Advanced Animation Programming

GPR-450
Daniel S. Buckstein

Hierarchies & Skeletal Animation: Applications
Weeks 5 – 7

License

Skeletal Animation Applications

- Applications of skeletal animation
  - Combined with other forms of animation
    - Sprite-based animation
  - Traditional mesh skinning
    - Factoring out the base pose
  - Intro to inverse kinematics
  - Advanced mesh skinning

Skeletal + Sprite-Based Animation

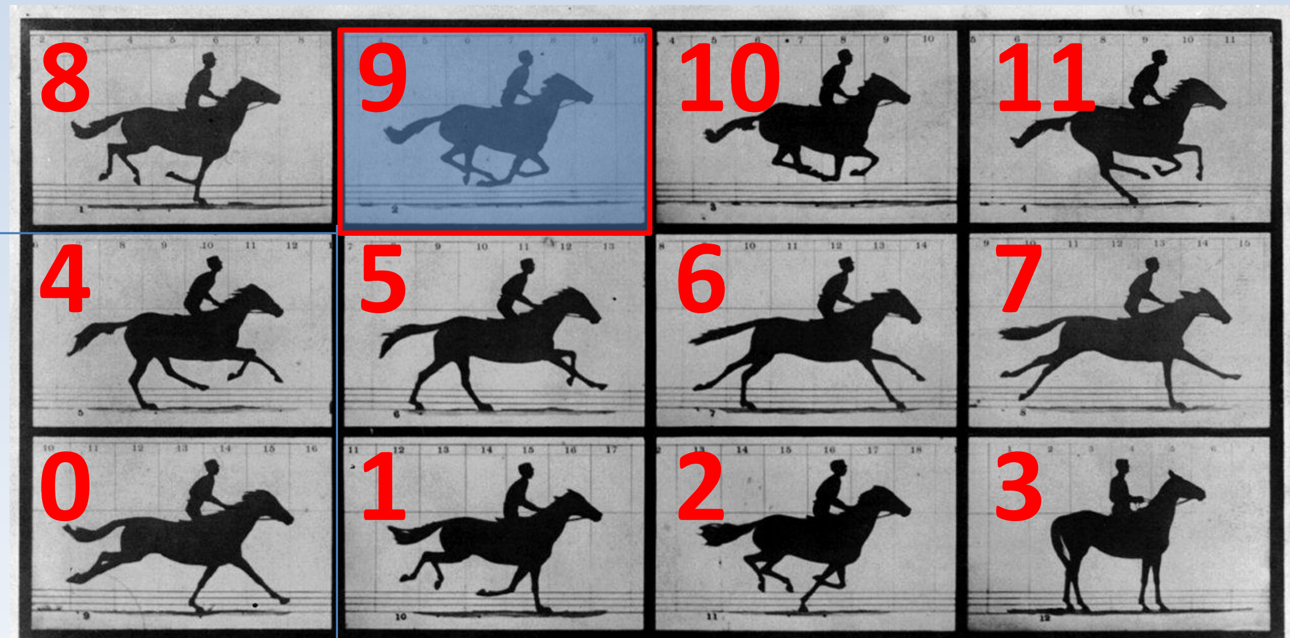
- Using the hierarchy as a tool to facilitate other forms of animation:
- Dragon Quest:
  sprites

Skeletal + Sprite-Based Animation

- Primary form of animation is ***sprites***…
- …secondary animation is ***skeletal***

- **Sprite animation (briefly):**
- Keyframe controller tells you the cell index
- Index represents element in an array of UV scales and offsets: …

Sprite-Based Animation



Cell height = 1/3

Frame 9
offset Y = 2/3

Texture UV height= 1.0

Texture UV width = 1.0

Cell width = 1/4 = 0.25

Frame 9 offset X = 1/4 = 0.25

Algorithms for an ***evenly-divided sprite sheet:***
Cell width = 1 / num columns
Cell height = 1 / num rows
Offset x = column index * cell width
Offset y = row index * cell height

Sprite-Based Animation

```
// animation code (CPU)
unsigned int index = keyframeCtrl->current;
vec2 uvOffset = spriteSheet->offsets[index];
vec2 uvScale = spriteSheet->scales[index];
```

- Passed as shader uniforms used to transform default UVs of a simple plane (range 0 – 1):

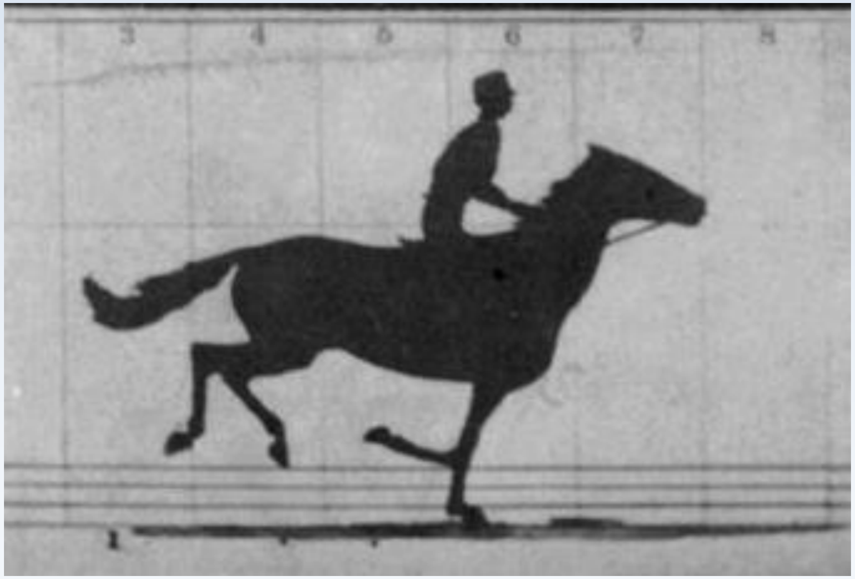
```
// vertex shader (GPU)
uvFinal = uvAttr * uvScaleUnif + uvOffsetUnif;
```

Sprite-Based Animation

- Transforms your sprite plane's texturing, kind of like a rectangular magnifying glass:

Sprite-Based Animation

- Changing the keyframe index over time results in a film-like effect on the plane:



Skeletal + Sprite-Based Animation

- What do we need to achieve Dragon Quest’s animation style?

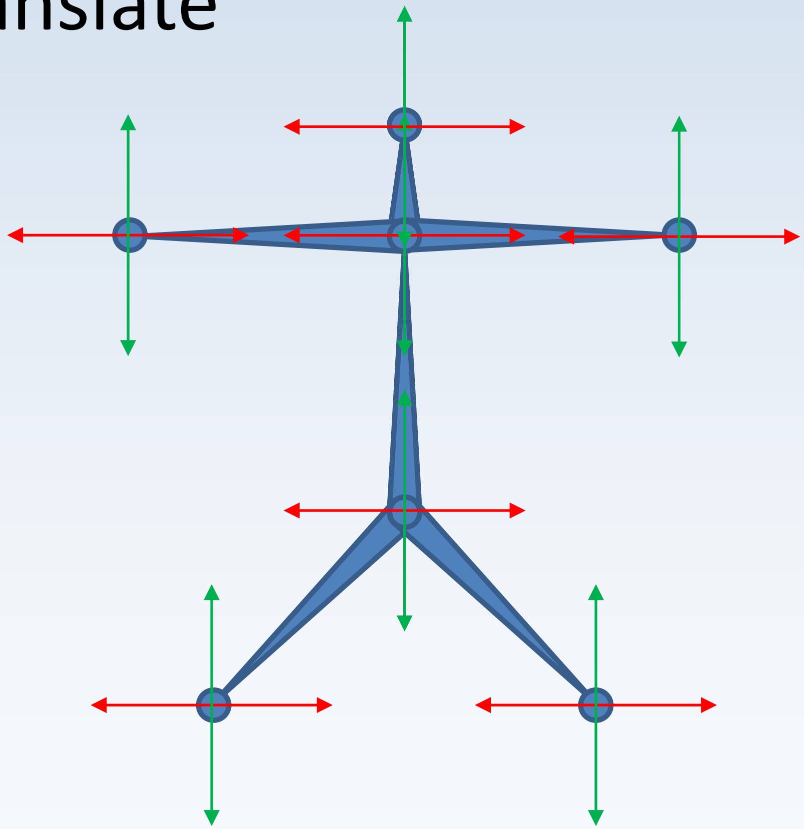
1) Hierarchy of rigid and non-rigid joints
2) Animated *pieces* of the character are drawn on planes as sprite sheet sequences
3) Planes’ *transforms* are relative to their respective joint

