



# Channels & Keyframes

---

CSE169: Computer Animation

Instructor: Steve Rotenberg

UCSD, Winter 2020



# Animation

---

# Rigging and Animation

Animation  
System

Pose

Rigging  
System

Triangles

Renderer

$$\Phi = [\phi_1 \quad \phi_2 \quad \dots \quad \phi_N]$$

# Animation

---

- When we speak of an 'animation', we refer to the data required to pose a rig 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)
-

# Pose Space

- If a character has  $N$  DOFs, then a pose can be thought of as a point in  $N$ -dimensional pose space

$$\Phi = [\phi_1 \quad \phi_2 \quad \dots \quad \phi_N]$$

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

$$\Phi = \Phi(t)$$

- ‘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)

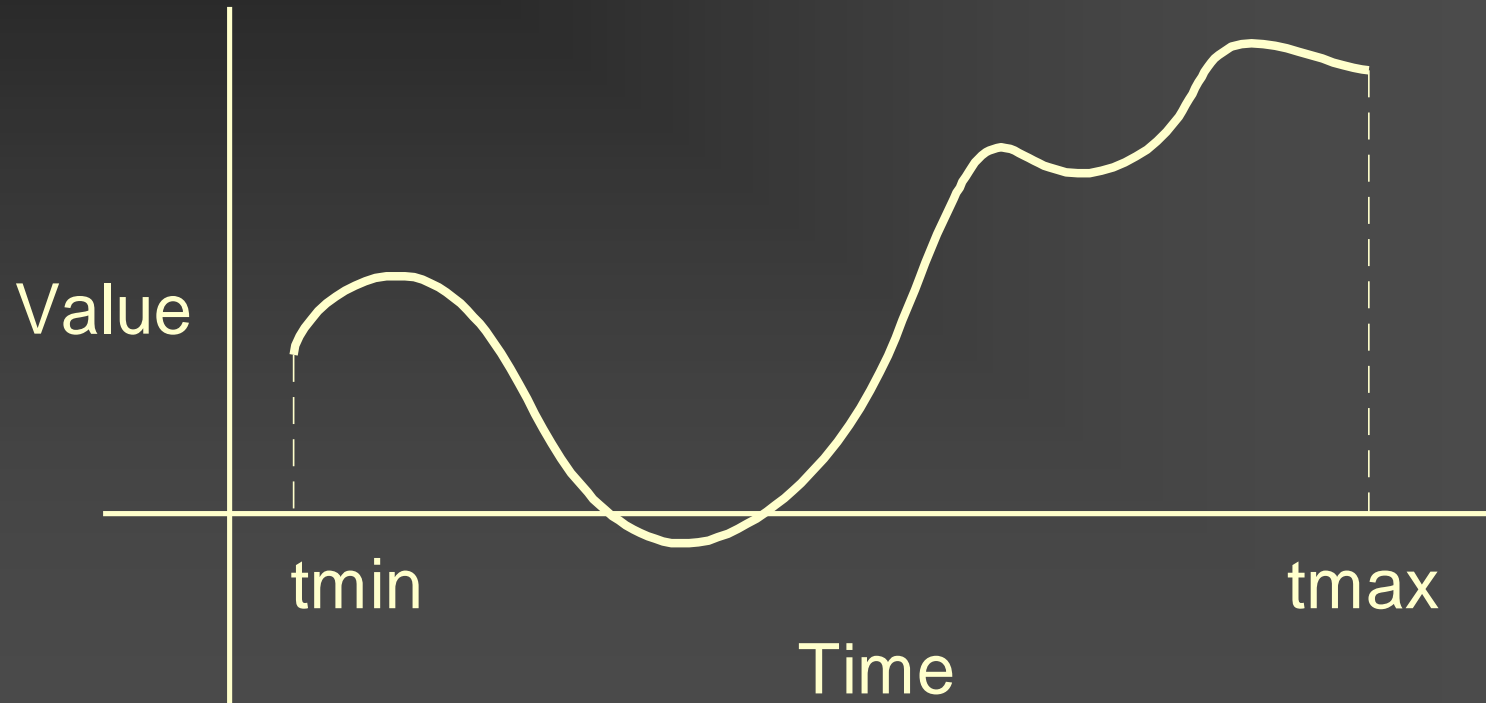
# Channels

- 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

$$\phi_i = \phi_i(t)$$

- 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



# Channels

---

- As a channel represents pre-recorded data, evaluating the channel for a particular value of  $t$  should always return the same result
  - We allow channels to be discontinuous in value, but not in time
  - Most of the time, a channel will be used to represent a DOF changing over time, but occasionally, we will use the same technology to relate some arbitrary variable to some other arbitrary variable (i.e., torque vs. RPM curve of an engine...)
-



# Array of 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 (NumFrames x 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)
-

# Channels

---

- As the array of poses method is very simple, there's not much more to say about it
  - Therefore, we will concentrate on channels on their various storage and manipulation techniques
-

