Methods of Motion Generation

- Automatic Discovery (High-Level Control)
- Modeling/Simulation (Physics, Behaviors)
- Performance Capture (Motion Capture)
- Traditional Principles (Keyframing)

High-Level Control (I)

Task level description using AI techniques:
- Collision avoidance
- Motion planning
- Rule-based reasoning
- Genetic algorithms
  ... etc.

High-Level Control (II)

Advantages
- Very easy to specify/generate motions
- Can reproduce realistic motions

Problems
- Need to specify all possible “rules”
- The intelligence of the system is limited by its input or training
- May not be reusable across different applications/domains

AI

- General CS audience: machine/deep learning
- General public: fictionalized super-smart machines
- Game devs: virtual agents (VAs)
Bad AI makes VR/AR users very uncomfortable...

- E.g. Virtual agents walking through you
  - "Evaluating Collision Avoidance Effects on Discomfort in Virtual Environments" Sahra 2017
  - We’re naturally inclined to avoid an immersive virtual agent

Bad AI makes VR/AR users very uncomfortable...

- Eye Contact can freak users out
  - E.g. too much is creepy, too little is isolating (like IRL.)
- Can also be a good thing
  - E.g. therapy for autistic kids usually involves eye contact with virtual characters

Another example (Social cues for Autism therapy)

Uncanny Valley

- VA is relatively realistic but something is very off...
  - Causing uncanny valley usually worse than just having unrealistic-looking characters
  - Often caused by face movements, body motion, eye contact, talking like robot, etc.
  - Motion generally makes things worse, we’re really good at perceiving inaccurate human motion

Handling Uncanny Valley....

- Trends....
  - (Old) Hand-craft everything really well.... Hasn’t really worked thus far & very tedious/unscaleable
  - (Modern) Directly replicate real human motion (animation & tracking)
  - (Future) Learn what real human motion is (deep learning...especially GANs & RNNs) & extrapolate

Constructing AI

- Factors to consider (topics for next few weeks)
  - Collision avoidance (how to stop them from walking through each other & the player)
  - Social cues (what happens when they get close to another person.... How does it affect trajectory)
  - Walk pattern/gait (how do they walk naturally)
  - Behavior (what do they do when)
  - Animation (what else are they doing visually besides walking)
  - Constraints
    - Navigational (how to limit where they walk in VE)
    - Kinematics (body motion, interactions with other dynamic bodies)
  - Networking (how to handle multiple REAL users)
  - Body Representation, Avatars, Anonymity (how to abstract bodies while keeping features like gait)
Compare the body representation:

Compare the physical interactions:

Recall: “Immersive VR Self” Schwartz 2018 (Oculus)

- Elements of the accurate virtual avatar
  - Visual: perspective-correct visual representation
  - Audio: spatialized sounds
  - Generalize to accurate audio field, good HRTFs, traversal, etc,
  - Movement: physical body gestures
  - Gestures are not the only important factor

What makes Boneworks, Asgard’s Wrath, & Half-Life Alyx different from older VR games?

- Overall graphical quality is better, sure…. But there are fundamental differences
  - Body is represented in Boneworks & Asgard’s Wrath despite only having controllers
  - Skeletal rigging + inverse kinematics (IK) from robotics

What makes Boneworks, Asgard’s Wrath, & Half-Life Alyx different from older VR games?

- Overall graphical quality is better, sure…. But there are fundamental differences
  - Physical interaction with most objects in VE
    - IK necessary for good collision meshes (e.g. shot in arm with arrow)

Methods of Motion Generation

- Automatic Discovery (High-Level Control)
- Modeling/Simulation (Physics, Behaviors)
- Performance Capture (Motion Capture)
- Traditional Principles (Keyframing)
Physically-based Simulation (I)

- Use the laws of physics (or a good approximation) to generate motions
- Primary vs. secondary actions
- Active vs. passive systems
- Dynamic vs. static simulation

Physically-based Simulation (II)

Advantages
- Relatively easy to generate a family of similar motions
- Can be used for describing realistic, complex animation, e.g. deformation
- Can generate reproducible motions

