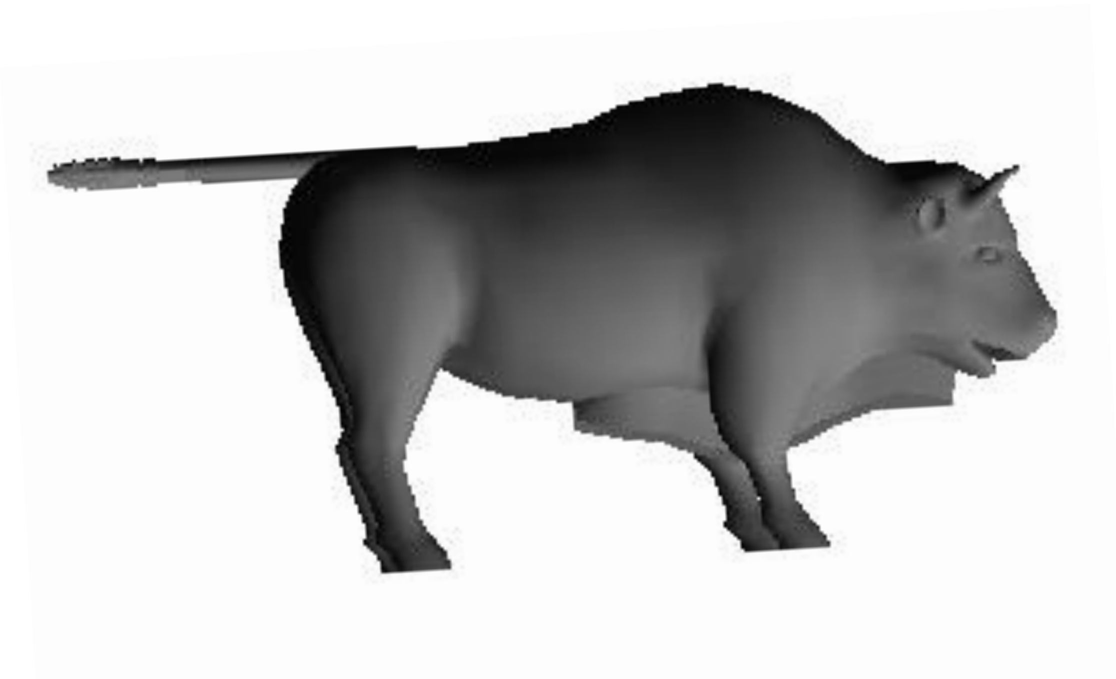


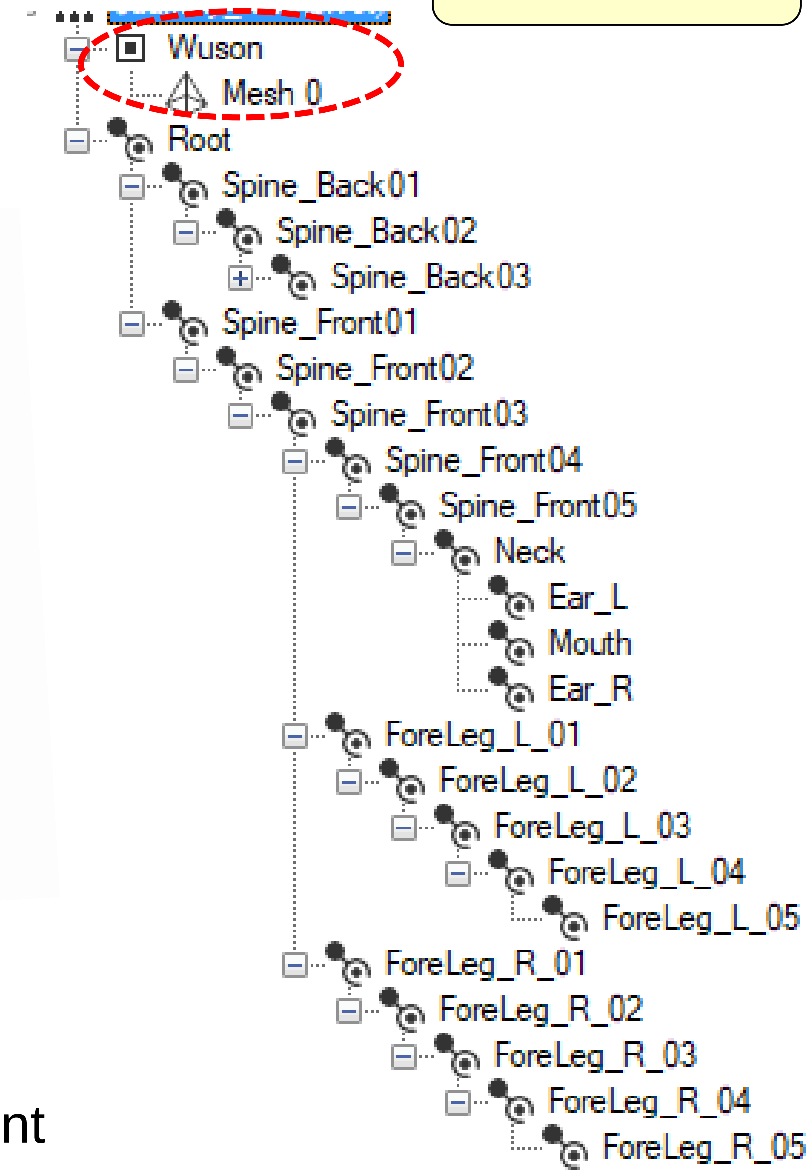
Animating a Character Mesh

- The method on the previous slide works only for a **segmented skeletal mesh**
  - Each node has a mesh object which can be independently transformed using a TR product matrix.
  - Each mesh is initially positioned at the origin (defined in the local coordinate space) so that it can be rotated in-place, and then translated relative to its parent.
  - Character mesh models are **not** specified as above. Usually, there will be only one single mesh for the whole model. Even if the model consists of multiple mesh objects, all meshes will be defined in a single mesh space.
  - The scene graph for a character model usually consists of a separate joint hierarchy (joints do not contain mesh definitions)

