CS 4204 Computer Graphics

Computer Animation
Adapted from notes by Yong Cao
Virginia Tech
Outline

- Principles of Animation
- Keyframe Animation
- Additional challenges in animation
Classic animation – Luxo Jr. (1986)

http://www.youtube.com/watch?v=vsKdRmqqQI
### Principles of Traditional Animation – Disney

- Squash and Stretch
- Slow In and Out
- Anticipation
- Exaggeration
- Follow Through and Overlapping Action
- Timing
- Staging
- Straight Ahead Action and Pose-to-Pose Action
- Arcs
- Secondary Action
- Appeal

[Link to Principles of Traditional Animation](http://www.siggraph.org/education/materials/HyperGraph/animation/character_animation/principles/prin_trad_anim.htm)
Slow In and Out
Exaggeration
Timing and Follow through
Secondary actions
Keyframe Animation

Define Character Poses at Specific Time Steps Called “Keyframes”
Keyframe Animation

Interpolate Variables Describing Keyframes to Determine Poses for Character in between
In-betweening

Linear Interpolation

Usually not enough continuity

Linear interpolation
In-betweening

Spline Interpolation

*Maybe good enough*
In-betweening

Cubic Spline Interpolation

Maybe good enough

May not follow physical laws !!
In-betweening

Cubic Spline Interpolation

Maybe good enough

May not follow physical laws!!
Articulated Figures

Character Poses Described by Set of Rigid Bodies Connected by “Joints”

Base
Arm
Hand
Scene Graph
Articulated Figures

Well-Suited for Humanoid Characters
Articulated Figures

Joints Provide Handles for Moving Articulated Figure
In-betweening

Compute Joint Angles between Keyframes
Example: Walk Cycle

Articulated Figure:

- Hip
- Upper Leg
  - Knee
  - Lower Leg
    - Ankle
    - Foot
- Upper Leg (Hip Rotate)
  - Hip Rotate
- Lower Leg (Knee Rotate)
  - Hip Rotate + Knee Rotate
- Foot (Ankle Rotate)
Example: Walk Cycle

Hip Joint Orientation:
Example: Walk Cycle

*Knee Joint Orientation:*

![Diagram showing knee joint orientation with angles and phases labeled 1 through 5.](image)
Example: Walk Cycle

Ankle Joint Orientation:
When we speak of an ‘animation’, we refer to the data required to pose a skeleton over some range of time.

This should include information to specify all necessary DOF values over the entire time range.

Sometimes, this is referred to as a ‘clip’ or even a ‘move’ (as ‘animation’ can be ambiguous).
- If a character has $N$ DOFs, then a pose can be thought of as a point in $N$-dimensional pose space

- An animation can be thought of as a point moving through pose space, or alternately as a fixed curve in pose space

- ‘One-shot’ animations are an open curve, while ‘loop’ animations form a closed loop

- Generally, we think of an individual ‘animation’ as being a continuous curve, but there’s no strict reason why we couldn’t have discontinuities (cuts)
Channels

- If the entire animation is an N-dimensional curve in pose space, we can separate that into N 1-dimensional curves, one for each DOF.

- We call these ‘channels’.

- A channel stores the value of a scalar function over some 1D domain (either finite or infinite).

- A channel will refer to pre-recorded or pre-animated data for a DOF, and does not refer to the more general case of a DOF changing over time (which includes physics, procedural animation...).
Channels
An animation can be stored as an array of channels.

A simple means of storing a channel is as an array of regularly spaced samples in time.

Using this idea, one can store an animation as a 2D array of floats (NumDOFs x NumFrames).

However, if one wanted to use some other means of storing a channel, they could still store an animation as an array of channels, where each channel is responsible for storing data however it wants.
Array of Poses

An alternative way to store an animation is as an array of poses

This also forms a 2D array of floats

\((\text{NumFrames} \times \text{NumDOFs})\)

Which is better, poses or channels?
Poses vs. Channels

Which is better?

It depends on your requirements.