- How many keyframe controllers???

Skeletal + Sprite-Based Animation

1) Hierarchy of rigid and non-rigid joints

- Rotate, translate and scale enabled



Skeletal + Sprite-Based Animation

2) Animated *pieces* of the character are drawn on planes as sprite sheet sequences

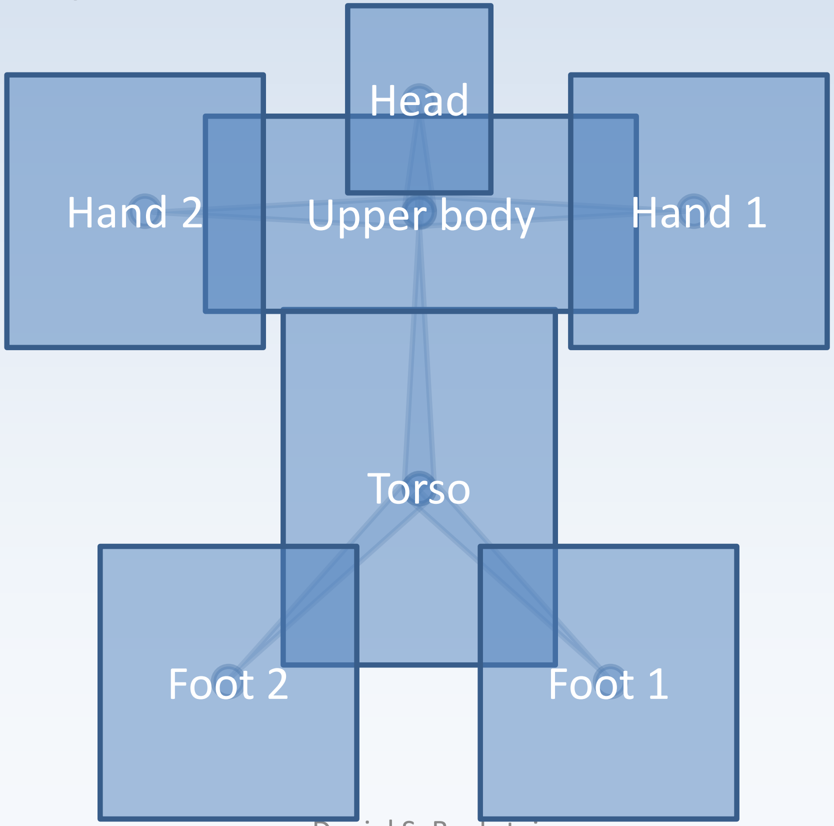
Head	Upper body	Torso	Foot 1	Foot 2	Hand 1	Hand 2

Daniel S. Buckstein

https://www.spriters-resource.com/ds_dsi/dragonquest6realmsofrevelation/

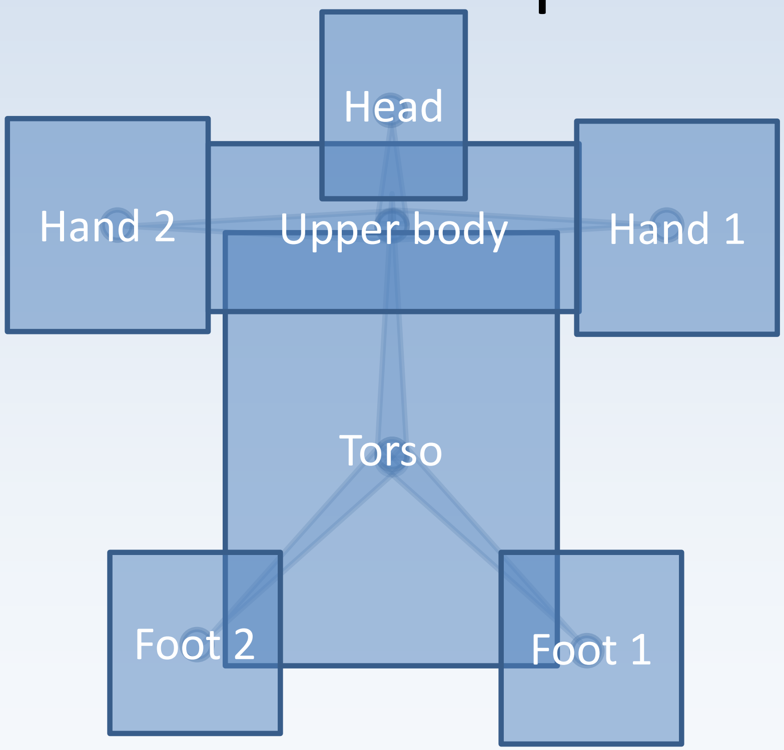
Skeletal + Sprite-Based Animation

3) Planes' *transforms* are relative to their respective joint



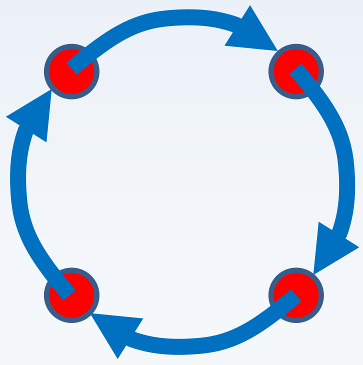
Skeletal + Sprite-Based Animation

- While sprites animate, skeletal controller changes planes’ sizes and positions over time:



Skeletal + Sprite-Based Animation

- Why do bones rotate, translate *and* scale?
- Why minimal number of skeleton poses?
- E.g. 4 poses on continuous loop???
- Any common interpolation algorithm that has 4 influences on a path???