Example 1: wuson.x

open3mod

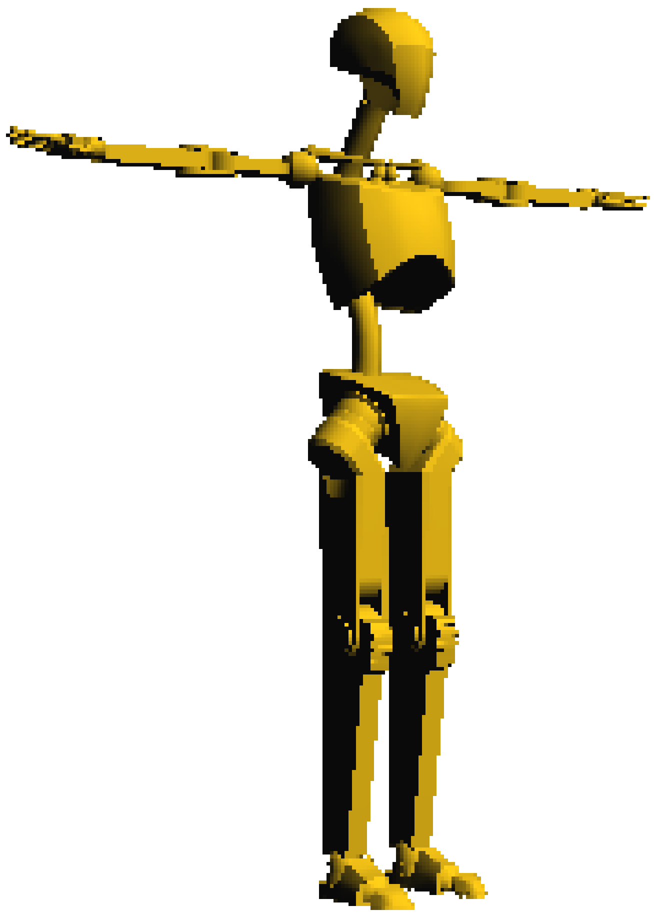


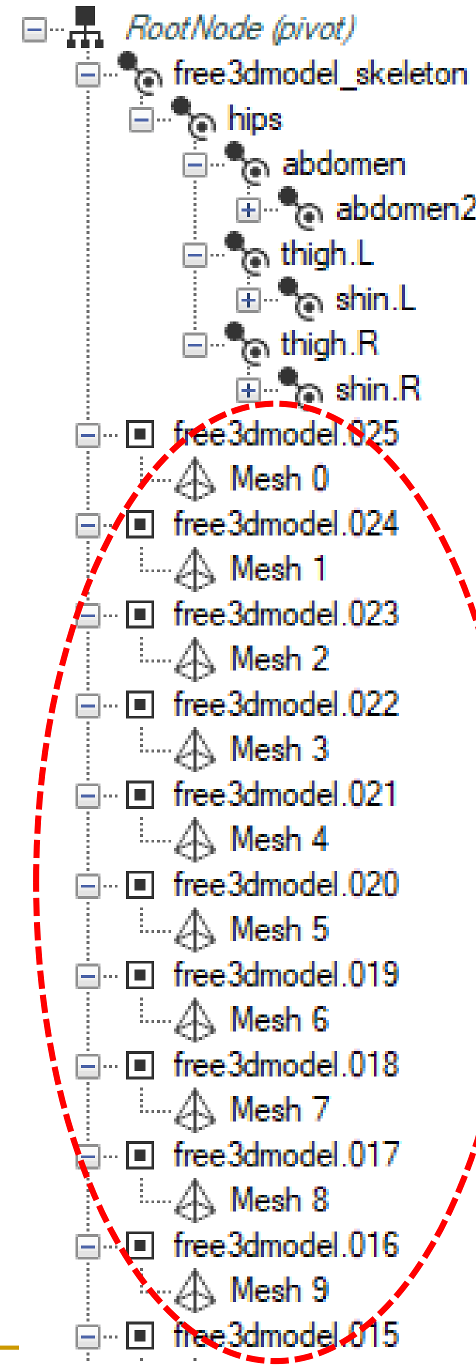


Number of animations = 2

Notation: = Mesh Node = Joint

Example 2: Mannequin.fbx





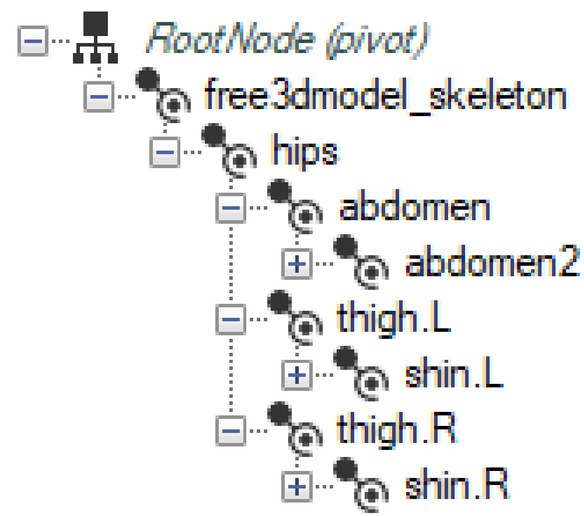
Number of animations = 0

Example 2 (contd.)

For the model on the previous slide, the animations are stored in separate files (eg. jump.fbx, run.fbx etc.).

