Animated characters are usually built on top of an underlying skeleton. The skeleton is a hierarchy of joints, where each joint performs some linear transformation. Each joint has one or more degrees of freedom (DOFs) that parameterize its motion. The DOF values are used to generate the joint’s local matrix $L$, which is a transformation relative to its parent matrix. The world space matrix of the joint is the local matrix times the parent’s world matrix:

$$L = L_{\text{joint}}(\phi_1, \phi_2, \ldots, \phi_m)$$

$$W = W_{\text{parent}} \cdot L$$
Skinning: Smooth Skin Algorithm

- The deformed vertex position is a weighted average over all of the joints that the vertex is attached to. Each attached joint transforms the vertex as if it were rigidly attached. Then these values are blended using the weights:

\[ \mathbf{v}'' = \sum w_i \mathbf{W}_i \cdot \mathbf{B}_i^{-1} \cdot \mathbf{v}' \]

- Where:
  - \( \mathbf{v}'' \) is the final vertex position in world space
  - \( w_i \) is the weight of joint \( i \)
  - \( \mathbf{v}' \) is the untransformed vertex position (output from the shape interpolation)
  - \( \mathbf{B}_i \) is the binding matrix (world matrix of joint \( i \) when the skin was initially attached)
  - \( \mathbf{W}_i \) is the current world matrix of joint \( i \) after running the skeleton forward kinematics

- Note:
  - \( \mathbf{B} \) remains constant, so \( \mathbf{B}^{-1} \) can be computed at load time
  - \( \mathbf{W} \cdot \mathbf{B}^{-1} \) can be computed for each joint before skinning starts

- All of the weights must add up to 1:

\[ \sum w_i = 1 \]
Weighted Blending & Averaging

- Weighted sum: \( x' = \sum_{i=0}^{w_i x_i} \)
- Weighted average: \( \sum_{i=0}^{w_i} = 1 \)
- Convex average: \( 0 \leq w_i \leq 1 \)
- Additive blend: \( x' = x_0 + \sum_{i=1}^{w_i (x_i - x_0)} \)

\[
= \left( 1 - \sum_{i=1}^{w_i} \right) x_0 + \sum_{i=1}^{w_i x_i}
\]
Shape Interpolation Algorithm

- To compute a blended vertex position:
  \[ v' = v_{\text{base}} + \sum \phi_i \cdot (v_i - v_{\text{base}}) \]

- The blended position is the base position plus a contribution from each target whose DOF value is greater than 0

- To blend the normals, we use a similar equation:
  \[ n' = n_{\text{base}} + \sum \phi_i \cdot (n_i - n_{\text{base}}) \]

- We won’t normalize them now, as that will happen later in the skinning phase
Rigging and Animation

Animation System

Pose

Rigging System

Triangles

Renderer
Rig Data Flow

\[ \Phi = \begin{bmatrix} \phi_1 & \phi_2 & \ldots & \phi_N \end{bmatrix} \]

\[ \downarrow \]

Rigging System

\[ \downarrow \]

\( v'', n'' \)
Rigging: Data Flow

\[
\begin{bmatrix}
\phi_1 & \phi_2 & \ldots & \phi_M \\
\phi_{M+1} & \phi_{M+2} & \ldots & \phi_N
\end{bmatrix}
\]

\[
L = L_{jn}(\phi_1, \phi_2, \ldots, \phi_m)
\]

\[
W = W_{\text{parent}} \cdot L
\]

\[
v' = v_{\text{base}} + \sum \phi_i \cdot (v_i - v_{\text{base}})
\]

\[
n' = n_{\text{base}} + \sum \phi_i \cdot (n_i - n_{\text{base}})
\]

\[
v'' = \sum w_i W_i \cdot B_i^{-1} \cdot v'
\]

\[
n^* = \sum w_i W_i \cdot B_i^{-1} \cdot n'
\]

\[
n'' = \frac{n^*}{n^*}
\]

\[
v'', n''
\]
DOF Mapping

- For additional control, DOF values can be manipulated within the rig in various ways
- For example:
  - A single virtual DOF can be used to control multiple real DOFs (one DOF to control the flexing of several joints in the finger)
  - A single DOF can control both the bending of a joint and the deformation of the skin around the joint
  - DOFs can be used to control arbitrary parameters such as colors, lights, texture blending, and other effects
  - One can run mathematical expressions with DOFs to generate new DOF values…
Rigging: Layered Approach