Problems
- Challenging to build a simulator, as it requires in-depth understanding of physics & mathematics
- Less low-level control by the user

Methods of Motion Generation

- Automatic Discovery (High-Level Control)
- Modeling/Simulation (Physics, Behaviors)
- Performance Capture (Motion Capture)
- Traditional Principles (Keyframing)

Motion Capture (I)

1. Use special sensors (trackers) to record the motion of a performer

2. Recorded data is then used to generate motion for an animated character (figure)

Motion Capture (II)

Advantages
- Ease of generating realistic motions

Problems
- Not easy to accurately measure motions
- Difficulty to "scale" or "adjust" the recorded motions to fit the size of the animated characters
- Limited capturing technology & devices
  - Sensor noise due to magnetic/metal trackers
  - Restricted motion due to wires & cables
  - Limited working volume

Methods of Motion Generation

- Automatic Discovery (High-Level Control)
- Modeling/Simulation (Physics, Behaviors)
- Performance Capture (Motion Capture)
- Traditional Principles (Keyframing)
Keyframing (I)

1. Specify the key positions for the objects to be animated.
2. Interpolate to determine the position of in-between frames.

Keyframing (II)

Advantages
- Relatively easy to use
- Providing low-level control

Problems
- Tedious and slow
- Requiring the animator to understand the intimate details about the animated objects and the creativity to express their behavior in key-frames

What is 3D Animation?
- Thus far we’ve been using static meshes; 3D models that always look the same
- Some things require motion (e.g. human moving, talking, etc.)
- Some motions are predefined (always the same; e.g., this jump)
- Some motions are functions/graphs/FSMs
  - E.g., crouch/walk/run/sprint
- Some motions are physics-based/constrained (e.g., ragdoll, IK)
- Animation methods describe these motions

Common Animation Methods
- Geometry Cache
- Simple transformations
- Higher-order transforms (splines)
- Vertex Animation/Morph Targets/Shapekeys/Blendshapes
- Skeletal Animation
- Inverse Kinematics (IK) & Biomechanical Constraints

Simplest Method (at High Level): Geometry Cache
- Closest method to 2D sprite animations
- Single mesh for each frame
  - A frame is a segment of the animation (e.g., 120 frames/second)
- Store it in a single file that swaps between them
- In 2D, all videos do this
  - E.g., mp4, flv, avi, mov, gif, webm
- In 3D, we have other formats for this
  - E.g., abc, Alembic; file format by Sony & LucasFilms

Geometry Cache Pros
- Can represent arbitrary animation sequences
- Easy to understand
- Last resort for animations not describable by other methods
  - E.g., physics: fluid sim, baked destruction, etc.
- Describe very detailed animations (e.g., in movies)
Geometry Cache Cons
- Tend to be very slow in runtime
  - Much harder to swap between meshes than 2D frames; recall that GPUs traditionally aren’t designed for 3D data
- Very resource intensive
  - E.g., our 1 Min animation here took 40gb RAM & 80GB disk
    - 20GB to describe each mesh frame
    - 20GB for baked Alembic animation
    - 20GB initial UE4 import (convert to asset)
    - 20GB cached version in Intermediate folder
- Game engines didn’t support for a long time
- Usually only used as last resort or for impossible anims

Moving On...
- Obviously, geometry caches aren’t a great option for realtime game engines with memory limitations
- Important optimizations came about…
- Generally:
  - Stick to keyframe-based methods for individual animations
  - Interpolate/blend between individual animations
  - We’ll discuss these topics separately

Keyframes for simple transformations
- **Keyframes** are frames in which a part of transform is explicitly defined
- Interpolate smoothly between keyframes (linearly, sine, etc.)
- We’ll see a Blender example

Motion Interpolation
- **Interpolate using mathematical functions:**
  - Linear
  - Hermite
  - Bezier
  ... and many others (see Appendices of Richard Parent’s online book)
- **Forward & inverse kinematics for articulation**
- **Specifying & representing deformation**

Higher-Order Trajectories: Spline Animation
- Higher-order version of simple transform animations
- Can be applied as some part of almost any other anim method
- Used in 3D games to have things move along pre-defined trajectory
- Gives impression of good navigation when real-time navigation unnecessary
- Often used for flying things and cameras (e.g., simulate camera dolly)
- Spline itself can be recomputed each frame to give more dynamic appearance

