Chapter 5.2
Character Animation

Overview

- Fundamental Concepts
- Animation Storage
- Playing Animations
- Blending Animations
- Motion Extraction
- Mesh Deformation
- Inverse Kinematics
- Attachments & Collision Detection
- Conclusions

Fundamental Concepts

- Skeletal Hierarchy
- The Transform
- Euler Angles
- The 3x3 Matrix
- Quaternions
- Animation vs Deformation
- Models and Instances
- Animation Controls

Skeletal Hierarchy

- The Skeleton is a tree of bones
  - Often flattened to an array in practice
- Top bone in tree is the “root bone”
  - May have multiple trees, so multiple roots
- Each bone has a transform
  - Stored relative to its parent’s transform
- Transforms are animated over time
- Tree structure is often called a “rig”

The Transform

- "Transform" is the term for combined:
  - Translation
  - Rotation
  - Scale
  - Shear
- Can be represented as 4x3 or 4x4 matrix
- But usually store as components
- Non-identity scale and shear are rare
  - Optimize code for common trans+rot case

Euler Angles

- Three rotations about three axes
- Intuitive meaning of values
- But… "Euler Angles Are Evil"
  - No standard choice or order of axes
  - Singularity "poles" with infinite number of representations
  - Interpolation of two rotations is hard
  - Slow to turn into matrices

3x3 Matrix Rotation

- Easy to use
- Moderately intuitive
- Large memory size - 9 values
  - Animation systems always low on memory
- Interpolation is hard
  - Introduces scales and shears
  - Need to re-orthonormalize matrices after

Quaternions

- Represents a rotation around an axis
- Four values <x,y,z,w>
- <x,y,z> is axis vector times sin(angle/2)
- w is cos(angle/2)
- No singularities
  - But has dual coverage: Q same rotation as –Q
  - This is useful in some cases!
- Interpolation is fast

Animation vs Deformation

- Skeleton + bone transforms = “pose”
- Animation changes pose over time
  - Knows nothing about vertices and meshes
  - Done by “animation” system on CPU
- Deformation takes a pose, distorts the mesh for rendering
  - Knows nothing about change over time
  - Done by “rendering” system, often on GPU

Model

- Describes a single type of object
- Skeleton + rig
- One per object type
- Referenced by instances in a scene
- Usually also includes rendering data
  - Mesh, textures, materials, etc
  - Physics collision hulls, gameplay data, etc

Instance

- A single entity in the game world
- References a model
- Holds current position & orientation
  - (and gameplay state – health, ammo, etc)
- Has animations playing on it
  - Stores a list of animation controls

Animation Control

- Links an animation and an instance
  - 1 control = 1 anim playing on 1 instance
- Holds current data of animation
  - Current time
  - Speed
  - Weight
  - Masks
  - Looping state

Animation Storage

- The Problem
- Decomposition
- Keyframes and Linear Interpolation
- Higher-Order Interpolation
- The Bezier Curve
- Non-Uniform Curves
- Looping

Storage – The Problem

- 4x3 matrices, 60 per second is huge
  - 200 bone character = 0.5Mb/sec
- Consoles have around 32-64Mb
- Animation system gets maybe 25%
- PC has more memory
  - But also higher quality requirements

Decomposition

- Decompose 4x3 into components
  - Translation (3 values)
  - Rotation (4 values - quaternion)
  - Scale (3 values)
  - Skew (3 values)
- Most bones never scale & shear
- Many only have constant translation
- Don’t store constant values every frame