Psssst… pose-to-pose does not necessarily use LERP

Skeletal Animation: The Base Pose

- In addition to *key poses*, skeletons usually have a "*base pose*" or "*initial pose*"
- All pose data can be simplified to represent a *change* from the base pose to current pose:

$$\mathrm{pose}_{n,t} = \mathrm{combine}\left(\mathrm{pose}_{n,\mathrm{base}}\ , \mathrm{poseData}_{n,t}\right)$$

where '*combine*' could be concatenation or simply adding the respective pose components

Skeletal Animation: The Base Pose

- You could say that the base pose is a special case, where the pose itself is constant data:

$$\text{pose}_{n,\text{base}} = \text{poseData}_{n,\text{base}}$$

- Fun fact: if the *raw data* doesn't represent a change (i.e. *identity*), you get the base pose:

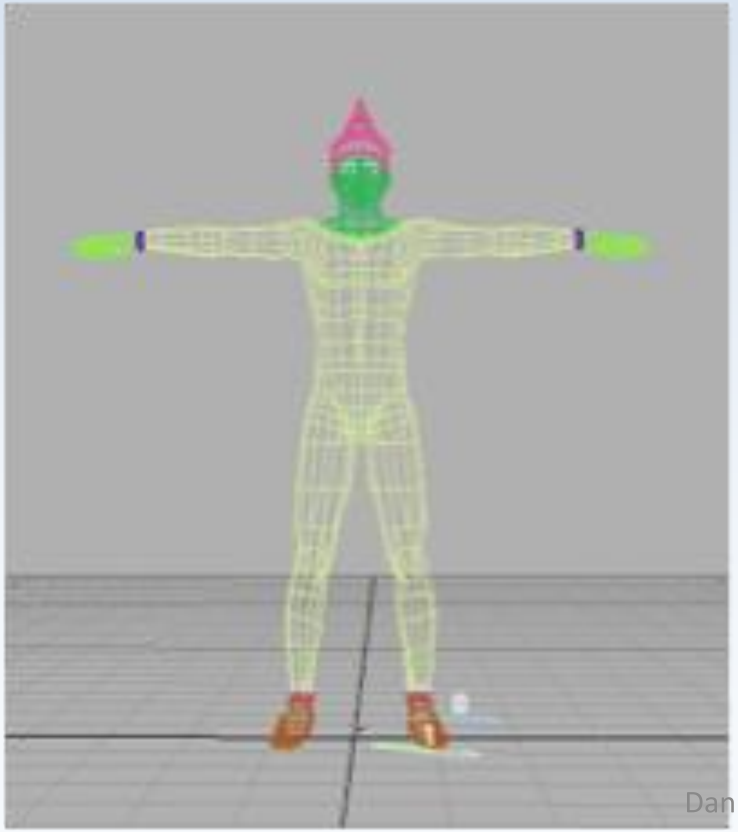
$$\text{pose}_{n,t} = \text{combine}\left(\text{pose}_{n,\text{base}}, \text{identityPose}\right)$$

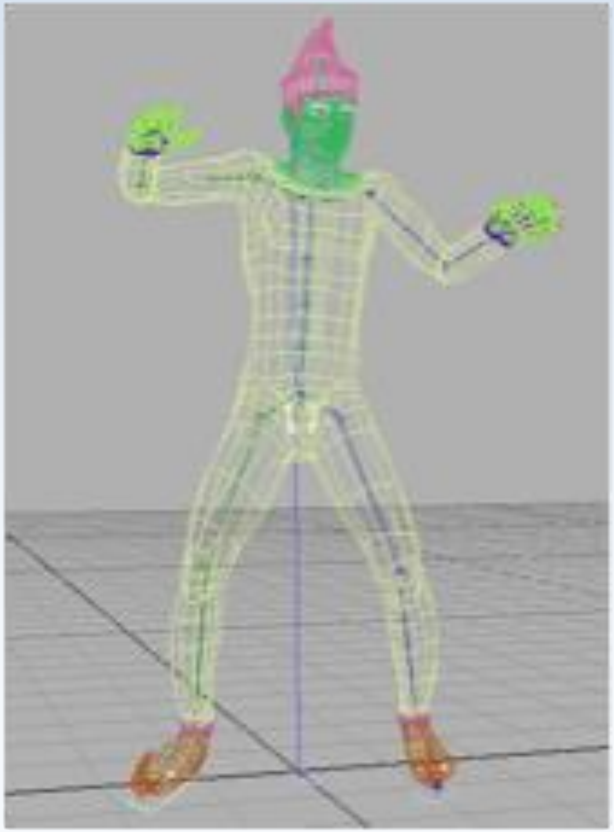
Skeletal Animation: The Base Pose

- Base pose to current pose:

Base pose → Current pose

Apply pose ***change***





Skeletal Animation: The Base Pose

- The base pose data is used ***on load*** to calculate the ‘***bind pose matrices***’:

1) Solve FK for base pose
2) Invert results (you’ll see why shortly)

1) For each joint, solve FK

$$^{\text{parent}}T_{n_{\text{base}}} = \text{convert}\left(\text{pose}_{n,\text{base}}\right)$$

Local transformation matrix converted from base pose data

Node *n*’s base pose data

Skeletal Animation: The Base Pose

- The base pose data is used ***on load*** to calculate the '***bind pose matrices***':

1) Solve FK for base pose
2) Invert results (you'll see why shortly)

1) For each joint, solve FK

$$ {}^{\mathrm{world}}T_{n_{\mathrm{base}}} = {}^{\mathrm{world}}T_{\mathrm{parent}_{\mathrm{base}}}\,{}^{\mathrm{parent}}T_{n_{\mathrm{base}}} $$

Joint *n*'s solved base pose FK

Parent's solved base pose FK

Local transformation matrix (from previous slide)

Skeletal Animation: The Base Pose

- The base pose data is used ***on load*** to calculate the '***bind pose matrices***':

1) Solve FK for base pose
2) Invert results (you'll see why shortly)

2) For each joint, invert FK result

$$B_n = {}^{\text{world}}T_n^{-1}{}_{\text{base}}$$

Bind pose matrix for joint *n*

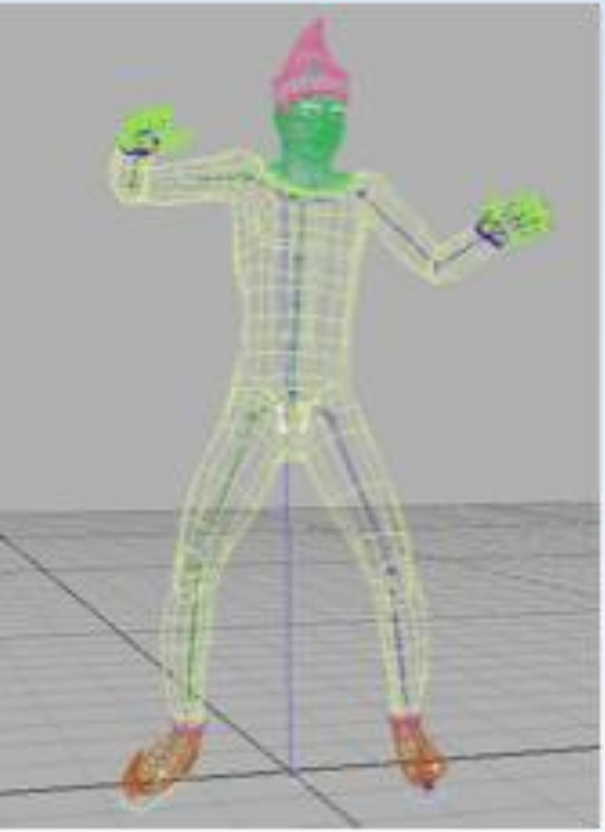
Joint *n*'s solved base pose FK (from previous slide), inverted

Skeletal Animation: Mesh Skinning

- Ultimately we have a skeleton and a mesh
- Mesh can be thought of as the "***skin***" or what we will actually see when we render
- "***Skinning***" means we bind mesh to skeleton
- …much more versatile than morph targets!
- Animate the *skeleton instead* of the vertices
- When the skeleton animates, the mesh deforms!