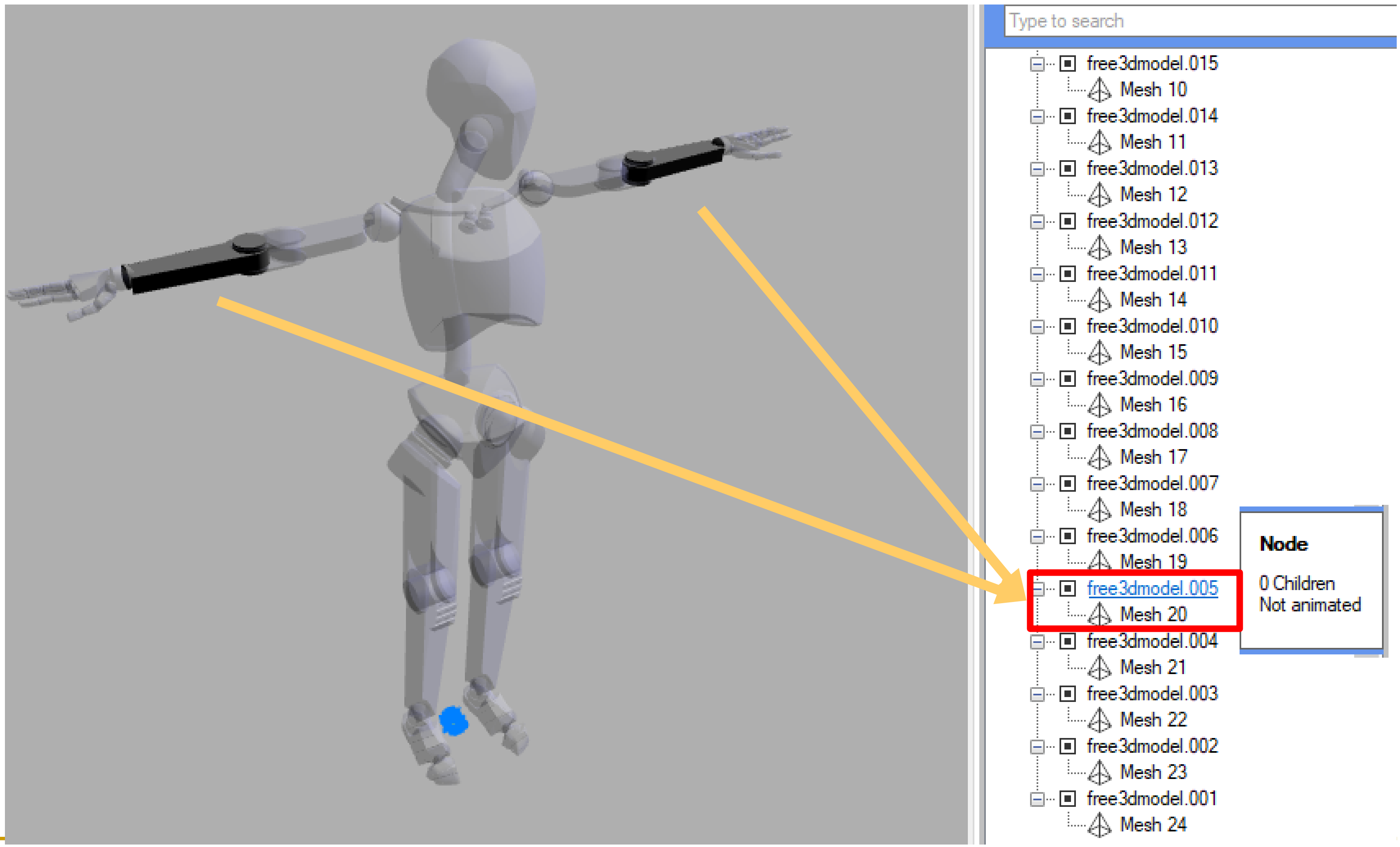
Each motion file contains

- Joint hierarchy
- Animation data (channels)
- No mesh data

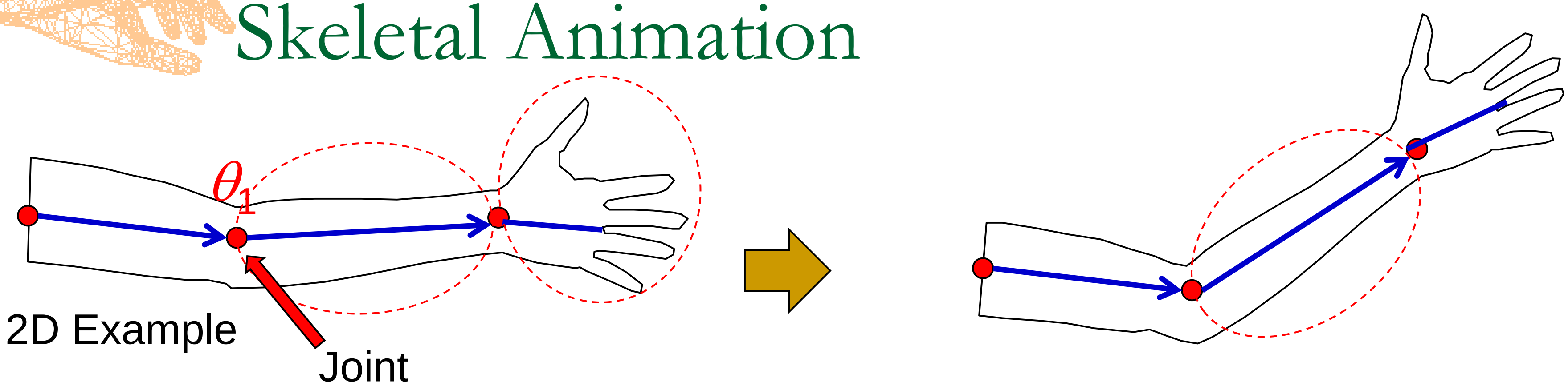


Example 2 (contd.)

A mesh object may represent multiple disconnected parts of a character's body, and cannot be transformed as a whole.

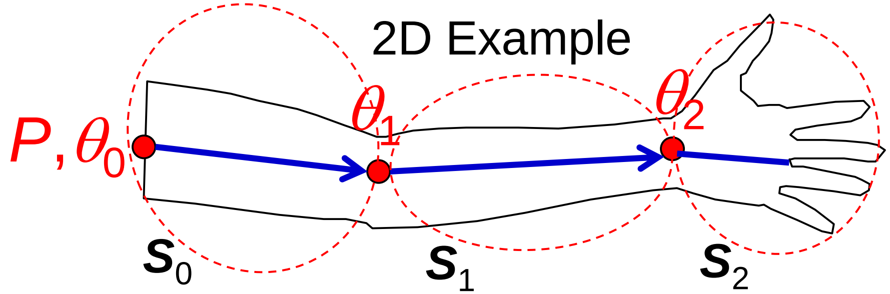


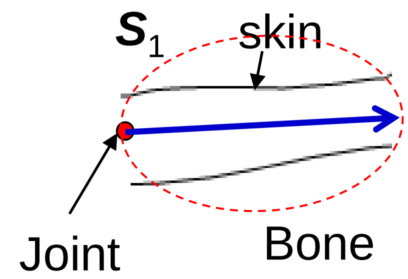
Skeletal Animation



- Joints are pivot points about which parts of a mesh are rotated by angles stored in keyframes.
- In order to rotate a part of a mesh, we need to segment the mesh, identifying sets of vertices that move together as a unit.
- We can then move the vertices to the joint's local space (where the pivot point is at the origin), apply the rotational transformation, and move them back to the mesh space.

Bone

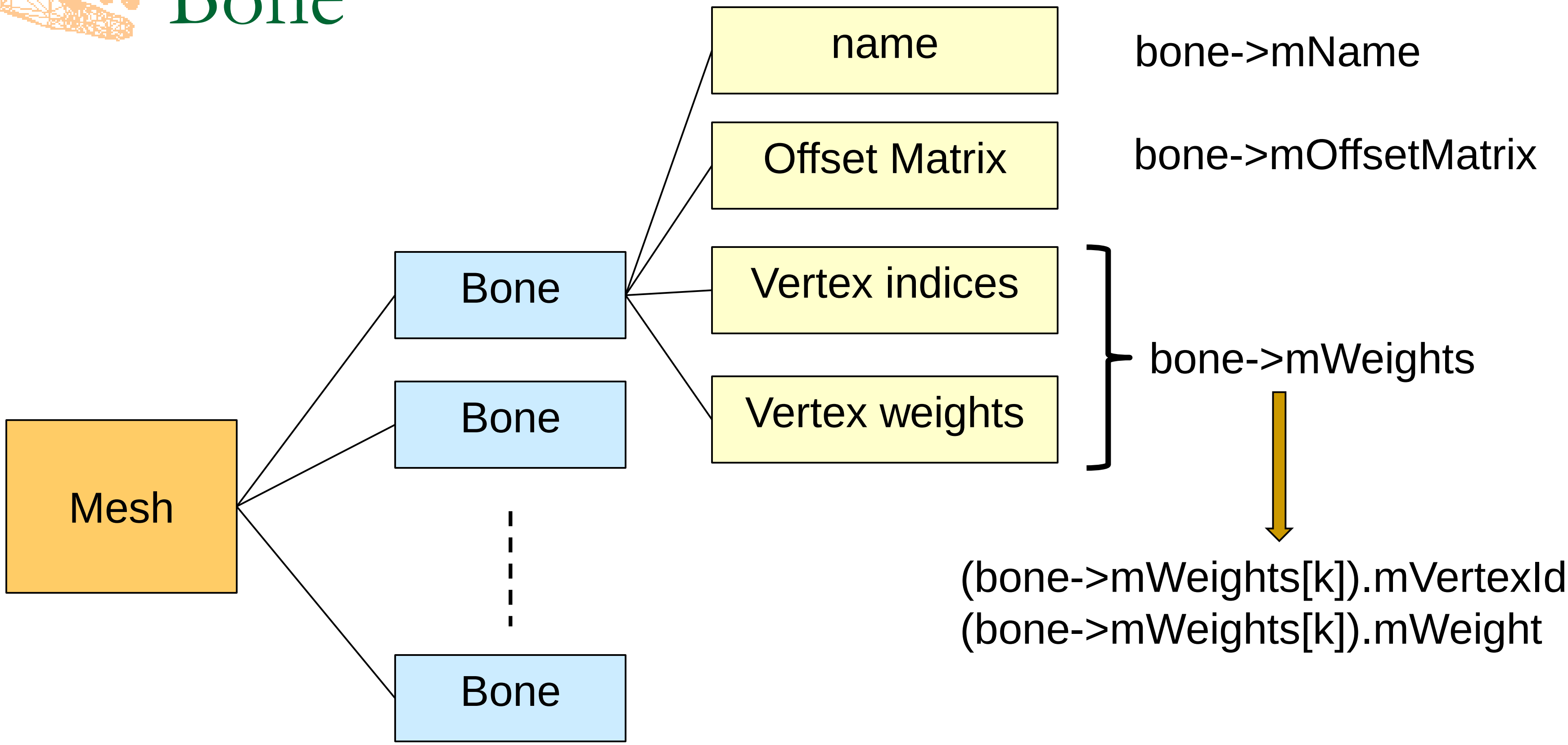




Each bone defines a mesh segment using

- a set of indices of mesh vertices specifying the region of influence (skin) of the bone
- an offset matrix using which we can transform the mesh vertices (skin) to joint's space, where the joint (point of rotation of the bone) is at the origin.
- Corresponding to each joint, there exists a bone with the same name.