Keyframes

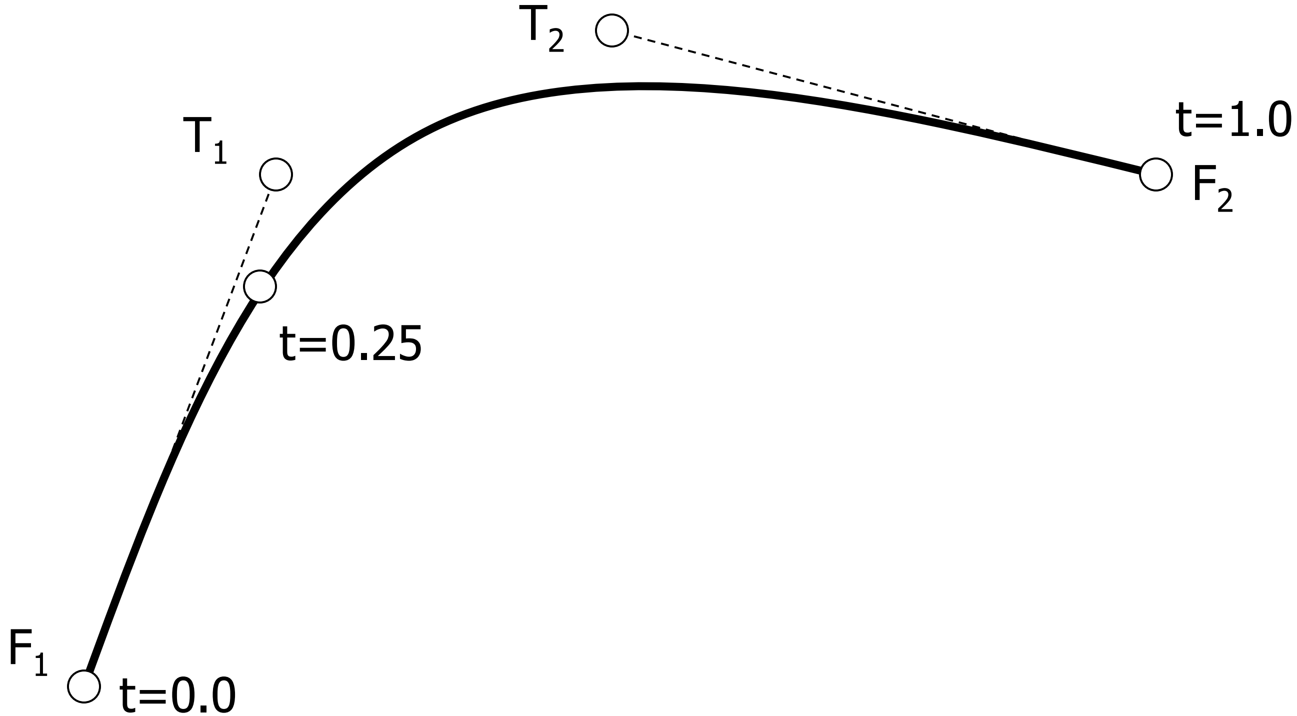
- Motion is usually smooth
- Only store every n^{th} frame
  - Store only “key frames”
- Linearly interpolate between keyframes
  - Inbetweening or “tweening”
- Different anims require different rates
  - Sleeping = low, running = high
  - Choose rate carefully

Higher-Order Interpolation

- Tweening uses linear interpolation
- Natural motions are not very linear
  - Need lots of segments to approximate well
  - So lots of keyframes
- Use a smooth curve to approximate
  - Fewer segments for good approximation
  - Fewer control points
- Bézier curve is very simple curve

The Bézier Curve

- $(1-t)^3F_1+3t(1-t)^2T_1+3t^2(1-t)T_2+t^3F_2$



The Bézier Curve (2)

- Quick to calculate
- Precise control over end tangents
- Smooth
  - C0 and C1 continuity are easy to achieve
  - C2 also possible, but not required here
- Requires three control points per curve
  - (assume F2 is F1 of next segment)
- Far fewer segments than linear

Bézier Variants

- Store $2F_2$-T_2 instead of T_2
  - Equals next segment T_1 for smooth curves
- Store F_1-T_1 and T_2-F_2 vectors instead
  - Same trick as above – reduces data stored
  - Called a “Hermite” curve
- Catmull-Rom curve
  - Passes through all control points

Non-Uniform Curves

- Each segment stores a start time as well
- Time + control value(s) = “knot”
- Segments can be different durations
- Knots can be placed only where needed
  - Allows perfect discontinuities
  - Fewer knots in smooth parts of animation
- Add knots to guarantee curve values
  - Transition points between animations
  - “Golden poses”

Looping and Continuity

- Ensure C0 and C1 for smooth motion
- At loop points
- At transition points
  - Walk cycle to run cycle
- C1 requires both animations are playing at the same speed
  - Reasonable requirement for anim system

Playing Animations

- "Global time" is game-time
- Animation is stored in "local time"
  - Animation starts at local time zero
- Speed is the ratio between the two
  - Make sure animation system can change speed without changing current local time
- Usually stored in seconds
  - Or can be in "frames" - 12, 24, 30, 60 per second

Scrubbing

- Sample an animation at any local time
- Important ability for games
  - Footstep planting
  - Motion prediction
  - AI action planning
  - Starting a synchronized animation
    - Walk to run transitions at any time
- Avoid delta-compression storage methods
  - Very hard to scrub or play at variable speed

Blending Animations

- The Lerp
- Quaternion Blending Methods
- Multi-way Blending
- Bone Masks
- The Masked Lerp
- Hierarchical Blending