Skeletal Animation: Mesh Skinning

- "Current pose" is the result of calculating the global transform for all joints *every update*:



$$ {}^{\text{parent}}T_{n_t} = \text{convert}\left(\text{pose}_{n,t}\right) $$

$$ {}^{\text{world}}T_{n_t} = {}^{\text{world}}T_{\text{parent}_t}\,{}^{\text{parent}}T_{n_t} $$

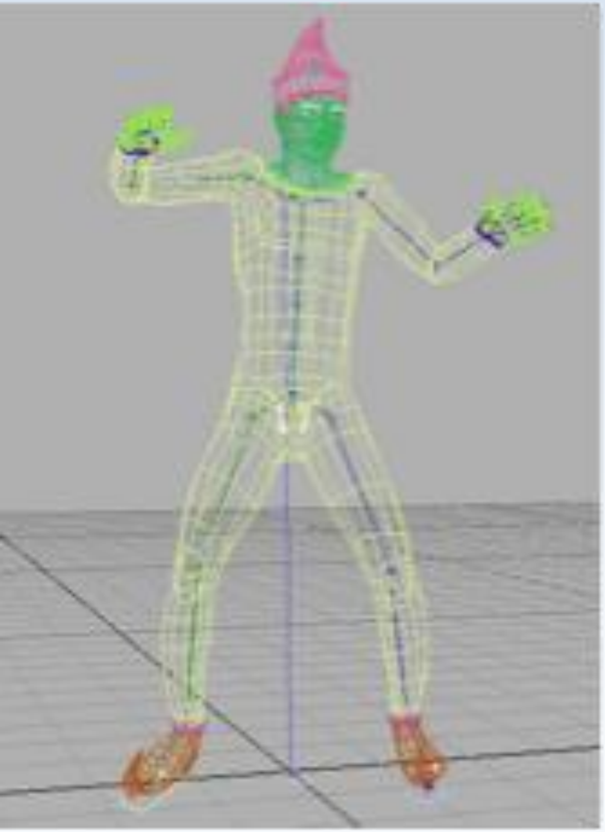
Solved FK for joint *n* at time *t*

Parent's solved FK at time *t*

Local transform for joint *n* at time *t* (converted from current pose data)

Skeletal Animation: Mesh Skinning

- "Current pose": Let's create a simpler variable naming convention for this…



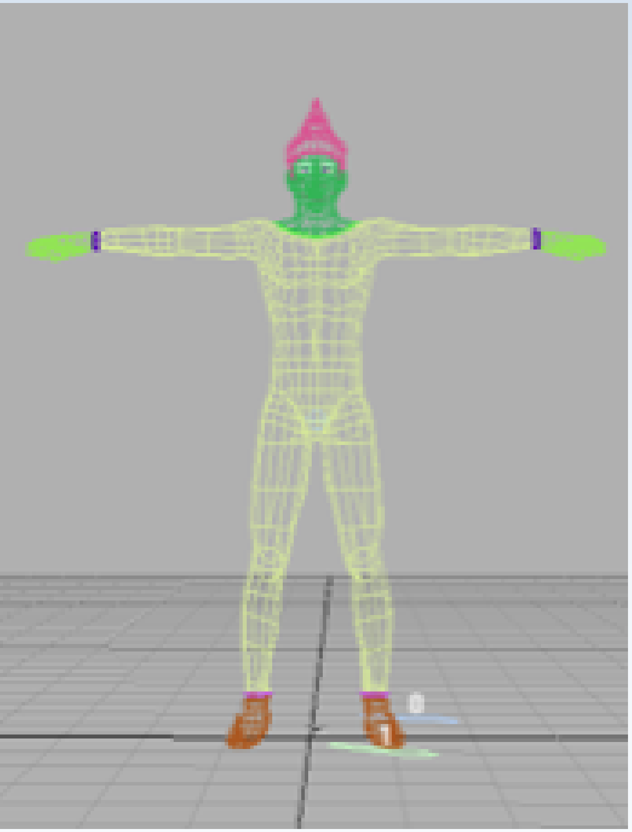
$$\boldsymbol{K_n = {}^{world}T_{n_t}}$$

This is what we need for skinning!

How about *K* for '*kinematic*' matrix?

Skeletal Animation: Mesh Skinning

- "Bind pose" is also required, we calculated this a bit earlier:



$$^{\text{world}}T_{n_{\text{base}}} = {}^{\text{world}}T_{\text{parent}_{\text{base}}} \; {}^{\text{parent}}T_{n_{\text{base}}}$$

$$\boldsymbol{B_n = {}^{world}T_n^{-1}{}_{base}}$$

Also need this for skinning!

Skeletal Animation: Mesh Skinning

- ***Why do we care about the base pose*** when we skin a mesh???
- The "magic" of mesh skinning is literally this:
- Start with T-pose (base pose), post-FK
- We want to end up in the current world pose
- We have just described a *transformation* from ***base pose* to *current pose***

Skeletal Animation: Mesh Skinning

- What is the formula that describes ***base pose to current pose***???
- Use this rule: ${}^{c}T_{a} = {}^{c}T_{b}\,{}^{b}T_{a} = {}^{c}T_{b}\,{}^{a}T_{b}^{-1}$

$$
\begin{aligned}
S_n &= {}^{\text{world}}T_{n_t}\,{}^{n}T_{\text{world}_{\text{base}}} \\
&= {}^{\text{world}}T_{n_t}\,{}^{\text{world}}T_{n}^{-1}{}_{\text{base}}
\end{aligned}
$$

Skeletal Animation: Mesh Skinning

- The skinning matrix for joint *n* at time *t*:

$$\boldsymbol{S_n = K_n B_n}$$

where

- B_n is the bind-pose matrix, calculated ***once, on load***
- K_n is the result of kinematics, calculated ***every update***
- S_n is the skinning matrix, calculated ***every update***

Skeletal Animation: Mesh Skinning

- The skinning matrix for joint *n* at time *t*:

$$\boldsymbol{S_n} = \boldsymbol{K_n} \boldsymbol{B_n}$$

- Recall that *B* is the ***inverse of the base FK calculation***… which implies that the base FK is ***undone first*** (because it's on the right)…
- …then transform ***to*** the current global frame!