Skeletal Animation
- **General idea:**
  - Create a skeleton/character rig that represents bones of mesh
  - Position bones to be inside mesh for reference pose (e.g., T-pose, A-pose)
  - Define how much each bone affects each vertex of mesh (aka “weights”). aka “skinning” the mesh
  - E.g., shoulder vertices deform during head & shoulder motion
  - Define animations as transforms of bones
  - Like blendshapes; animations can be described as weights + deltaTransforms; except much fewer
    deltaTransforms required for skeletons b/c only transforms of bones, not vertices
  - Baking skeletal anim as transforms of vertices achieves effect of blendshapes
    ... as we’ll see, they often using in unknown
Skeleton Tree Structures

- Bone head/tail
- For game engines:
  - Single root, usually a bone near the abdomen (e.g., Mixamo uses hip)
  - Typically use tree structure like on the right
  - Works same way as parenting system for GoGIActors
- Children follow parent but not vice versa

Try to use only local deltaRotations

- Bone joint position typically doesn’t change
  - e.g., upper and lower leg always joined by knee… will not diverge at that point
- Translating bone can cause awful mesh distortions
- Thus, typically describe animations as only rotations, esp. for humans
  - (very few bones in human body are translational/prismatic)
- Stick to local space b/c skeleton can be oriented many ways in game world
- How to limit how much bones can rotate?
  - Kinematic constraints

Skeletal Animation Pros

- One of the least-CPU-intensive methods
- Bones are natural way of thinking of many anims like body motion
- Can describe very complex motions
- Natural option for tracking systems which only return a few points (more later)
  - This covers the majority of consumer trackers… E.g., Kinects, Leap Motion, etc.
- W idely supported, unlike blendshapes & geometry cache
- Allows for nice tree based optimizations
- Allow for physics-based extensions like ragdolling
- Can share skeletons/anim (not possible w/ other methods)
- Can describe anims w/ only skeletons (no need for mesh), great for tracking/mocap & applying to other mesh

Skeletal Animation Cons

- Rigging quality is inconsistent b/w 3D modelling programs
- Getting weights right is really hard
  - Fortunately, tons of auto-riggers for human meshes (e.g., Mixamo)
- Can’t easily describe things similar to muscle movement (e.g., face)
- Can be hard to separate rigid/non-rigid parts
- Parenting structure in game engines means skeleton should only describe individual animated objects (bad for large-scene reconstruction)

Ragdolling

- Often combination of skeletal anim+physics engine
- Treat bones like edges/control points of cloth with mesh collider
  - E.g. bunch of rigid bodiies attached to each other, limited by skeleton

Kinematics: study of motion independent of underlying forces

- Methods for figuring out the transform of different components of parenting structure (usually a tree)
- Some terms:
  - Rigid body: a non-deformable object constrained by physics
  - Manipulator: a device capable of manipulating things in the environment
    - Usually a kinematic chain of components (usually rigid bodies) linked by joints
    - Controllable joints are actuators (e.g., knuckle is, middle proximal are not)
    - E.g., robot arm, human arm… Usually structures representing limbs
  - End effector (EE): final device in manipulator chain
    - E.g., robot claw, human hand… the leaf of the bone tree
Forward Kinematics (FK)

- Given the transforms of parent joints, where is the end effector?
- Motion of all joints is explicitly specified
- Very similar to the method to get world transform of GO/Actor from local transform of this child & parents

Recall: From Local/Relative Space to Global/World Space

- Put very simply, if 3D model is the node of a tree:
  - traverse upwards through tree, adding all relative location & rotation, multiply scales
  - Stop after reaching world origin (aka root)
  - (transform of root relative to itself is (position=(0,0,0), rotation=(0,0,0), scale=[1,1,1])...so you can keep iterating but the result won't change)

Forward Kinematics (FK)