The Lerp

- Foundation of all blending
- “Lerp”=**L**inear int**erp**olation
- Blends A, B together by a scalar weight
  - lerp (A, B, i) = iA + (1-i)B
  - i is blend weight and usually goes from 0 to 1
- Translation, scale, shear lerp are obvious
  - Componentwise lerp
- Rotations are trickier

Quaternion Blending

- Normalizing lerp (nlerp)
  - Lerp each component
  - Normalize (can often be approximated)
  - Follows shortest path
  - Not constant velocity
  - Multi-way-lerp is easy to do
  - Very simple and fast

Quaternion Blending (2)

- Spherical lerp (slerp)
  - Usual textbook method
  - Follows shortest path
  - Constant velocity
  - Multi-way-lerp is not obvious
  - Moderate cost

Quaternion Blending (3)

- Log-quaternion lerp (exp map)
  - Rather obscure method
  - Does not follow shortest path
  - Constant velocity
  - Multi-way-lerp is easy to do
  - Expensive

Quaternion Blending (4)

- No perfect solution!
- Each missing one of the features
- All look identical for small interpolations
  - This is the 99% case
  - Blending very different animations looks bad whichever method you use
- Multi-way lerping is important
- So use cheapest - nlerp

Multi-way Blending

- Can use nested lerps
  - lerp (lerp (A, B, i), C, j)
  - But n-1 weights - counterintuitive
  - Order-dependent
- Weighted sum associates nicely
  - (iA + jB + kC + …) / (i + j + k + …)
  - But no i value can result in 100% A
- More complex methods
  - Less predictable and intuitive
  - Can be expensive

Bone Masks

- Some animations only affect some bones
  - Wave animation only affects arm
  - Walk affects legs strongly, arms weakly
    - Arms swing unless waving or holding something
- Bone mask stores weight for each bone
  - Multiplied by animation's overall weight
  - Each bone has a different effective weight
  - Each bone must be blended separately
- Bone weights are usually static
  - Overall weight changes as character changes animations

The Masked Lerp

- Two-way lerp using weights from a mask
  - Each bone can be lerped differently
- Mask value of 1 means bone is 100% A
- Mask value of 0 means bone is 100% B
- Solves weighted-sum problem
  - (no weight can give 100% A)
- No simple multi-way equivalent
  - Just a single bone mask, but two animations

Hierarchical Blending

- Combines all styles of blending
- A tree or directed graph of nodes
- Each leaf is an animation
- Each node is a style of blend
  - Blends results of child nodes
- Construct programmatically at load time
  - Evaluate with identical code each frame
  - Avoids object-specific blending code
  - Nodes with weights of zero not evaluated

Motion Extraction

- Moving the Game Instance
- Linear Motion Extraction
- Composite Motion Extraction
- Variable Delta Extraction
- The Synthetic Root Bone
- Animation Without Rendering

Moving the Game Instance

- Game instance is where the game thinks the object (character) is
- Usually just
  - pos, orientation and bounding box
- Used for everything except rendering
  - Collision detection
  - Movement
  - It's what the game is!
- Must move according to animations

Linear Motion Extraction

- Find position on last frame of animation
- Subtract position on first frame of animation
- Divide by duration
- Subtract this motion from animation frames
- During animation playback, add this delta velocity to instance position
- Animation is preserved and instance moves
- Do same for orientation

Linear Motion Extraction (2)

- Only approximates straight-line motion
- Position in middle of animation is wrong
  - Midpoint of a jump is still on the ground!
- What if animation is interrupted?
  - Instance will be in the wrong place
- Incorrect collision detection
  - Purpose of a jump is to jump *over* things!

Composite Motion Extraction

- Approximates motion with circular arc
- Pre-processing algorithm finds:
  - Axis of rotation (vector)
  - Speed of rotation (radians/sec)
  - Linear speed along arc (metres/sec)
  - Speed along axis of rotation (metres/sec)
    - e.g. walking up a spiral staircase

Composite Motion Extraction (2)

- Very cheap to evaluate
- Low storage costs
- Approximates a lot of motions well
- Still too simple for some motions
  - Mantling ledges
  - Complex acrobatics
  - Bouncing

Variable Delta Extraction

- Uses root bone motion directly
- Sample root bone motion each frame
- Find delta from last frame
- Apply to instance pos+orn
- Root bone is ignored when rendering
  - Instance pos+orn is the root bone

Variable Delta Extraction (2)

- Requires sampling the root bone
- More expensive than CME
  - Can be significant with large worlds
  - Use only if necessary, otherwise use CME
- Complete control over instance motion
- Uses existing animation code and data
  - No “extraction” needed