Skeletal Animation: Mesh Skinning

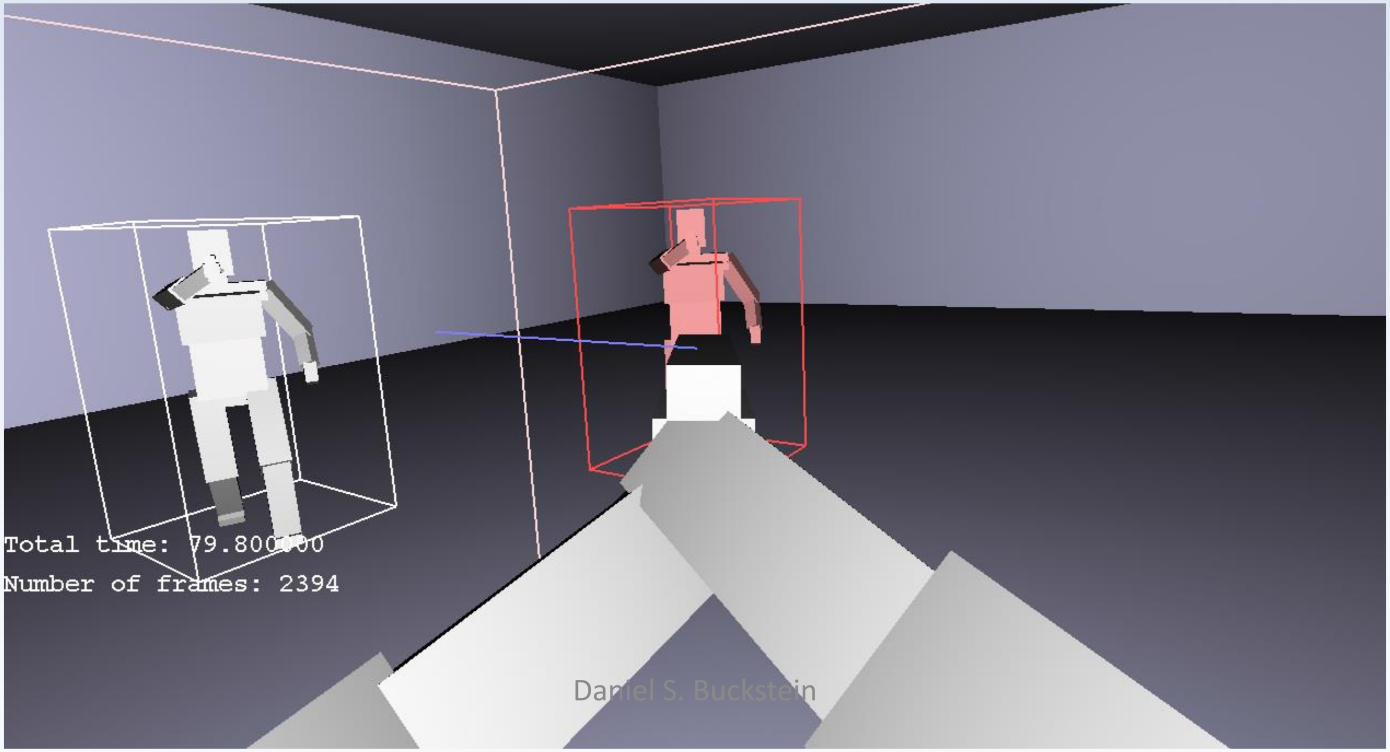
- ***Mesh skinning***: we have the transforms that we want the mesh to follow…
- Two types of skinning:

1) Rigid bind: each vertex influenced by one joint’s transformation
2) Weighted bind: each vertex influenced by multiple joints’ transformations

Skeletal Animation: Mesh Skinning

1) Rigid bind: Every vertex has ***one*** joint influencing its change

(great for robots and debug characters)



Skeletal Animation: Mesh Skinning

1) Rigid bind: Every vertex has ***one*** joint influencing its change

$$v' = S_n v$$

where

- v is a vertex
- n is the node influencing the vertex
- S_n is the skinning matrix for node *n*
- v' is the skinned vertex

Skeletal Animation: Mesh Skinning

2) Weighted bind: *multiple joints influencing each vertex*

- *Weighted sum*: each vertex has a list of influences, each influence has a *strength* (how much it changes the skinning)
- *Total strength of all influences must equal* ***1***
- Usually, every vertex in the mesh has a *maximum number of influences* (how about 4… because vec4)

Skeletal Animation: Mesh Skinning

- ***"The Skinning Equation"***:
- *For each spatial vector* (vertices, normals, etc.) in the mesh, solve ***the skinning equation***:

$$v' = \sum_{0 \le i < c} \left(S_{n_i} \, w_i \, v \right)$$

where

c is the *influence count* (e.g. 4)

i is the *influence index* (e.g. less than 4)

n_i is the *joint influence index*

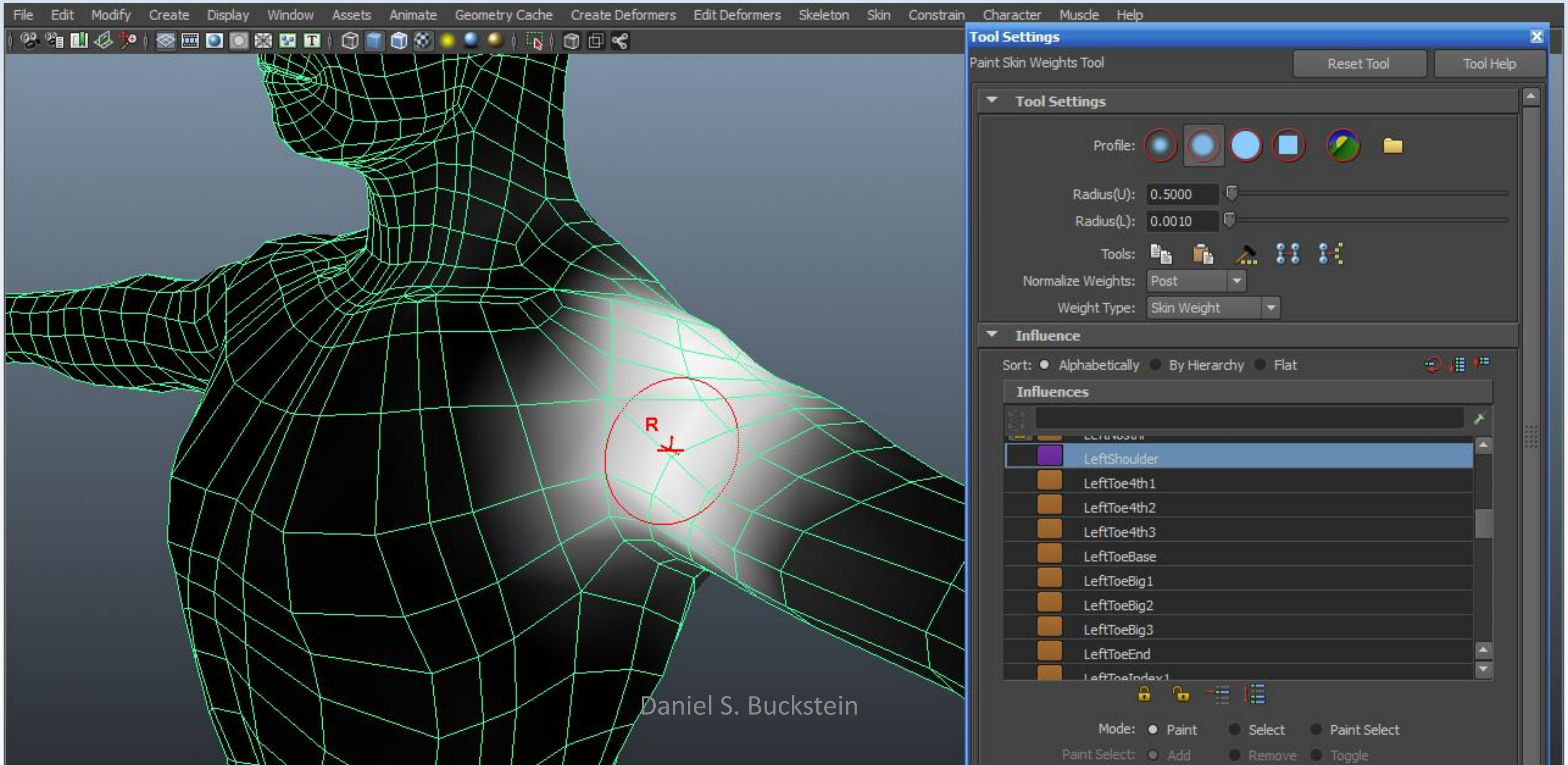
S_{n_i} is the *skinning matrix for influence* i

w_i is the *weight/strength of influence* i

v is the ***input vector***, and v' is the ***skinned vector***

Skeletal Animation: Mesh Skinning

- Where do influence weights come from?
- Maya's "paint skin weights tool":



Skeletal Animation: Mesh Skinning

- Algorithmically:
- On load:
  - Calculate *base* world transform for each joint
  - Calculate inverse matrix to be used in skinning
- Every frame
  - Calculate *current* world transform for each joint
  - Update “base-to-current” matrix
  - For each vertex: compute weighted sums… done!