- Simple enough… no need to know all parent joint transforms
- Can we animate a VR avatar with FK?
  - Not with most commercial devices which only track HMD & controller... We don’t track shoulder or elbow
  - Devices like Quest can only see in front of you + some peripherals
  - Could technically use camera to figure out other joint positions... But tracking tech isn’t quite there. Extra cameras:
    - Are expensive
    - Can hurt ergonomics
    - Require specific placement
  - Use a lot of hardware resources
  - Would still suffer from occlusion... What happens if elbow gets blocked?
  - Need to be egocentric... We’re moving away from external devices & external trackers to minimize equipment setup
- Need smarter methods of predicting the parent transforms….enter IK

IK: Working with only EE + constraints

- Same structure as prev example except now know where the red cylinder is & constraints on parents
- Which one is the end effector (EE)?
- Inverse Kinematics (IK): given the position of the end effector, find the position and orientation of all joints in a hierarchy of linkages; also called “goal-directed motion”
  - Know global and/or local transform of parents (and/or local transform of EE)
  - Which 1 we want specifically depends heavily on API… e.g. in game engines might convert from world space -> local space of camera but not necessarily skeletal space…more later

Rotation Example

- More practical for animation
- Given this parenting structure & set of constraints, what are angles at joints, where is the white joint?
- Similar to human arm kinematic chain
- Answer might not be intuitive yet…. But it will be. It’s surprisingly simple geometry!

These questions only get more complex...

These AI are basically only given translation constraints bw rigidbodies... they learn rotations needed for actions. They learn the rotational constraints over time (becomes clear how their body is oriented over time when it’s originally random)
Inverse Kinematics (IK)

- Addresses limited knowledge of joint transforms
- Used to solve many problems in complex kinematic systems
- Similar to system of equations where variables are each component of transform
- Can have 0, 1, a range, or infinite solutions
  - Use constraints to adjust this

Solving IK

- Can figure the parent transforms out analytically or iteratively
  - Analytically: Find some solution by deriving it from valid relevant equations
    - E.g. compute derivatives/Jacobians & set to 0
  - Iteratively:
    - Define error (e.g. deltaRotation too high, breaks constraints)
    - Make a guess
    - Calculate error on the guess
    - Make another guess that results in less error (usually with gradient descent)
    - Rinse & repeat
- Can mix & match the methods; e.g. some parameters are easy to find analytically but might result in free/bound variables which require iterative methods
  - Free variable: completely unknown; can choose any value (infinite solutions)
  - Bound variable: Variable with a range of possible solutions (possibly including 0 solutions)
- In graphics, analytical calculations are usually geometric…. Way fewer parameters than in ML so it’s more feasible to do it this way
  - Also have the benefit of parameters have a graphical definition & intuitive constraints

Long-time preferred iterative method: CCD

- Cyclic coordinate descent
- Coordinate descent similar to gradient descent except it updates each axis in sequence instead of simultaneously
- Basically keeps rotating each joint by a bit until EE is as close to target EE position as possible (like reaching for a wall in pitch darkness)

Problem with many human-centric IK methods….

- Often do not consider constraints or continuity…. E.g. CCD known for sudden rotation jumps/strange poses
- Large range of proposed solutions is impossible anyway given degrees of freedom (dof) of a joint
- Our limbs can’t move with rotational flexibility of most robot arms
- Important consideration as VR industry moves forward: biomechanical constraints

In game engines: FABRIK algorithm (2011)

- Forward-and-Backward Reaching Inverse Kinematics (FABRIK)
- Improvement on CCD which better considers continuity & constraints
- Treats rigidbodies like line segments instead of rotations
  - Knowing what you know about skeletal animation…. What limitations are there?
    - We’ll need to eventually calculate the bone rotations anyway….
- First make reasonable prediction about what the parent joint positions are based on end effector position (backwards step)
- Then, start from the parent and move towards the end effector, calculating corrected points
- Use previous frame joint positions as initial guess to ensure smoothness & continuity in motion