The Synthetic Root Bone

- All three methods use the root bone
- But what is the root bone?
- Where the character "thinks" they are
  - Defined by animators and coders
- Does not match any physical bone
  - Can be animated completely independently
- Therefore, "synthetic root bone" or SRB

The Synthetic Root Bone (2)

- Acts as point of reference
- SRB is kept fixed between animations
  - During transitions
  - While blending
- Often at centre-of-mass at ground level
  - Called the “ground shadow”
  - But tricky when jumping or climbing – no ground!
- Or at pelvis level
  - Does not rotate during walking, unlike real pelvis
- Or anywhere else that is convenient

Animation Without Rendering

- Not all objects in the world are visible
- But all must move according to anims
- Make sure motion extraction and replay is independent of rendering
- Must run on all objects at all times
  - Needs to be cheap!
  - Use LME & CME when possible
  - VDA when needed for complex animations

Mesh Deformation

- Find Bones in World Space
- Find Delta from Rest Pose
- Deform Vertex Positions
- Deform Vertex Normals

Find Bones in World Space

- Animation generates a “local pose”
  - Hierarchy of bones
  - Each relative to immediate parent
- Start at root
- Transform each bone by parent bone’s world-space transform
- Descend tree recursively
- Now all bones have transforms in world space
  - “World pose”

Find Delta from Rest Pose

- Mesh is created in a pose
  - Often the “da Vinci man” pose for humans
  - Called the “rest pose”
- Must un-transform by that pose first
- Then transform by new pose
  - Multiply new pose transforms by inverse of rest pose transforms
  - Inverse of rest pose calculated at mesh load time
- Gives “delta” transform for each bone

Deform Vertex Positions

- Deformation usually performed on GPU
- Delta transforms fed to GPU
  - Usually stored in “constant” space
- Vertices each have *n* bones
- *n* is usually 4
  - 4 bone indices
  - 4 bone weights 0-1
  - Weights must sum to 1

Deform Vertex Positions (2)

```
vec3 FinalPosition = {0,0,0};
for ( i = 0; i < 4; i++ )
{
  int BoneIndex = Vertex.Index[i];
  float BoneWeight = Vertex.Weight[i];
  FinalPosition +=
    BoneWeight * Vertex.Position *
    PoseDelta[BoneIndex]);
}
```

Deform Vertex Normals

- Normals are done similarly to positions
- But use inverse transpose of delta transforms
  - Translations are ignored
  - For pure rotations, inverse(A)=transpose(A)
  - So inverse(transpose(A)) = A
  - For scale or shear, they are different
- Normals can use fewer bones per vertex
  - Just one or two is common

Inverse Kinematics

- FK & IK
- Single Bone IK
- Multi-Bone IK
- Cyclic Coordinate Descent
- Two-Bone IK
- IK by Interpolation

FK & IK

- Most animation is “forward kinematics”
  - Motion moves *down* skeletal hierarchy
- But there are feedback mechanisms
  - Eyes track a fixed object while body moves
  - Foot stays still on ground while walking
  - Hand picks up cup from table
- This is “inverse kinematics”
  - Motion moves back *up* skeletal hierarchy

Single Bone IK

- Orient a bone in given direction
  - Eyeballs
  - Cameras
- Find desired aim vector
- Find current aim vector
- Find rotation from one to the other
  - Cross-product gives axis
  - Dot-product gives angle
- Transform object by that rotation

Multi-Bone IK

- One bone must get to a target position
  - Bone is called the "end effector"
- Can move some or all of its parents
- May be told which it should move first
  - Move elbow before moving shoulders
- May be given joint constraints
  - Cannot bend elbow backwards

Cyclic Coordinate Descent

- Simple type of multi-bone IK
- Iterative
  - Can be slow
- May not find best solution
  - May not find any solution in complex cases
- But it is simple and versatile
  - No precalculation or preprocessing needed

Cyclic Coordinate Descent (2)

- Start at end effector
- Go up skeleton to next joint
- Move (usually rotate) joint to minimize distance between end effector and target
- Continue up skeleton one joint at a time
- If at root bone, start at end effector again
- Stop when end effector is “close enough”
- Or hit iteration count limit

Cyclic Coordinate Descent (3)

- May take a lot of iterations
- Especially when joints are nearly straight and solution needs them bent
  - e.g. a walking leg bending to go up a step
  - 50 iterations is not uncommon!
- May not find the "right" answer
  - Knee can try to bend in strange directions