Bone



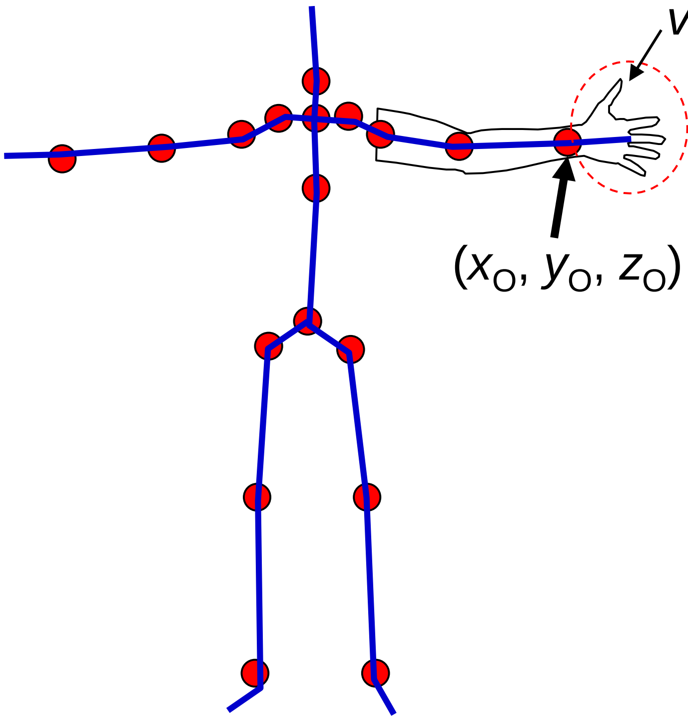
aiBone bone = mesh->mBones[i]

- Bones are associated with a mesh object. Each bone represents a collection of vertices of that mesh.
- Bones do not have a hierarchical structure.

Offset Matrix

A mesh vertex

v

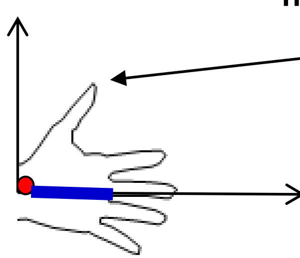


(x_O, y_O, z_O)

Bone → Offset Matrix:
$T(-x_O, -y_O, -z_O)$

The offset matrix transforms vertices of that bone to the local space of the joint

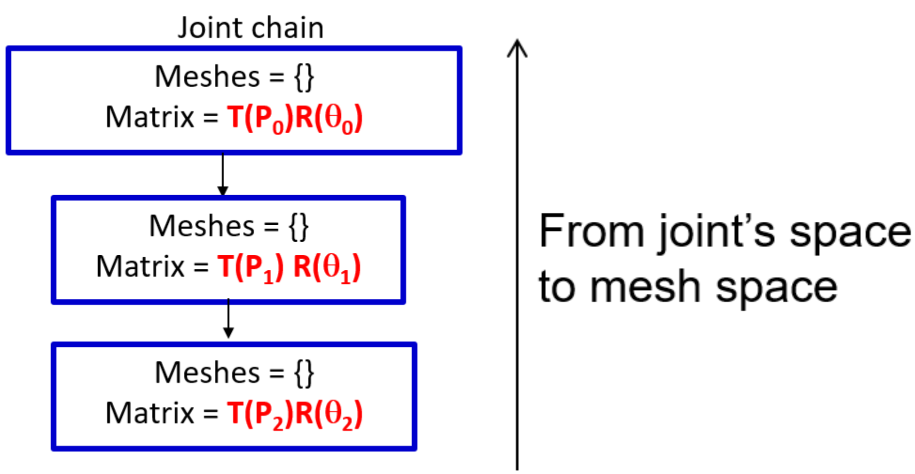
A mesh vertex
in joint space
$v' = Tv$



Local space of joint

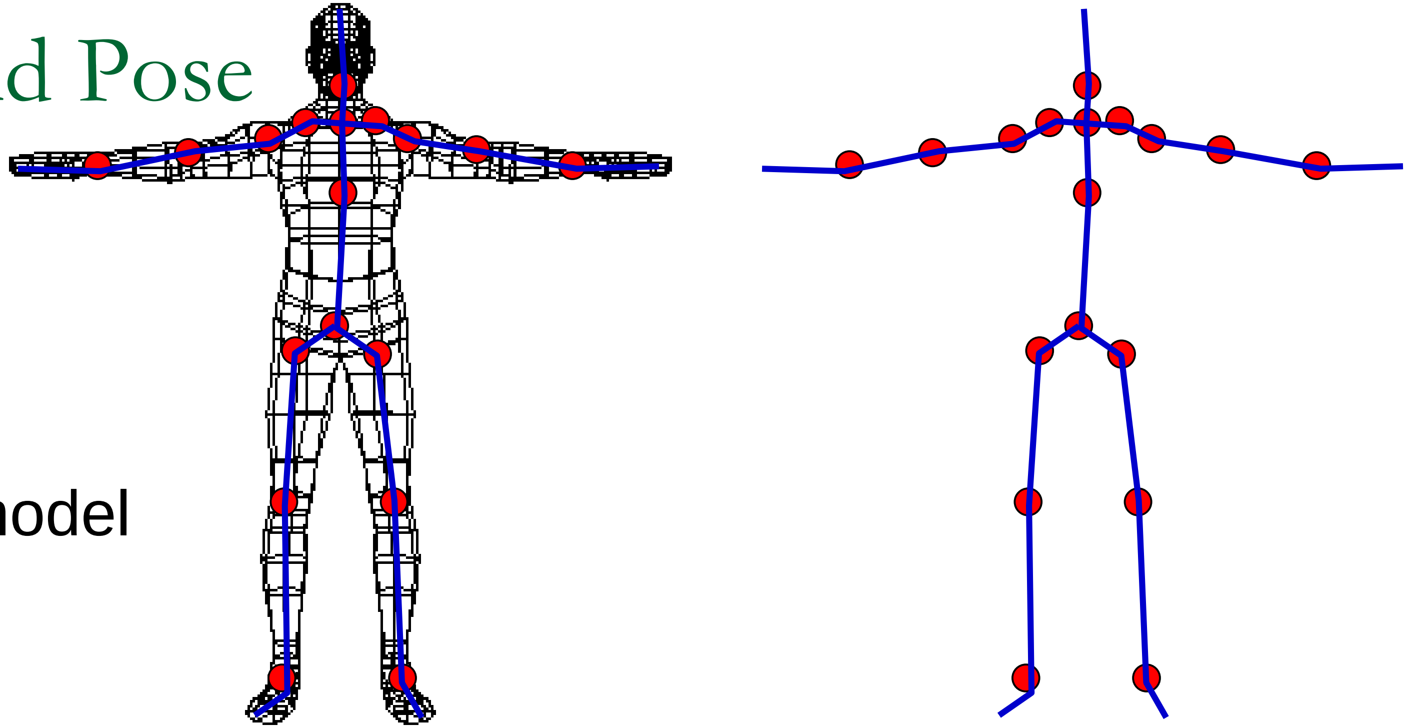
Vertex Transformation

- When a vertex is in the local space of a joint, it can be transformed using the joint angle obtained from the joint's channel.
- The vertex can then be transformed back to the mesh space by applying the transformations along the path from the joint to the root node of the joint hierarchy.



Bind Pose

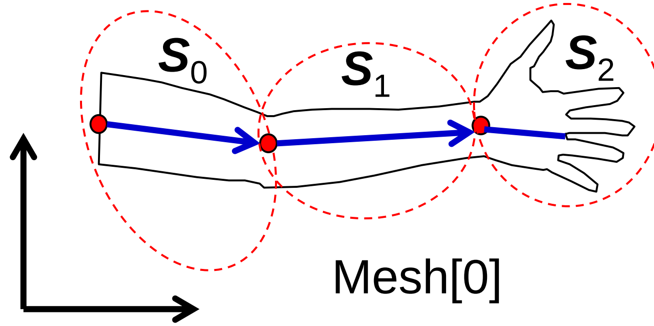
Rigged
character model



- The initial configuration of the mesh to which a skeleton is bound is called the bind pose.
- The joint positions are specified in the bind pose.
- The offset matrices of the bones are defined based on the joint positions.

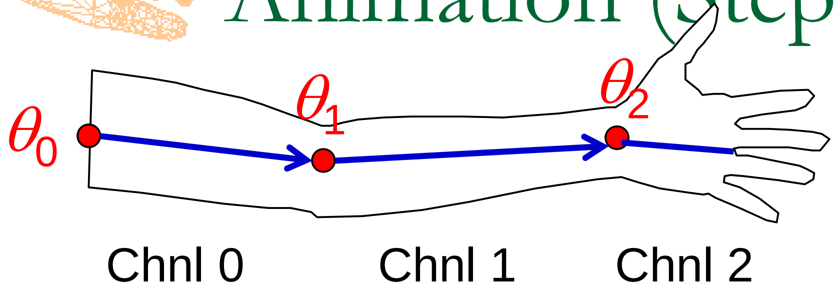
Mesh Representation

In the following example, Mesh[0] contains an array of three bones. Each bone defines a region of the mesh using an array of vertex indices. The offset matrix of a bone gives a transformation from mesh space to joint space.