FABRIK performance compared
Rough FABRIK process
1. Make sure it's actually possible to reach that point given rigid body lengths. Otherwise, fully extend limb:
   a. e.g., wrist can't be farther from shoulder than upperArmLength + forearmLength
   b. e.g., ankle can't be farther than hip than thighLength + shinLength
2. Backwards step:
   a. Create normalized vectors from current joint to parent joint (let's call it dirB)
   b. Multiply dirB by length from current joint to parent (e.g., forearmLength: wrist to elbow) (call it deltaB)
   c. currentJointLocation + deltaB to get location of parent (call it locP)
   d. locP will be used for next iteration as the currentJointLocation
   e. Do this until reaching root of limb
3. Unless extremely lucky (or if the EE didn't move), the root location won't be in the right place from just the backwards step. So we go in reverse with forwards step:
   a. Create normalized vectors from current joint to child joint (let's call it dirF)
   b. Multiply dirF by length from current joint to child (e.g., upperArm: length: shoulder to elbow) (call it deltaF)
   c. currentJointLocation + deltaF to get location of parent (call it locC)
   d. locC will be used for next iteration as the currentJointLocation
   e. Do this until reaching EE
4. Repeat the process until backwards & forwards step converge (e.g., backwards step root position is very close to real root position or forwards step very close to actual EE)

Rough FABRIK process (pseudocode only)
→ if (EE inImbRoot.magnitude >= 2/(rigidBodyLengths))
   → Parent joints are placed along the line from ImbRoot to EE based on rigidBodyLengths
→ else
   → def Backwards:
      → For each joint starting at EE ending at joint right before limb root
         → dirB = (currentJoint.location - currentJoint.location).normalized
         → deltaB = dirB * lengthOfRigidBodyFromCurrentJointToParent
         → locP = currentJoint.location + deltaB
         → Place parent at locP
         → currentJoint becomes currentParent
   → def Forwards:
      → For each joint starting at EE ending at joint right before limb root
         → dirF = (currentChild.location - currentJoint.location).normalized
         → deltaF = dirF * lengthOfRigidBodyFromCurrentJointToChild
         → locC = currentJoint.location + deltaB
         → Place child at locC
         → currentJoint becomes currentChild
→ while (EEAsDeterminedByForwardStep - EE.location).magnitude < ε
   → Backwards
   → Forwards

Great visual tutorial

Biomechanical constraints
- Think of your skeleton like a tree as in skeletal animation
- Try holding the "parent" bone (edges) static and see how much you can move the child joint
- Assume bones can only rotate, simplifies constraints & calculation a lot and is a good enough approximation unless simulation must be perfectly accurate
- E.g., tarsals technically translate... but how often do you see characters feet?
- Limiting def allows for stronger, faster analytical & geometric methods
- These will motivate IK constraints

Biomechanical constraints
- How many dof do these joint types have?
  - Pelt: 1
  - BeltAndGusset: 3
  - Hinge: 1
  - Saddle: 2
  - Conoid: 2
  - Plano: 2
- How much does this reduce system complexity? (e.g., arm)
  - No constraints (rotation & translation): 8
  - No biomechanical constraints (only rotations):
    - Shoulder: 6, elbow: 6, wrist: 6 = 18dof
  - With biomechanical constraints:
    - Shoulder: 3, elbow: 3, wrist: 3 = 9dof
  - % of the original variables to solve for!
  - Much fewer constraints to worry about
  - Both technically more feasible AND motion will be more natural!
  - Complexity is at least O(numIter/ToConverge*numJoints)
Solving Human Arm IK

- In VR, a simple task is figuring out elbow location from just HMD & controllers so the player can have a mesh.
- Any ideas how we can start? Simplify to 2D for now (looking at character profile, so 1 dof rotations... up/down rotations are all that matter for now)
  ○ Have a target mesh
  ○ Calibrate rigid body lengths
    - For the arm, need to know upperArm.length & forearm.length
  ○ Need neck height & shoulder width to predict shoulder position
  ○ Figure out biomechanical constraints of skeletal mesh
  ○ Make assumption that shoulder will rotate with head rotation by some amount or shoulder rotation determined by using both controllers somehow... These would be very rough assumptions

Biomechanical constraints

- Figure out biomechanical constraints (assume angles calculated from horizontal of parent bone (local x axis))
  ○ Shoulder can move from (-155°, 75°)
  ○ Elbow can move from (-90°, 150°) (meaning only bend elbow up)
  ○ Wrist can move from (-45°, 45°)