Skeletal Animation: Mesh Skinning

- Skin weights: no "standard" formats ☹
- Maya exports XML file with all vertices and their skin weights
- .md5 extension contains all of the skeletal, skin weights and mesh data in one package
- …also FBX

- …do some research, pick the easiest combo!

Skeletal Animation Applications

- Extended data structure: ***hierarchy pose set***

```
struct HierarchyPoseSet
{
    const Hierarchy *hierarchy;
    HierarchyNodePose **keyPoseList; // deltas
    HierarchyNodePose *basePose; // base pose
    mat4 *bindPoseMatrix; // inverse base pose FK
    unsigned int keyPoseCount;
};
```

Skeletal Animation Applications

- Extended data structure: ***hierarchy state***

```
struct HierarchyState
{
    const Hierarchy *hierarchy;
    HierarchyNodePose *localPoseList; // base+delta
    mat4 *localTransformList; // converted pose
    mat4 *worldTransformList; // kinematics result
    mat4 *skinningMatrixList; // kinematics * bind
};
```

Skeletal Animation Applications

- Extended algorithm: ***updating skeleton state***

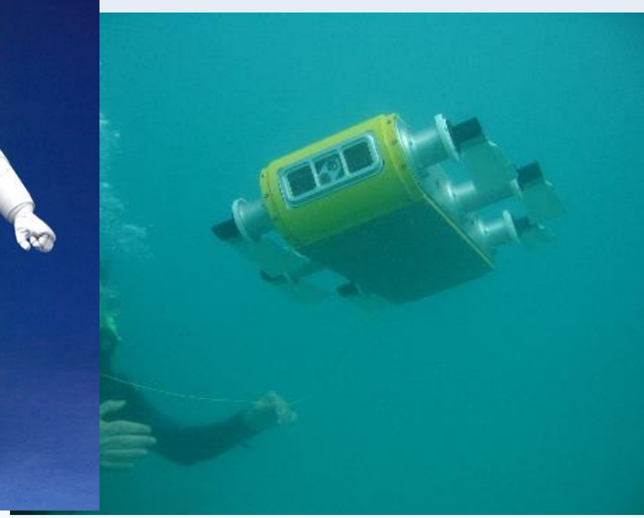
```
////////////////////////////////////////////////
// update local matrices
for (n = 0; n < state->hierarchy->nodeCount; ++n)
  state->localTransformList[n] =
    convertPoseToMatrix(state->localPoseList[n]);

////////////////////////////////////////////////
// proceed to do kinematics...
////////////////////////////////////////////////
// calculate skinning matrices
for (n = 0; n < state->hierarchy->nodeCount; ++n)
  state-> skinningMatrixList[n] =
    concatMatrix( state->worldPoseList[n] ,
                  poseSet->bindPoseMatrix[n] );

////////////////////////////////////////////////
// send skinning matrices to skinning program...
```

Inverse Kinematics

- Real-world problems:
- Robots: generally goal-oriented
- Critical motions need to be *perfect*



Inverse Kinematics

- FK vs. IK:

Forward kinematics (FK): we know all transformations ***between*** *root* and *goal*, therefore we know the transformation of *goal*

- Controller rotates joints directly, affecting orientation of child joints

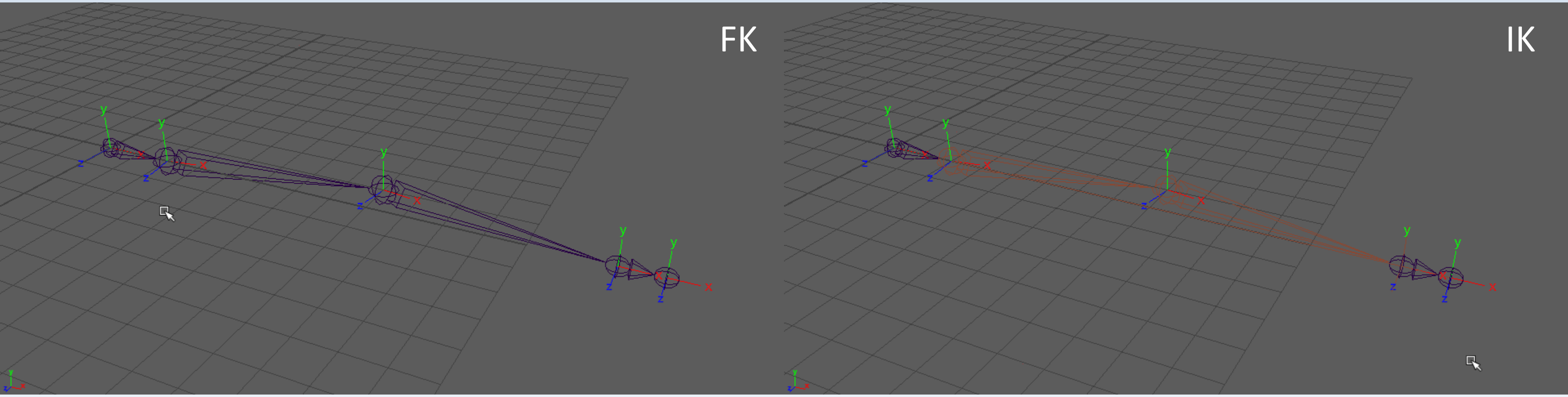
Inverse kinematics (IK): given ***only*** the *root* and the *goal*, determine the in-betweens!

- Controller changes the position of the *end effector*, influencing all of the joints between the base and the end

http://www.cs.cmu.edu/~./15464-s13/lectures/lecture6/IK.pdf

Inverse Kinematics

- FK vs. IK (Autodesk Maya):



- In terms of animating, IK is easier and faster…
- …but it's a computational nightmare… why???

Inverse Kinematics

- ***Inverse kinematics solvers***:
- Ultimately, our animation is defined by the final result (skinning, sprites)
- …and ultimately, the final result is determined using *forward kinematics* (FK)
- Therefore, *inverse kinematics* (IK) must be related to FK…

Inverse Kinematics

- ***Inverse kinematics solvers***:
- The job of an *inverse kinematics* solver:
- *Determine* ***local poses*** *such that FK will yield our end effectors'* ***world transforms***
- Corollary: IK is used to *control* FK to give us our desired output
- Here's a diagram (or two)…

Inverse Kinematics

- Our current FK-based solution:

1. Manually select or set local joint poses

$$\text{pose}_{n,t}$$

↓

2. Convert local pose to local transformation

$$^{\text{parent}}T_{n_t} = \text{convert}\left(\text{pose}_{n,t}\right)$$

↓

3. Calculate global transformations with FK

$$^{\text{world}}T_{n_t} = {}^{\text{world}}T_{\text{parent}_t}\,{}^{\text{parent}}T_{n_t}$$ (for all joints)

Inverse Kinematics

- ***Inverse kinematics-based solution***:

1. Set end effectors' global transforms — ${}^{\text{world}}T_{n_t}$ for all *effectors* → effectorList_t

↓

2. ***Calculate local poses (this is the IK problem)*** — $\text{localPoseList}_t = \textbf{solveIK}(\text{effectorList}_t\ , \ldots)$