Joint chain	Mesh[0]	Offset Matrix
Meshes = {} Name = “Shoulder” Matrix = **I**	Bone: 0 Name = “Shoulder” VertexWeights = S_0	$\mathbf{L_0}$
Meshes = {} Name= “Elbow” Matrix = **I**	Bone: 1 Name = “Elbow” VertexWeights = S_1	$\mathbf{L_1}$
Meshes = {} Name = “Wrist” Matrix = **I**	Bone: 2 Name = “Wrist” VertexWeights = S_2	$\mathbf{L_2}$

Animation (Step 1)



Matrix Product

		Posn key	*	Rotn key
Chnl 0	⟶	$\mathbf{T(P_0)}$		$\mathbf{R(\theta_0)}$
Chnl 1	⟶	$\mathbf{T(P_1)}$		$\mathbf{R(\theta_1)}$
Chnl 2	⟶	$\mathbf{T(P_2)}$		$\mathbf{R(\theta_2)}$

Get the transformation matrix for each channel and replace the corresponding joint's transformation matrix

Joint chain

Meshes = {}
Name = "Shoulder"
Matrix = $\mathbf{T(P_0)R(\theta_0)}$

↓

Meshes = {}
Name = "Elbow"
Matrix = $\mathbf{T(P_1)R(\theta_1)}$

↓

Meshes = {}
Name = "Wrist"
Matrix = $\mathbf{T(P_2)R(\theta_2)}$

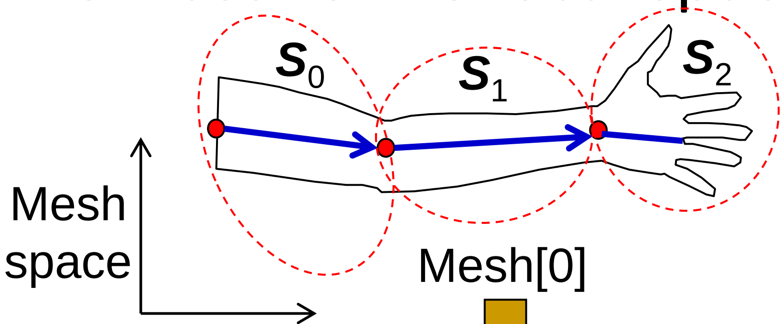
```
updateNodeMatrices()
```

Animation (Step 2)

Use the offset matrices of bones to transform mesh vertices to the local space of the joint.

To be implemented!

```
transformVertices()
```



Mesh[0]

Bone: 0
Name = “Shoulder”
VertexWeights = S_0

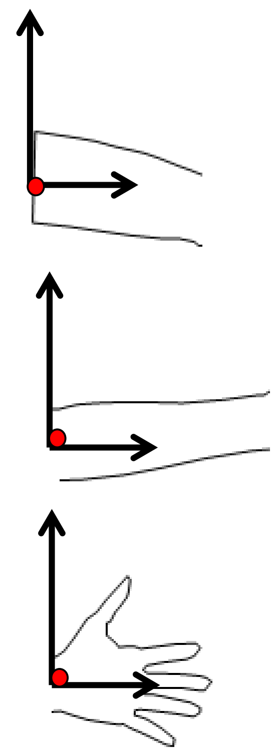
Bone: 1
Name = “Elbow”
VertexWeights = S_1

Bone: 2
Name = “Wrist”
VertexWeights = S_2

L_0S_0

L_1S_1

L_2S_2



Animation (Step 3)

With the node joints at the origin of the local coordinate space, we can apply transformations associated with the joints to each vertex set.

`transformVertices()`

$$S'_2 = T(P_0)R(\theta_0)\, T(P_1)\, R(\theta_1)\, T(P_2)R(\theta_2)\, L_2S_2$$

$$S'_1 = T(P_0)R(\theta_0)\, T(P_1)\, R(\theta_1)\, L_1S_1$$

$$S'_0 = T(P_0)R(\theta_0)L_0S_0$$

Bone: 0
Name = "Shoulder"
VertexWeights = S_0

L_0S_0

Bone: 1
Name = "Elbow"
VertexWeights = S_1

L_1S_1

Bone: 2
Name = "Wrist"
VertexWeights = S_2

L_2S_2

Meshes = {}
Name = "Shoulder"
Matrix = $T(P_0)R(\theta_0)$

Meshes = {}
Name = "Elbow"
Matrix = $T(P_1)R(\theta_1)$

Meshes = {}
Name = "Wrist"
Matrix = $T(P_2)R(\theta_2)$

Programming Considerations

- The transformations listed on the previous slide are applied to sets of vertices S_i , not primitives. The vertices can be in any order.
- Animated character models can in general have multiple meshes, each mesh with its own array of bones and vertices.
- Bone-Node correspondence:

```
node = scene->mRootNode->FindNode(bone->mName);
```

- OpenGL 1/2: We need to maintain a list of vertices (for each mesh).

Vertex Transformation Matrix

`transformVertices()`

For each mesh of the scene:

For each bone i of the mesh:

Get offset matrix of the bone: L_i

Find the node corresponding to the bone

Get the node's transformation matrix Q_a

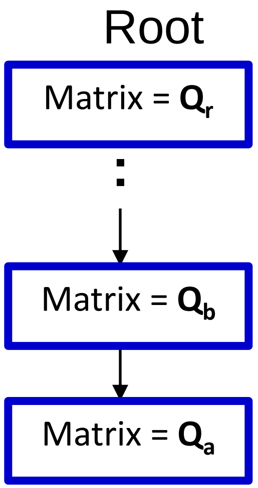
Get its parent's transformation matrix Q_b

... Continue up to the root node Q_r

Form the matrix product $M_i = Q_r \ldots Q_b Q_a L_i$

Form the normal matrix $N_i = M_i^{-T}$

Using the above matrices, **transform vertices** and normal vectors attached to the bone.



Storing Initial Mesh Data

```
struct meshInit
{
    int mNumVertices;
    aiVector3D* mVertices;
    aiVector3D* mNormals;
};

meshInit* initData;
```

numMesh = number of meshes
numVert = number of vertices of mesh[i]

loadModel():

```
initData = new meshInit[numMesh];
```

For each mesh i:

```
(initData + i)->mNumVertices = numVert;
(initData + i)->mVertices = new aiVector3D[numVert];
(initData + i)->mNormals = new aiVector3D[numVert];
```

Populate the above two arrays with mesh data.