- We use a simple layered approach
  - Skeleton Kinematics
  - Shape Interpolation
  - Smooth Skinning
- Most character rigging systems are based on some sort of layered system approach combined with general purpose data flow to allow for customization
Channels

- A channel stores the animation data for a particular DOF over some range of time.
- An animation clip would contain channels for all of a character’s DOFs.
- Channels can be stored in various formats, but most formats tend to be some variation of either uniformly sampled values or keyframes.
- Keyframe channels provide a powerful user interface for interactively adjusting animation data, and so are the preferred format in most animation systems.
- When only high speed playback is desired (such as in a video game), it is often much faster not to use channels and instead store animation as an array of frames, where each frame is just an array of values—one for each DOF.
Channels: Keyframes

- We use a piecewise cubic Hermite keyframe system
- Keyframes store a time, value, tangent in and tangent out
- The spans between keyframes are 1D cubic Hermite curves
- Tangents are generated from rules (flat, linear, smooth, fixed)
- Outside of the defined time range, we use extrapolation rules (constant, linear, cycle, cycle-offset, bounce)
Channels: Hermite Curve (1D)
For each span we pre-compute the cubic coefficients:

\[
\begin{bmatrix}
    a \\
    b \\
    c \\
    d
\end{bmatrix}
= \begin{bmatrix}
    2 & -2 & 1 & 1 \\
    -3 & 3 & -2 & -1 \\
    0 & 0 & 1 & 0 \\
    1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
    p_0 \\
    p_1 \\
    (t_1 - t_0)v_0 \\
    (t_1 - t_0)v_1
\end{bmatrix}
\]
Channels: Evaluating the Cubic

- To evaluate the cubic equation for a span, we must first turn our time $t$ into a 0..1 value for the span (we’ll call this parameter $u$)

\[
u = \text{InvLerp}(t, t_0, t_1) = \frac{t - t_0}{t_1 - t_0}\]

\[
x = au^3 + bu^2 + cu + d = d + u(c + u(b + u(a)))\]
A simple animation player might work like a VCR. It can play, stop, pause, rewind, slow motion, etc.

An animation player outputs a pose, which is just an array of floats that can be used to pose the DOFs of the rig.

The pose could also be blended with other poses, or manipulated in other ways.
Blending

- One can construct an animation blending system that makes use of multiple animation players and combines the results to generate new motions.
- Blend operations take one or more poses as inputs and generate a new pose as output.
- Blend operation might also have other inputs such as control parameters (Lerp values, etc.).
- Common blend operations include: lerp (linear interpolate), add, scale, clamp, combine, mirror.
Consider a situation where we want a character to blend from a stand animation to a walk animation.
Blending: Body Turn

ADD

output pose

SCALE

f

SUBTRACT

look_right

default

walk
State Machines

- To control the sequencing of animations over time, one can use a state machine.
- Each state represents an animation, and each transition represents an event.
- The state machine is in exactly one state at any given time, and transitions are considered to be instantaneous.
- Events can be mapped to keyboard or joystick buttons for interactive control.
- Individual states don’t just have to be simple animations. They can be a network of animation blenders, or even another entire state machine...
State Machine: Jump

- **Stand**
  - JUMP_PRESS: **Stand2crouch**
- **Stand2crouch**
  - JUMP_RELEASE: **Crouch**
  - JUMP_RELEASE: **Hop**
  - NEAR_GROUND: **Takeoff**
  - **Float**
  - **Land**
- **Crouch**
- **Hop**
- **Takeoff**
- **Float**
- **Land**
Inverse Kinematics