Solving Human Arm IK in 2D

1. Estimate shoulder position, not rotation, for now
2. Assume rotational of controller is actually the wrist rotation with some translational offset (pretty much always the case)
3. Use law of cosines to figure out all angles in world space
   a. 2 side lengths are the rigid body lengths, 1 of the sides is distance from shoulder to wrist (SSS triangle)
   b. Biomechanical constraints are useful for figuring out when these happen (e.g. if elbow suddenly bends other way). Also probably some 90 degree rotation offsets
4. Really useful & valid assumption in VR: the VR user cannot move the controller in orientations the human body wouldn’t allow. Thus, we don’t need to try to constrain them... We’re not robots
   a. Assuming they’re actually holding the controller
   b. Biomechanical constraints in 2D used to limit angles, but prediction not really necessary due to simplicity

Solving Human Arm IK in 3D

1. Biomechanical constraints vital for 3D due to prediction
2. In 3D...
   a. Need biomechanical constraints for all dof
   b. Often, each axis handled by projecting to local 2D planes & using law of cosines/other geometry
   c. Usually, at least 1 dof (joint need to be predicted iteratively & something like FABRIK)
     i. E.g. Maybe I can get elbow location from profile... But then how do I know left/right rotation?
     1. Need to guess... But fortunately arm lengths help restrict rotational range
   d. Hardest part tends to be converting to rotational constraints that look ok in skeletal mesh

Game engines already have IK solvers

- Most calculations unnecessary to do manually
- Can specify constraints programatically or in 3D modelling program during mesh creation (readable by game engines)
- Unity, UE4, & most other game engines have some function for skeletal/skinned meshes automatically handling all the bone rotations if EE is moved
- (lots of great tutorials... better than what I can probably show)

Some resources on IK:

- https://www.youtube.com/watch?v=1llQ5Za9VJk
- https://www.youtube.com/watch?v=Qs6Qz4G1yXM
- https://www.youtube.com/watch?v=48Za9VJk
- https://www.youtube.com/watch?v=48Za9VJk
- https://www.youtube.com/watch?v=48Za9VJk
- https://www.youtube.com/watch?v=48Za9VJk
- https://www.youtube.com/watch?v=48Za9VJk
**Vertex Animation/Morph Targets/Shapekeys/Blendshapes**

- Better-optimized animation using only shape keys (usually extreme transforms)
- Most commonly used for facial animation & lip sync
- Basic idea:
  - You have a default/basis pose for a driven mesh
  - Each blendshape or target pose is an extreme (linearly-interpolated) transformation to vertices of basis pose
  - Have a scale of [0, 100%] describing how close to blendshape face is
  - Combination of these scales allows for complex animations
- Blender example

---

**Fun Example of Bad Facial Animations**

They probably tried to make the face align with statistical estimation rather than blendshapes...

---

**How to actually make the blendshapes?**

- Usually done w/ facial skeletal bones
- Sometimes done manually (less feasible nowadays)
- Sometimes done with tracking (a topic for the AR lectures)
- Sometimes done w/ reconstruction w/ markers (topic for later)
- Recent trend is to use CV/GANs to figure them out

---

**Important Modern Use of Blendshapes in AR/VR: Lip Sync**

- General idea:
  - Make blendshapes for extreme facial expressions (e.g. mouth open/close)
  - Define phonemes (extremes in possible syllables) as linear combination of these blendshapes
  - Sequence of graphemes makes a word, word -> sentence, etc.
  - Can start with NLP word recognition for real-time voice chat
- Game engine details
  - Blendshapes in Blender, Maya, etc., stored by FBX
  - Unity/UE4 natively read blendshapes
  - Blendshapes + skeletons can work simultaneously on same mesh (not possible for other methods, more later)
  - Blendshapes used for non-skeletons (accessories, hair, etc.)