Transforming Mesh Vertices

`transformVertices()`

…continued from slide 20:

Get the vertex ids attached to the current bone:

```
vid = (bone->mWeights[k]).mVertexId;
```

Get **initial** vertex and normal data:

```
vert = (initData + imesh)->mVertices[vid];
norm = (initData + imesh)->mNormals[vid];
```

Transform the above using matrices M_i, N_i and store them in the mesh object:

```
mesh->mVertices[vid] =
mesh->mNormals[vid] =
```

Transforming Mesh Vertices

- In Assimp, vertices and normals are objects of type aiVector3D
- The matrices have type aiMatrix4x4

$$m = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

m.a4

aiMatrix3x3(m)

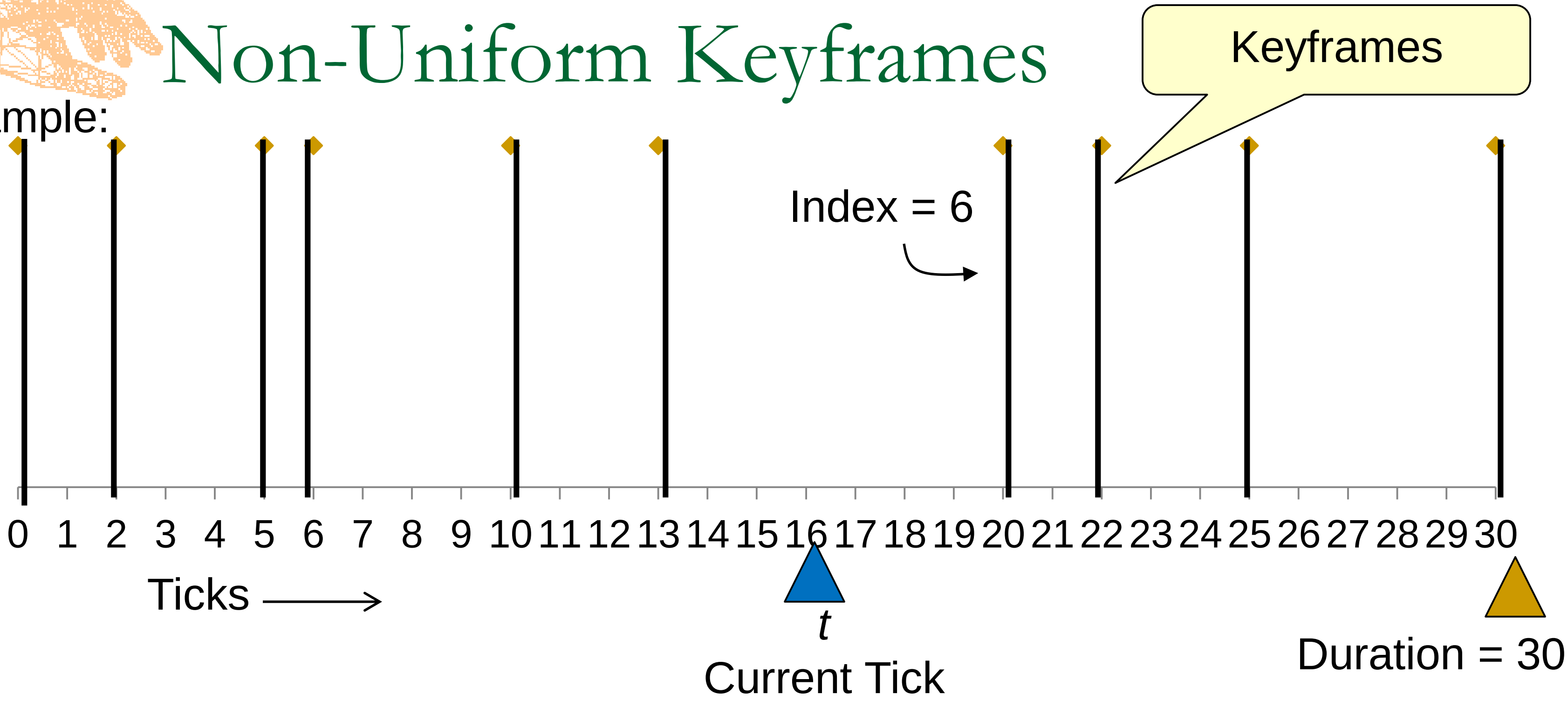
Non-Uniform Keyframes

- BVH files contained a set of uniformly distributed keyframes (one keyframe per tick).
- For animated character models, keyframes may not be uniformly distributed. Example (dwarf.x):

```
---------------- Animation Data -------------------
Number of animations = 1
 Anim 0:  nchanls = 45  ticksPerSec = 0  duration (ticks) = 55
Channel 7: nodeName = rknee  nposkeys = 25   nrotKeys = 25
          :
      rotnKey 4:  Time (ticks) = 4   Value = -0.859406 -0.511293 -0 0
      rotnKey 5:  Time (ticks) = 5   Value = -0.859406 -0.511293 -0 0
      rotnKey 6:  Time (ticks) = 12  Value = -0.859406 -0.511293 -0 0
      rotnKey 7:  Time (ticks) = 14  Value = -0.859406 -0.511293 -0 0
      rotnKey 8:  Time (ticks) = 15  Value = -0.859406 -0.511293 -0 0
          :
     rotnKey 24:  Time (ticks) = 55   Value = -1 -0 -0 0
```

Non-Uniform Keyframes

Example:



Find “index” such that

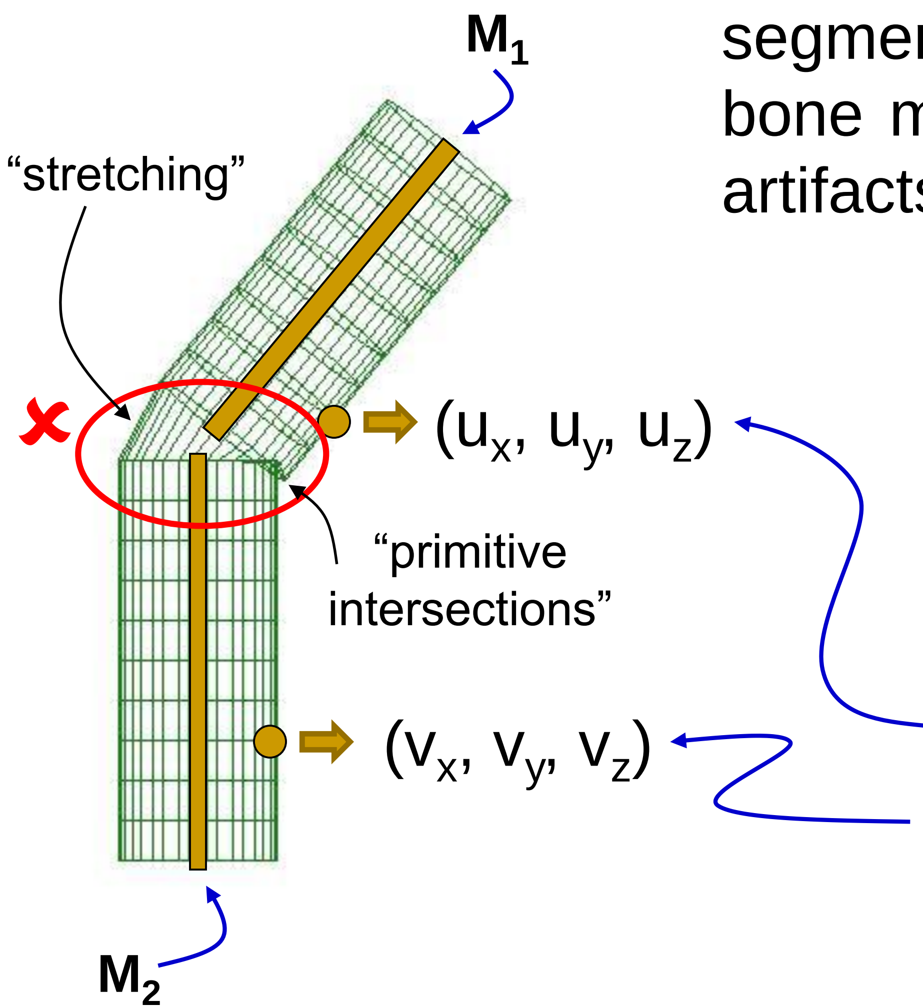
```
channel->mRotationKeys[index-1].mTime < t ≤ mRotationKeys[index].mTime
```

Interpolate between corresponding values:

```
rotn1 = (channel->mRotationKeys[index-1]).mValue;
rotn2 = (channel->mRotationKeys[index]).mValue;
time1 = (channel->mRotationKeys[index-1]).mTime;
time2 = (channel->mRotationKeys[index]).mTime;
factor = (t-time1)/(time2-time1);
rotn.Interpolate(rotn, rotn1, rotn2, factor);
```

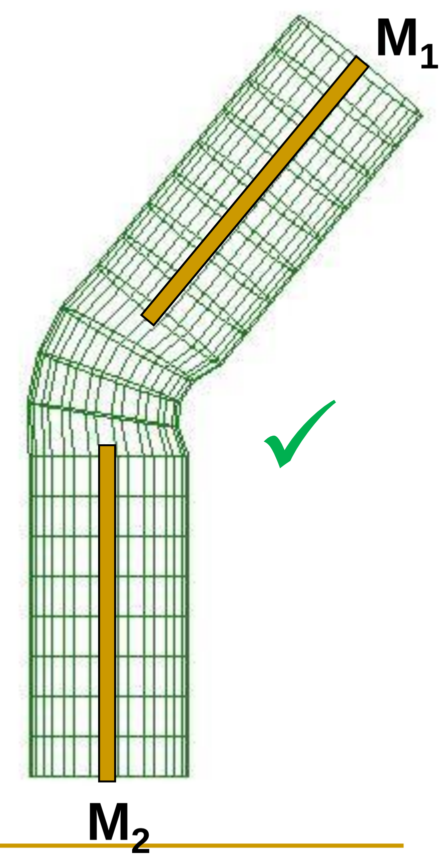
Vertex Transformations

Transformation of two adjoining segments of a mesh using separate bone matrices can create undesirable artifacts at the vertices near the joint.