- Inverse kinematics is a technique for posing a skeleton by specifying a set of goals.
- Goals usually specify the desired position and/or orientation of an ‘end effector’ such as the hand or foot.
- The IK algorithm automatically computes the joint DOF angles necessary to place the end effectors at their goals.
- For simple, specific chain configurations, one can use custom analytical solvers. For more general configurations, one must use a numerical solver such as a Jacobian based method or CCD (cyclic coordinate descent).
IK: Jacobians

- A Jacobian is a vector derivative with respect to another vector.
- If we have a vector valued function of a vector of variables $\mathbf{f}(\mathbf{x})$, the Jacobian is a matrix of partial derivatives—one partial derivative for each combination of components of the vectors.
- The Jacobian matrix contains all of the information necessary to relate a change in any component of $\mathbf{x}$ to a change in any component of $\mathbf{f}$.
- The Jacobian is usually written as $J(\mathbf{f}, \mathbf{x})$, but you can really just think of it as $\frac{d\mathbf{f}}{d\mathbf{x}}$. 
IK: Jacobians

\[ J(f, x) = \frac{df}{dx} = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_2} & \cdots & \frac{\partial f_M}{\partial x_N}
\end{bmatrix} \]
IK: Jacobian for a 2D Robot Arm

\[ J(e, \Phi) = \begin{bmatrix} \frac{\partial e_x}{\partial \phi_1} & \frac{\partial e_x}{\partial \phi_2} \\ \frac{\partial e_y}{\partial \phi_1} & \frac{\partial e_y}{\partial \phi_2} \end{bmatrix} \]
IK: Incremental Change in Effector

What if we wanted to move the end effector by a small amount $\Delta e$. What small change $\Delta \Phi$ will achieve this?

$\Delta e \approx J \cdot \Delta \Phi$

SO:

$\Delta \Phi \approx J^{-1} \cdot \Delta e$
IK: Basic Jacobian Technique

while (e is too far from g) {
    Compute $J(e, \Phi)$ for the current pose $\Phi$
    Compute $J^{-1}$ // invert the Jacobian matrix
    $\Delta e = \beta(g - e)$ // pick approximate step to take
    $\Delta \Phi = J^{-1} \cdot \Delta e$ // compute change in joint DOFs
    $\Phi = \Phi + \Delta \Phi$ // apply change to DOFs
    Compute new e vector // apply forward
        // kinematics to see
        // where we ended up
}
The Jacobian matrix is rarely square, and even if it were, it might be singular and non-invertible. Therefore, one must rely on more sophisticated techniques other than simple matrix inversion. Possible alternatives include:
- Pseudo-inverse
- Single value decomposition
- Jacobian transpose
Each step of the Jacobian method moves the end effector closer to the goal. The algorithm must be iterated several times in order to converge to a final solution.

There are several reasons to stop iterating:
- End effector successfully reaches goal
- Stuck in a local minimum
- Taking too long
IK: Cyclic Coordinate Descent

- CCD is an alternative technique to the Jacobian methods.
- Instead of using linearized approximations to make small steps, the CCD method uses inverse trigonometry functions to solve joint values more directly.
- Each iteration of the CCD algorithm is more expensive than an iteration of the Jacobian transpose, but the CCD method usually requires far fewer iterations to converge to a solution.
- It is faster, but tends to produce poorer quality motion, as it tends to favor joints closer to the end effector.
For specific chain configurations, one can implement custom analytical IK solvers.

Analytical solvers rely on heuristics and direct solutions to matrix and trigonometric equations (i.e., they use a lot of matrix inversion and inverse trig functions).

These methods can be extremely fast and well behaved, but their main drawback is their lack of generality. They also become more and more difficult to implement as the chains become more complex.