---

**Blendshape Application: Eyes/Blinking**

- Blendshapes for closed/open eyes
- Blendshapes for intermediate steps (e.g. try making blend less linear)
- Eye-tracking itself normally done with skeletons

---

**Emotions**

- Make blendshapes for primary emotions
- Nice example (from a student’s work!) here: https://tayarko.github.io/emo
  - Also where I copied fig. 2 from the right from
- These images from “Making Comics: Storytelling Secrets Of Comics, Manga, and Graphic Novels” by Scott McCloud
For realtime performance....

- Especially for tracking real faces
- Often use ML to learn blend weights after learning some faces
  - A detailed face pose is a linear combination of blend weights!
- Good, simple example is Olszewski 2018: “High-Fidelity Facial and Speech Animation for VR HMDs”

Blendshape Pros

- Can represent any animation step with linear movement of vertices
- Very efficient
  - Only requires mesh and delta locations of the vertices for blendshape
- For facial animation, arguably the best results and easiest to accomplish facial anims with Blendshapes themselves typically natural poses
- Excel in situations where anim sequence not known, but range of motions is
  - E.g. voice chat: don’t already know what someone will say, but know what a syllable looks like
- Also describes very detailed animations (e.g. in movies)

Blendshape Cons

- Can be very hard to make blendshapes (often use trackers/reconstruction)
- Inconsistent support between platforms
- Can only represent linear motions between vertices (traditionally; this is changing)
- Can have bad performance for highly-detailed meshes
- Requires that order and # of vertices is the same
  - E.g. if you reconstruct a real person’s extremities, reconstruction cannot guarantee vertex order, so blendshapes not possible here
- Usually don’t perform well or are overkill for less-detailed anims

Why we use blendshapes & skeletons simultaneously

- Skeletons are typically animated
- Animation often baked into FBX (e.g. jumping); blend/FSM in game engine
- Body motions rarely have dependency on face (e.g. expression while walking)
- So if face anims done with skeleton, there can be computationally-expensive or ugly conflicts when fighting with pre-defined anim
- Fighting with longer pre-defined anims also very hard, esp. For complex skeletons
- At high-level, we’re decoupling face and body and using the method that works best for each one individually to avoid problems/get best overall result
- Physiological they’re fundamentally different: facial anims controlled by muscles, body controlled by bones controlled by muscles
- Skin of rest of body usually only deforms heavily at joints…. So blendshapes overkill for most of the body anyway
- Can use skeleton to create blendshapes!
  - E.g. check out my video on this: https://www.youtube.com/watch?v=4kzPM11cV10

Face Skeletons

- Use bones for control points (usually landmarks)
- More advanced libraries (e.g. Facett!) categorize control points as primary/secondary/rigid
- Good for when the target pose is not known or we want unrestricted motion
- Usually hard to define constraints & ensure face shape maintained

Good Human-Making Libraries

- Mixamo by Adobe
  - Has both character & animations
  - All characters share a skeleton
  - Nice feature where you can upload your own mesh & they’ll rig it for you
  - Textures/materials pretty good as well
  - Works well w/ both Unity & UE4
  - Probably easiest for A10

- MakeHuman
  - For custom characters
  - Has more advanced skeleton w/ facial bones
  - Clothes options
  - Not particularly easy to use
Human-Making Library For Research: SMPL

- Similar to MakeHuman but easier to get working outside SMPL (e.g., Unity, Blender, Maya)
- Human rigging/weights much more realistic than MakeHuman, handles different body shapes better
- Works better w/ blendshapes than MakeHuman
- Can be a bit hard to get working... but tons of parameters

Research in 3D Animation

Motion Capture (Mocap)
- Transfer video of a person/creature moving and convert to animation
  - Body motions \rightarrow\text{skeletal animations}
  - Face motions \rightarrow\text{usually blendshapes}
- A couple of methods to do this (next few slides)
- Transferring of markers to skeletal joints?
  - Pretty poorly-documented... Usually the tracker API itself does it internally & data sent to 3D engine
  - Often some kind of solver that fits skeleton to points reconstruction-style
  - Example: https://support.realtimezm.com/documentation/notes/tracker_notes/mocap_skeleton.html

Active Markers
- E.g., blinking lights, infrared (IR), etc.
- Used for markers not easily distinguishable otherwise or very complex animation needs
  - E.g., if there are other light sources that interfere
- Limitation: require power source & lots of wires which can limit motion