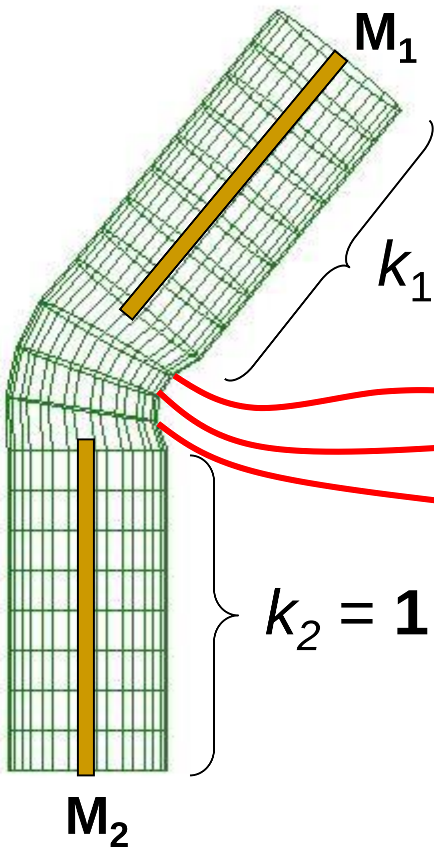
Bone

1	M_1
2	M_2
…	…
…	…



Vertex Blending

For smooth joint deformation, the vertices (skin) at a joint must be associated with both the neighboring bone matrices, using a set of weights.



$k_1 = 1,\ k_2 = 0$

Transformation: $k_1\,\mathbf{M}_1 + k_2\,\mathbf{M}_2$

$k_1 = 0.7;\ \ k_2 = 0.3$
$k_1 = 0.5;\ \ k_2 = 0.5$
$k_1 = 0.3;\ \ k_2 = 0.7$

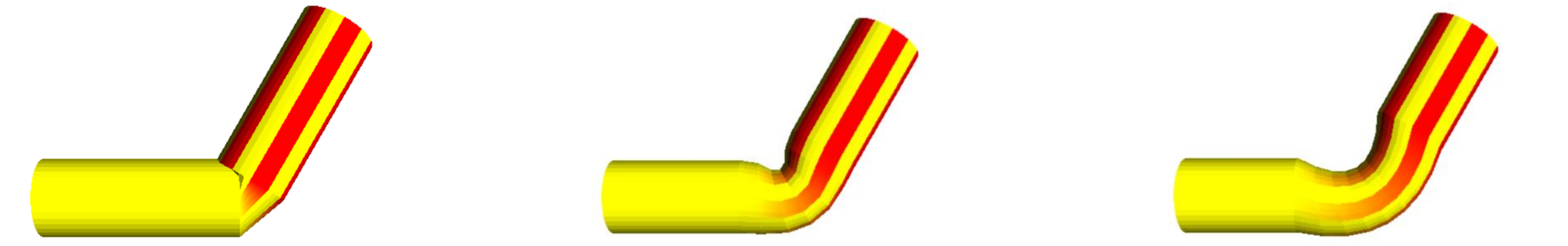
Both bones influence these vertices

$k_2 = \mathbf{1},\ k_1 = 0$

```
(bone->mWeights[k]).mVertexId
```

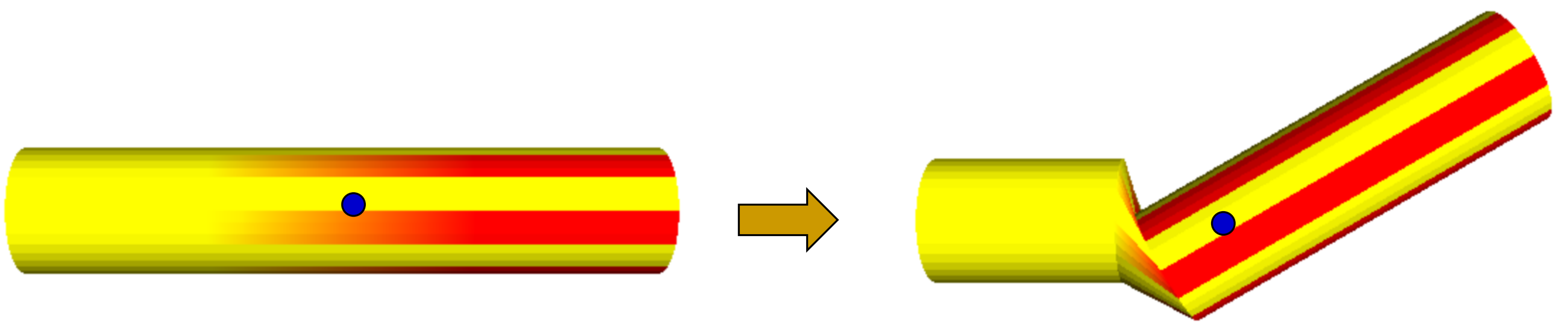
```
(bone->mWeights[k]).mWeight
```

Vertex Blending



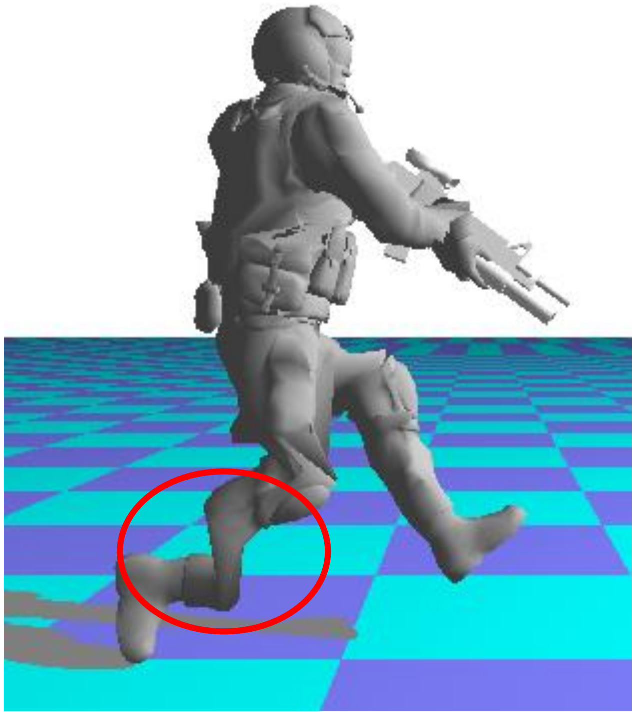
Vertex Blending: Problem

- When weights are assigned to vertices of a bone, its region of influence extends beyond the joint to another bone.
- Transforming those vertices with only a single bone matrix with unit weight can lead to improper transformations.