↓

3. ***Select local pose from IK solution***, use to do FK — $\text{pose}_{n,t}$

↓

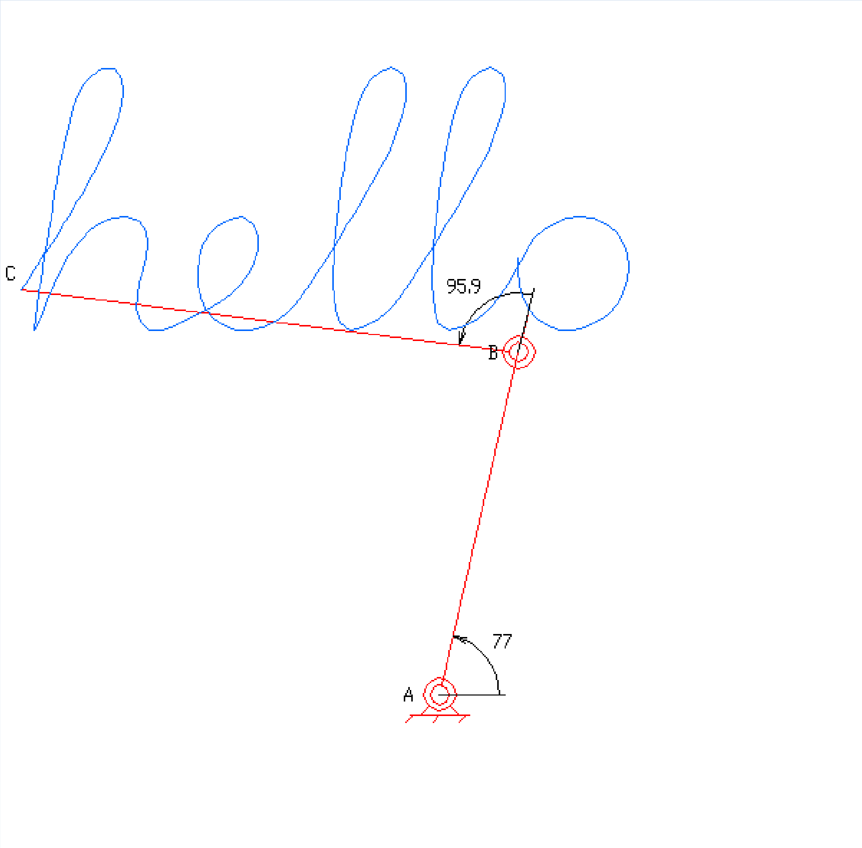
4. Convert local pose to local transformation — ${}^{\text{parent}}T_{n_t} = \text{convert}\left(\text{pose}_{n,t}\right)$

↓

5. Calculate global transformations with FK — ${}^{\text{world}}T_{n_t} = {}^{\text{world}}T_{\text{parent}_t}\,{}^{\text{parent}}T_{n_t}$ (for all joints)

Inverse Kinematics

- 2D example (with solution displayed):



- End effector follows the path ("hello")
- Joint angles (about Z axis) determined to achieve the goal of the end effector following the path
- Angles are pose data for revolute joints

https://en.wikipedia.org/wiki/SCARA

Inverse Kinematics

- Potential solutions:
- Jacobian inverse
    - https://en.wikipedia.org/wiki/Inverse_kinematics#The_Jacobian_inverse_technique
    - https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant
- Denavit-Hartenberg parameters
    - https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters
- Pure geometric solutions
    - trigonometry for the win

Advanced Mesh Skinning

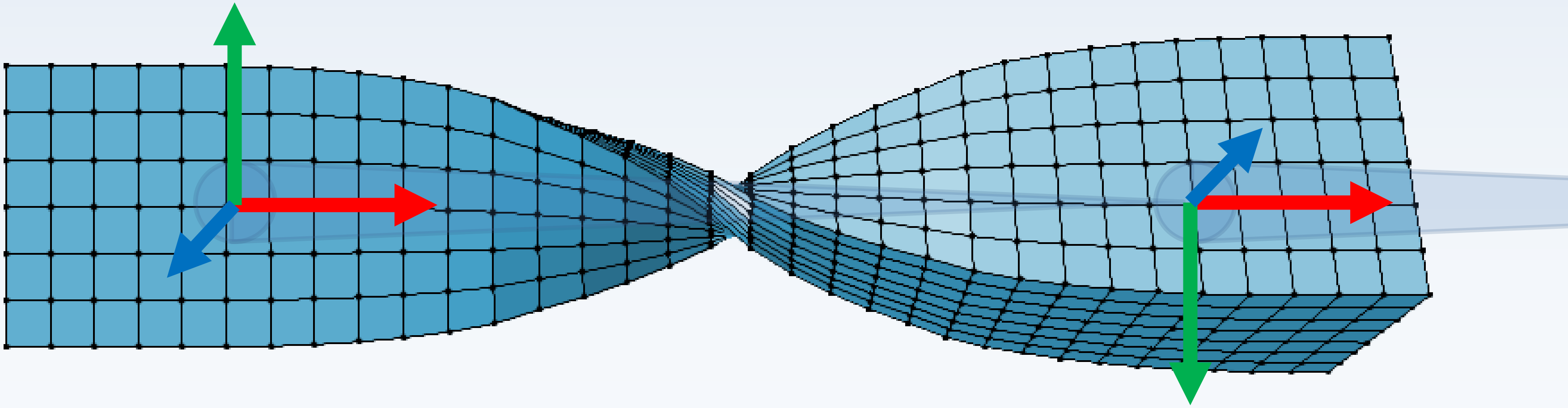
- Traditional mesh skinning has a problem:
- "***Candy wrapper effect***"
- Recall the skinning formula:

$$v' = \sum_{0 \le i < c} \left(S_{n_i} w_i v \right)$$

- Weighted sum of ***transformed vectors***
- A consequence of matrix math, the results may cancel or reduce each other…

Advanced Mesh Skinning

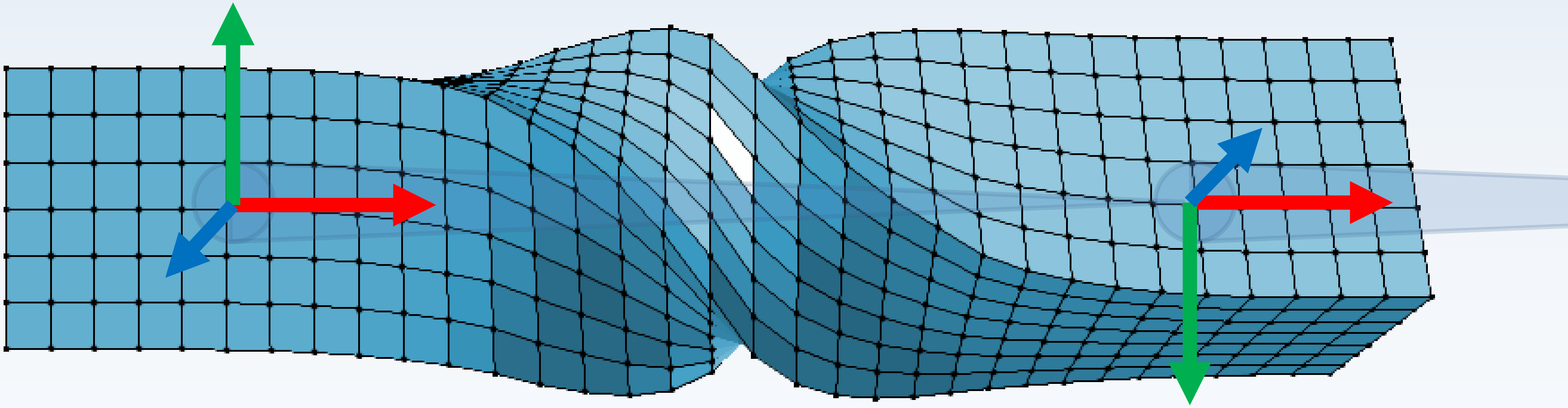
- "***Candy wrapper effect***"
- Say we have a mesh whose vertices are influenced by *two joints*…
- What happens when the transforms approach "opposites" (e.g. near 180 degrees about X)?