Passive Markers
- E.g., retroreflective surfaces (meaning they reflect light that gets shot at them), e.g., Spheres
- Infrared lights usually used to avoid light pollution/interference
- Luminous usually trivial to track w/ APIs like OpenCV
- Optic track is very common, mocap balls + IR cameras surrounded w/ IR light sources
- Still lots of equipment, but no need for wires on the person
- Great for tracking arbitrary objects (e.g., VR headsets use these for Idaho)

More on VR Tracking: LaValle '14: “Head Tracking for the Oculus Rift”
- Great read if you’re interested in tracking
- Key points:
  - VR tracking is not done with 1 tracking method
  - Gyroscope gives rough 3dof rotation
  - Tilt correction is done with accelerometer which detects gravity direction
  - Drift correction is done with filters
  - Yaw correction is done with magnetometer, sort of like a compass
  - Predictions are done for further corrections
Monocular Depth Cues

Estimating depth with 1 eye/camera

This is OK for 3D IF we can project to 2D surface representation (as with UV maps)

- Occlusion
- Relative size
- Cast Shadows
- Shading
- Distance to horizon
- Texture gradient
- Linear perspective

Rough markerless tracking method

- Know features of face
- Feature matching with CV
- Feature locations on mesh usually defined as control points of mesh/rig
- Movement of features projected from 2D image space to 3D
- Movement along depth axis usually handled w/ monocular cues, ML (e.g. which depth movements are natural), or depth sensor
  - Fortunately, face control points don't move that much in depth... mostly move along basis pose surface definition (e.g. muscles pull along skull... not INTO skull)

Markerless

- Done by feature-matching/tracking with computer vision
- Methods themselves are pretty complicated but you can read the papers
- Benefit: much less equipment required; usually just need single camera
- Human faces pretty easy b/c features pretty well-defined & lots of datasets & symmetry
- Gets into reconstruction: some cool examples in a bit

Importance of Markerless Tracking in XR

- Can work with much less equipment (usually only need 1 camera + lighting)
- Can work with weaker devices (e.g. face reconstruction since iPhone X)
- Tracking environment
  - Finding correspondence between physical & virtual environments (PE & VE)
  - VR: mostly used for inside-out tracking (like Oculus Quest's)
  - AR: mostly used for alignment of worlds
- Tracking people
  - Bodies & skeletons: e.g. Kinect, PSEye
  - Faces: e.g. single-view cameras, embedded MMO sensors
- Reconstruction Recreating person as 3D model (prescan)
  - Recreating person's complex motions (e.g. blinking, interacting with something else)
  - Recreating PE to be used in VE

Markerless/Reconstruction Examples: KinectFusion

Some Markerless Tracking/ Reconstruction Examples

(plus note that those generally don't track the skeleton... just some cool examples w/ similar methods)
Human Reconstruction

Markerless/Reconstruction Examples: “High-Fidelity Facial and Speech Animation for VR HMDs”

Markerless/Reconstruction Examples: EgoCap

Markerless/Reconstruction Examples: DynamicFusion

Markerless/Reconstruction Examples: “High-Quality Streamable Free-Viewpoint Video”

Markerless/Reconstruction Examples: “3D Scanning Deformable Objects with a Single RGBD Sensor”
Markerless/Reconstruction Examples: Holoportation

Markerless/Reconstruction Examples: “Towards Fully Mobile 3D Face, Body, and Environment Capture Using Only Head-worn Cameras”

Room Reconstruction

Markerless/Reconstruction Examples: BundleFusion

Markerless/Reconstruction Examples: RoomAlive

Markerless/Reconstruction Examples: “Semi-Dense Visual Odometry for AR on a Smartphone”
Physically-Based Animation

Procedural & Physically-Based Animation
- Animation that changes in response to runtime constraints (collisions, space constraints, gravity, etc.)

Different Bone Representations
- Allows for more complex physically-based trajectories

Affective/Emotive Animation

Affective/Emotive Animation
- Distort base animation/trajectory to give impression of emotion
- Character gestures based on emotional content