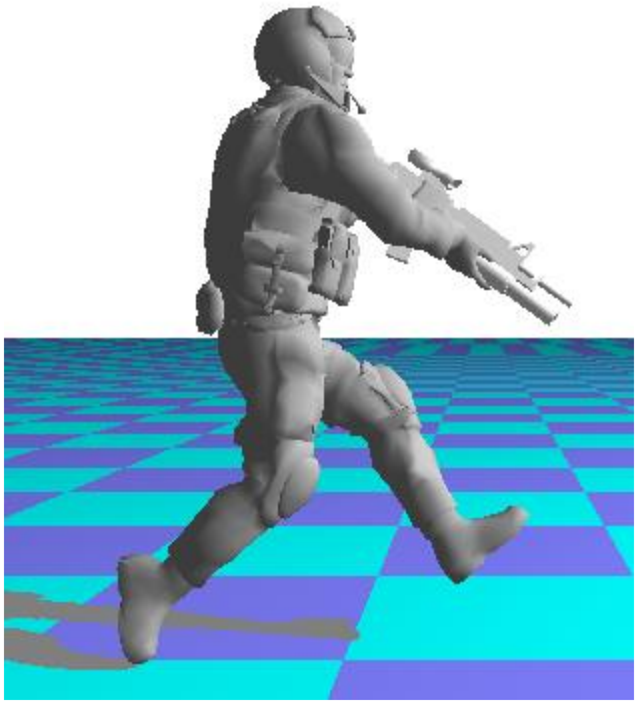


Vertex Blending

ArmyPilot.x



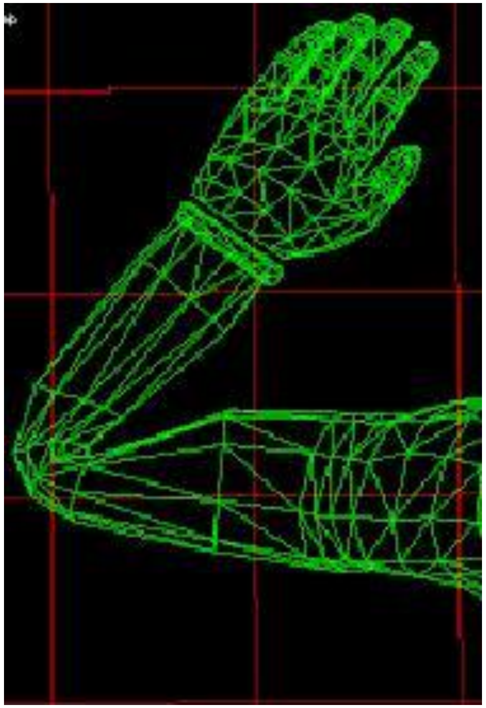
Without using bone weights



With bone weights

Vertex Transformations - Limitations

Transforming vertices using a combination of bone matrices can make joints shrink for large angles (an artifact known as the ***collapsing elbow***).



Vertex Transformations - Limitations

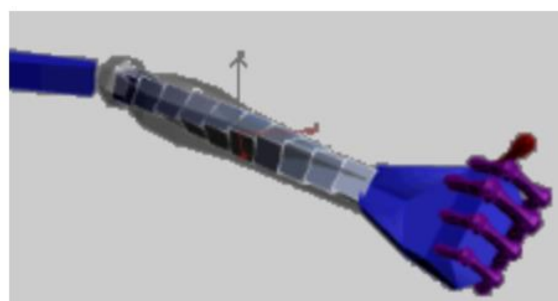
Rotations about bone axis can cause joints to undergo a twisting motion (***candy-wrapper effect***)

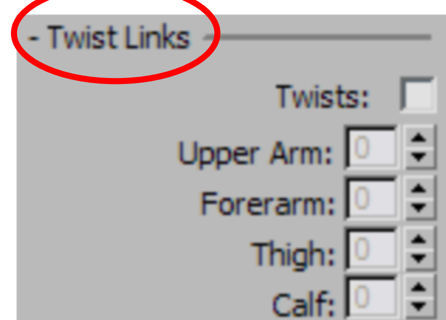
$$\mathbf{M_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

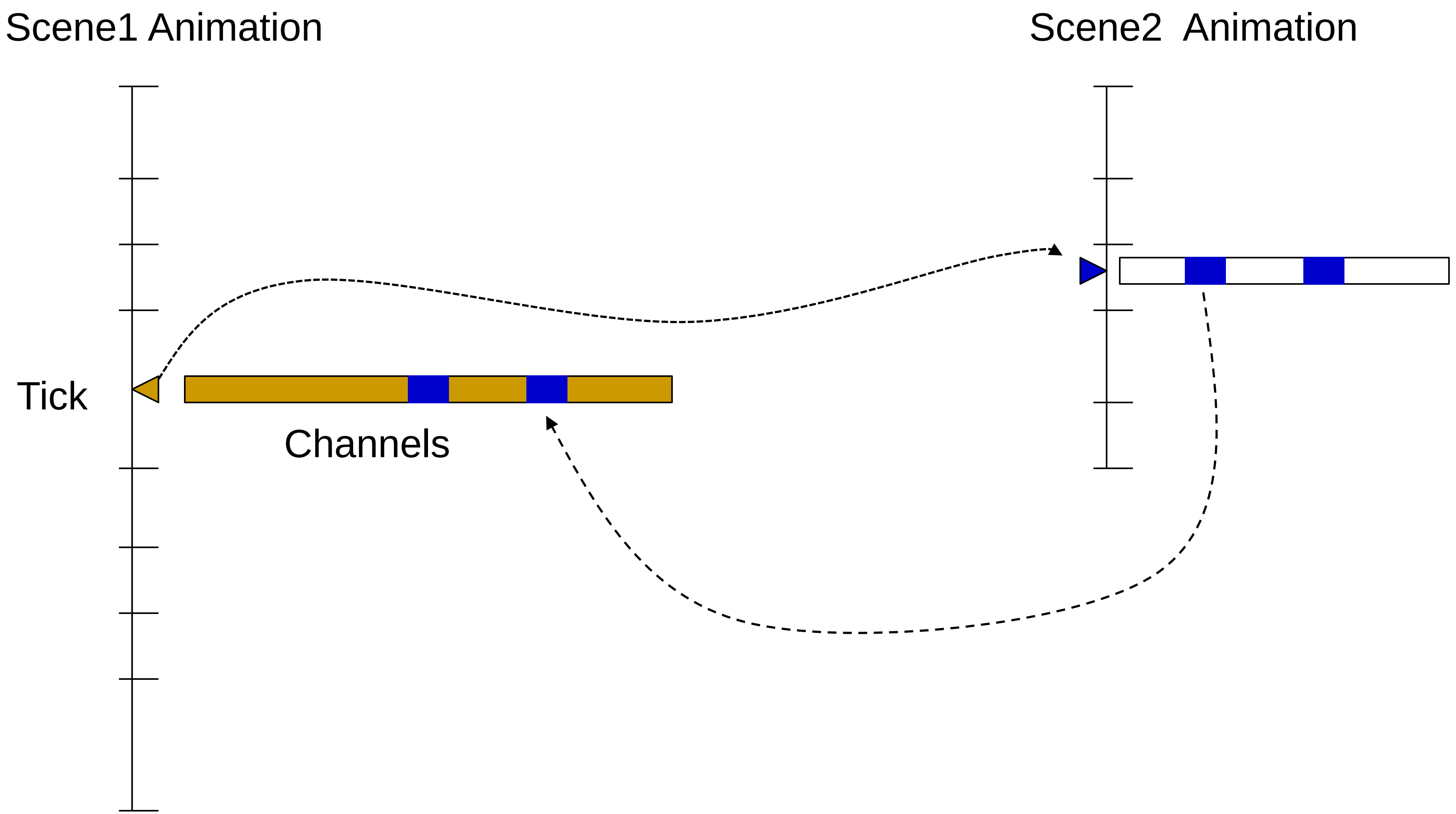
$$0.5*\mathbf{M_1} + 0.5*\mathbf{M_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Solution: Increase the number of joints (and bones) (***twist links***)





Animation Re-targetting



Animation re-targetting is the process of using some of the channel values from one animation in another animation sequence