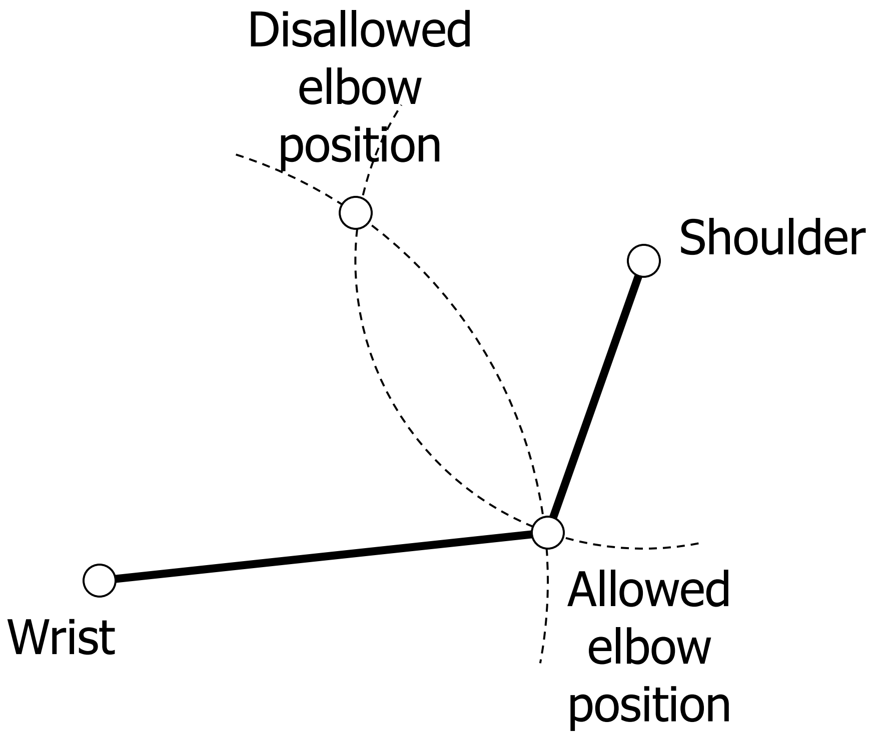
Two-Bone IK

- Direct method, not iterative
- Always finds correct solution
  - If one exists
- Allows simple constraints
  - Knees, elbows
- Restricted to two rigid bones with a rotation joint between them
  - Knees, elbows!
- Can be used in a cyclic coordinate descent

Two-Bone IK (2)

- Three joints must stay in user-specified plane
  - e.g. knee may not move sideways
- Reduces 3D problem to a 2D one
- Both bones must remain same length
- Therefore, middle joint is at intersection of two circles
- Pick nearest solution to current pose
- Or one solution is disallowed
  - Knees or elbows cannot bend backwards

Two-Bone IK (3)



Disallowed elbow position
Shoulder
Wrist
Allowed elbow position

IK by Interpolation

- Animator supplies multiple poses
- Each pose has a reference direction
  - e.g. direction of aim of gun
- Game has a direction to aim in
- Blend poses together to achieve it
- Source poses can be realistic
  - As long as interpolation makes sense
  - Result looks far better than algorithmic IK with simple joint limits

IK by Interpolation (2)

- Result aim point is inexact
  - Blending two poses on complex skeletons does not give linear blend result
- Can iterate towards correct aim
- Can tweak aim with algorithmic IK
  - But then need to fix up hands, eyes, head
  - Can get rifle moving through body

Attachments

- e.g. character holding a gun
- Gun is a separate mesh
- Attachment is bone in character's skeleton
  - Represents root bone of gun
- Animate character
- Transform attachment bone to world space
- Move gun mesh to that pos+orn

Attachments (2)

- e.g. person is hanging off bridge
- Attachment point is a bone in hand
  - As with the gun example
- But here the person moves, not the bridge
- Find delta from root bone to attachment bone
- Find world transform of grip point on bridge
- Multiply by inverse of delta
  - Finds position of root to keep hand gripping

Collision Detection

- Most games just use bounding volume
- Some need perfect triangle collision
  - Slow to test every triangle every frame
- Precalculate bounding box of each bone
  - Transform by world pose transform
  - Finds world-space bounding box
- Test to see if bbox was hit
  - If it did, test the tris this bone influences

Conclusions

- Use quaternions
  - Matrices are too big, Eulers are too evil
- Memory use for animations is huge
  - Use non-uniform spline curves
- Ability to scrub anims is important
- Multiple blending techniques
  - Different methods for different places
  - Blend graph simplifies code

Conclusions (2)

- Motion extraction is tricky but essential
  - Always running on all instances in world
  - Trade off between cheap & accurate
  - Use Synthetic Root Bone for precise control
- Deformation is really part of rendering
  - Use graphics hardware where possible
- IK is much more than just IK algorithms
  - Interaction between algorithms is key