http://rodolphe-vaillant.fr/images/2015-06/bar_lbs.png

Advanced Mesh Skinning

- “***Candy wrapper effect***”
- This is an extreme scenario, but we generally want to preserve the volume of our mesh…
- …luckily, there is a way to do this…



http://rodolphe-vaillant.fr/images/2015-06/bar_dqs.png

Advanced Mesh Skinning

- ***Complex numbers***: before we dive right in…
- A complex number is *the sum of a real number and an imaginary number*:

$$A = a + bi$$

where

A is the *complex number*

a is the *real* part: $\mathrm{Re}(A) = a$

b is the *imaginary* part: $\mathrm{Im}(A) = b$

Advanced Mesh Skinning

- ***Complex numbers***: the rules
- The "*imaginary unit*" i has one *explicit property/rule* that it must satisfy:

$$i^2 = -1$$

- The square root of -1 does not exist, which is why we call the "solution" to this problem "imaginary"

Advanced Mesh Skinning

- ***Complex numbers***: the rules
- Fun side note: sometimes we use *j* instead…
- …but it has the same fundamental rule:

$$j^2 = -1$$

…yeah:

$$i^2 = j^2 = -1$$

;)

Advanced Mesh Skinning

- ***Complex numbers***: mathematical operators

If $A = a + bi$ and $C = c + di$, then

Sum:

$$A + C = (a + bi) + (c + di) = (a + c) + (b + d)i$$
$$\mathrm{Re}(A + C) = a + c$$
$$\mathrm{Im}(A + C) = b + d$$

Difference:

$$A - C = (a + bi) - (c + di) = (a - c) + (b - d)i$$
$$\mathrm{Re}(A - C) = a - c$$
$$\mathrm{Im}(A - C) = b - d$$

Product:

$$AC = (a + bi)(c + di) = ac + adi + bci + bdi^2$$
$$= (ac - bd) + (ad + bc)i$$
$$\mathrm{Re}(AC) = ac - bd$$
$$\mathrm{Im}(AC) = ad + bc$$

Advanced Mesh Skinning

- ***Complex numbers***: mathematical operators

If $A = a + bi$, then

Complex conjugate: $\bar{A} = a - bi$

$$\mathrm{Re}(\bar{A}) = a$$
$$\mathrm{Im}(\bar{A}) = -b$$

Absolute value/magnitude:

$$|A| = \sqrt{a^2 + b^2}$$

(result is a *real number*, calculated using Pythagorean theorem)

Product of complex number and its conjugate:

$$A\bar{A} = (a + bi)(a - bi) = a^2 - abi + abi - b^2 i^2$$
$$= a^2 + b^2 = |A|^2$$

(result is *magnitude squared*)

Advanced Mesh Skinning

- ***Complex numbers***: mathematical operators

If $A = a + bi$, then

Inverse:

$$A^{-1} = \frac{1}{a+bi} = \left(\frac{1}{a+bi}\right)\left(\frac{a-bi}{a-bi}\right)$$

(multiply by conjugate/conjugate, which equals 1, to eliminate the complex denominator, then substitute existing formulas)

$$A^{-1} = \frac{\bar{A}}{A\bar{A}} = \frac{\bar{A}}{|A|^2} = \frac{a-bi}{a^2+b^2}$$

$$\mathrm{Re}(A^{-1}) = \frac{a}{a^2+b^2}$$

$$\mathrm{Im}(A^{-1}) = -\frac{b}{a^2+b^2}$$

Advanced Mesh Skinning

- ***Complex numbers***: mathematical operators

If $A = a + bi$ and $C = c + di$, then

Quotient: $\frac{A}{C} = AC^{-1} = \frac{A\bar{C}}{C\bar{C}} = \frac{A\bar{C}}{|C|^2}$

$$\frac{A}{C} = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2}$$

$$\mathrm{Re}\left(\frac{A}{C}\right) = \frac{ac+bd}{c^2+d^2}$$

$$\mathrm{Im}\left(\frac{A}{C}\right) = \frac{bc-ad}{c^2+d^2}$$

- What’s the point of this?

Advanced Mesh Skinning

- ***Complex numbers*** are ***just numbers***.
- Try these on for size:
- ***Dual numbers***: *the sum of a real number and a nilpotent number*
- *A nilpotent number* is a *positive integer* that satisfies this explicit property/rule:

$$\varepsilon^2 = 0$$

Advanced Mesh Skinning

- ***Dual numbers***: definition

$$A = a + b\varepsilon$$

where

A is the *dual number*

a is called the '*real*' part

b is called the '*dual*' part

- They have all the same operators as complex numbers… see what they give you!

Advanced Mesh Skinning

- ***Dual quaternions***: matrices do not play nicely with animation, so here's a concept that does:
- A *dual quaternion* is a dual number whose coefficients (the real and dual parts) are actually *quaternions*:

$$\boldsymbol{A} = \boldsymbol{a} + \boldsymbol{b}\varepsilon$$

- They *also* have all the same operators as complex numbers… and quaternions… ☺

Advanced Mesh Skinning

- ***Unit dual quaternions***: the magnitude of any dual number is that of the "real" part
- Same applies to dual quaternions: if the "real" part has unit length, the dual quaternion is a *unit dual quaternion*:

$$\hat{Q} = \hat{q} + r\varepsilon$$

Advanced Mesh Skinning

- ***Unit dual quaternions***: encoding rotation *and* translation for an animatable transformation:

$$\hat{Q} = \hat{q} + r\varepsilon$$

where

Rotation is stored as $\hat{q} = w + \vec{v} \;\rightarrow\; \theta = 2\,\mathrm{acos}(w)\,,\; \hat{n} = \frac{\vec{v}}{|\vec{v}|}$

Translation is stored as $r = \frac{1}{2}\, t\, \hat{q} \;\rightarrow\; t = 2\, r\, \hat{q}^*$

(the raw translation *t* is represented here as a pure quaternion)

Advanced Mesh Skinning

- ***Dual quaternion skinning***:
- The problem with the skinning equation earlier was that the result was a *weighted sum of transformed vectors*
- Now we can do a ***weighted sum of transformations*** (because quaternions blend)
- Result is used to transform input vertex

Advanced Mesh Skinning

- ***Dual quaternion skinning***:
  - Linear blend (matrices):

$$v' = \sum_{0 \le i < c} \left(S_{n_i} \, w_i \, v \right)$$

  - Dual quaternion blend (fastest example):

$$\hat{Q}_{\text{final}} = \text{normalize}\left(\sum_{0 \le i < c} \hat{Q}_{n_i} \, w_i \right)$$

  - The skinning happens here:

$$v' = \text{dqTransform}\left(\hat{Q}_{\text{final}}, v \right)$$

Advanced Mesh Skinning

- The skinning paper:
- https://www.cs.utah.edu/~ladislav/kavan07skinning/kavan07skinning.pdf

- The beginner's guide:
- http://cs.gmu.edu/~jmlien/teaching/cs451/uploads/Main/dual-quaternion.pdf

The end.

- Questions? Comments? Concerns?