The bottom line:

• Poses are faster
• Channels are far more flexible and can potentially use less memory
Array of Poses

- The array of poses method is about the fastest possible way to playback animation data
- A 'pose' (vector of floats) is exactly what one needs in order to pose a rig
- Data is contiguous in memory, and can all be directly accessed from one address
Array of Channels

- As each channel is stored independently, they have the flexibility to take advantage of different storage options and maximize memory efficiency.
- Also, in an interactive editing situation, new channels can be independently created and manipulated.
- However, they need to be independently evaluated to access the ‘current frame’, which takes time and implies discontinuous memory access.
Poses vs. Channels

- Array of poses is great if you just need to play back some relatively simple animation and you need maximum performance. This corresponds to many video games.
- Array of channels is essential if you want flexibility for an animation system or are interested in generality over raw performance.
- Array of channels can also be useful in more sophisticated game situations or in cases where memory is more critical than CPU performance (which is not uncommon).
Keyframe Channel

- A channel can be stored as a sequence of keyframes
- Each keyframe has a time and a value and usually some information describing the tangents at that location
- The curves of the individual spans between the keys are defined by 1-D interpolation (usually piecewise Hermite)

```cpp
class Keyframe;
class Channel {
    float *m_keyframe_array;
    float Evaluate(float time);
    bool Load(FILE*);
};
```
Keyframe Channel
Keyframe

Value

Tangent In

Keyframe (time,value)

Tangent Out

Time
Why Use Keyframes?

- Good user interface for adjusting curves
- Gives the user control over the value of the DOF and the velocity of the DOF
- Define a perfectly smooth function (if desired)
- Can offer good compression (not always)

- Every animation system offers some variation on keyframing
- Video games may consider keyframes for compression purposes, even though they have a performance cost
Tangent Rules

- Rather than store explicit numbers for tangents, it is often more convenient to store a 'rule' and recompute the actual tangent as necessary.

- Usually, separate rules are stored for the incoming and outgoing tangents.

- Common rules for Hermite tangents include:
  - Flat (tangent = 0)
  - Linear (tangent points to next/last key)
  - Smooth (automatically adjust tangent for smooth results)
  - Fixed (user can arbitrarily specify a value)

- Remember that the tangent equals the rate of change of the DOF (or the velocity).
Flat Tangents

*Flat tangents are particularly useful for making ‘slow in’ and ‘slow out’ motions (acceleration from a stop and deceleration to a stop)*
Linear Tangents

\[ q_{\text{out}} - q_{\text{in}} = \frac{q_0 - q_1}{t_0 - t_1} \]

\((p_0, t_0)\)

\((p_1, t_1)\)

\(v_{0\text{ out}}\)

\(v_{1\text{ in}}\)
Smooth Tangents

Keep in mind that this won’t work on the first or last tangent (just use the linear rule)
Cubic Coefficients

- Keyframes are stored in order of their time
- Between every two successive keyframes is a span of a cubic curve
- The span is defined by the value of the two keyframes and the outgoing tangent of the first and incoming tangent of the second
- Those 4 values are multiplied by the Hermite basis matrix and converted to cubic coefficients for the span
- For simplicity, the coefficients can be stored in the first keyframe for each span
Extrapolation Modes

- Channels can specify ‘extrapolation modes’ to define how the curve is extrapolated before \( t_{\text{min}} \) and after \( t_{\text{max}} \)
- Usually, separate extrapolation modes can be set for before and after the actual data
- Common choices:
  - Constant value (hold first/last key value)
  - Linear (use tangent at first/last key)
  - Cyclic (repeat the entire channel)
  - Cyclic Offset (repeat with value offset)
  - Bounce (repeat alternating backwards & forwards)
Extrapolation

Flat:

Linear:

$t_{\text{min}}$

$t_{\text{max}}$
Extrapolation

**Cyclic:**

**Cyclic Offset:**
Extrapolation

_Bounce:_
The main runtime function for a channel is something like:

```cpp
float Channel::Evaluate(float time);
```

This function will be called many times...

For an input time \( t \), there are 4 cases to consider:

- \( t \) is before the first key (use extrapolation)
- \( t \) is after the last key (use extrapolation)
- \( t \) falls exactly on some key (return key value)
- \( t \) falls between two keys (evaluate cubic equation)
The Channel::Evaluate function needs to be very efficient, as it is called many times while playing back animations.

There are two main components to the evaluation:

- Find the proper span
- Evaluate the cubic equation for the span
Random Access

- To evaluate a channel at some arbitrary time $t$, we need to first find the proper span of the channel and then evaluate its equation.
- As the keyframes are irregularly spaced, this means we have to search for the right one.
- If the keyframes are stored as a linked list, there is little we can do except walk through the list looking for the right span.
- If they are stored in an array, we can use a binary search, which should do reasonably well.
Higher-level animation issues

**Using motion capture data for animation** *(video)*

**Sequencing, blending, or adding multiple animations** *(video)*

**Generating animations**
- Procedurally *(video)*
- Based on physical simulation *(video)*

**Animation of phenomena that don’t fit the articulated body model (smoke, water, explosions, hair, cloth, etc.)** *(video)*