# Temporal Continuity

---

- Sometimes, we think of animations as having a particular frame rate (i.e., 30 fps)
  - It's often a better idea to think of them as being continuous in time and not tied to any particular rate. Some reasons include:
    - Film / NTSC / PAL conversion
    - On-the-fly manipulation (stretching/shrinking in time)
    - Motion blur
  - Certain effects (and fast motions) may require one to be really aware of individual frames though...
-



# Animation Storage

---

- Regardless of whether one thinks of an animation as being continuous or as having discrete points, one must consider methods of storing animation data
  - Some of these methods may require some sort of temporal discretization, while others will not
  - Even when we do store a channel on frame increments, it's still nice to think of it as a continuous function interpolating the time between frames
-

# Animation Class

---

```
class AnimationClip {  
    void Evaluate(float time, Pose &p);  
    bool Load(const char *filename);  
};
```

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

---

# Channel Storage

---

- There are several ways to store channels. Most approaches fall into either storing them in a 'raw' frame method, or as piecewise interpolating curves (keyframes)
  - A third alternative is as a user supplied expression, which is just an arbitrary math function. In practice, this is not too common, but can be handy in some situations.
  - One could also apply various interpolation schemes, but most channel methods are designed more around user interactivity
-

# Raw Data Formats

---

- Sometimes, channels are stored simply as an array of values, regularly spaced in time at some frame rate
  - They can use linear or smoother interpolation to evaluate the curve between sample points
  - The values are generally floats, but could be compressed more if desired
  - The frame rate is usually similar to the final playback frame rate, but could be less if necessary
-

# Compressing Raw Channels

- Rotational data can usually be compressed to 16 bits with reasonable fidelity
- Translations can be compressed similarly if they don't go too far from the origin
- One can also store a float min & max value per channel and store a fixed point value per frame that interpolates between min & max
- Lowering the frame rate will also save a lot of space, but can only be done for smooth animations
- One could use an automatic algorithm to compress each channel individually based on user specified tolerances
- Raw channels can also be stored using some form of *delta compression*



# Keyframe Channels

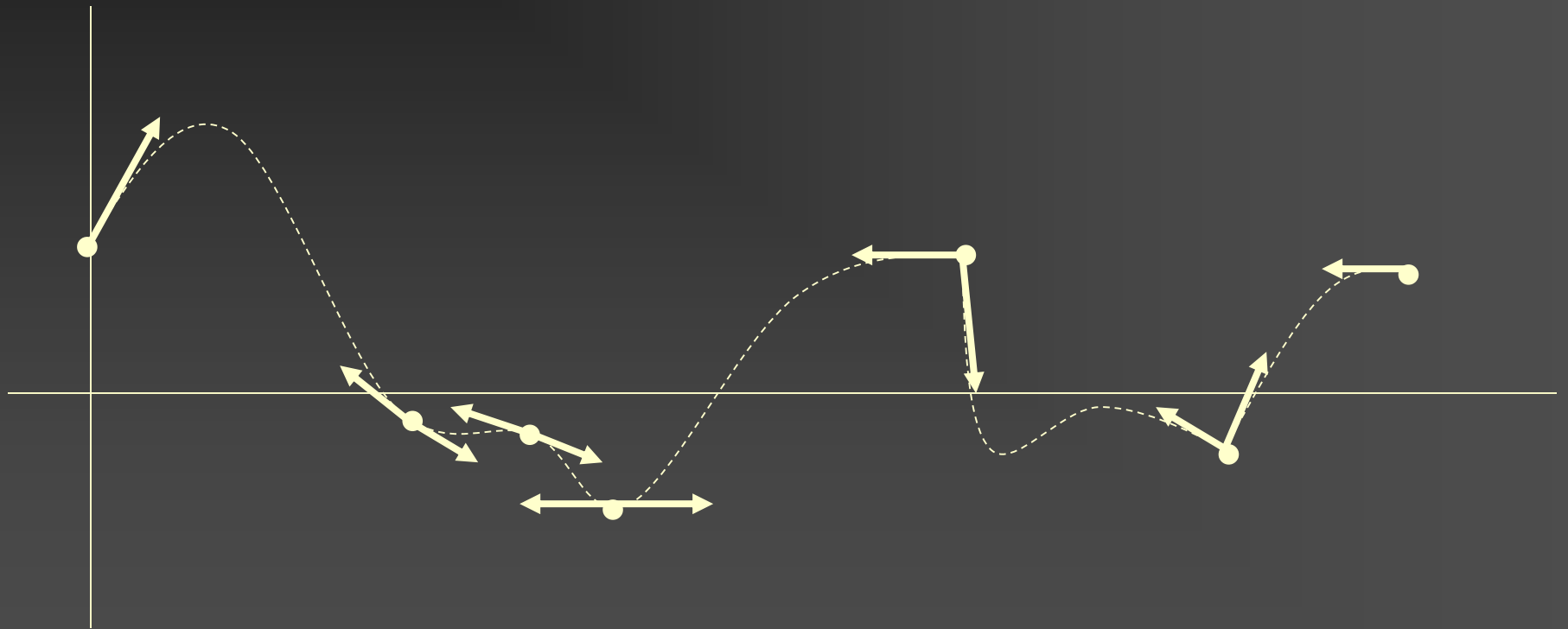
---

# Keyframe Channel

---

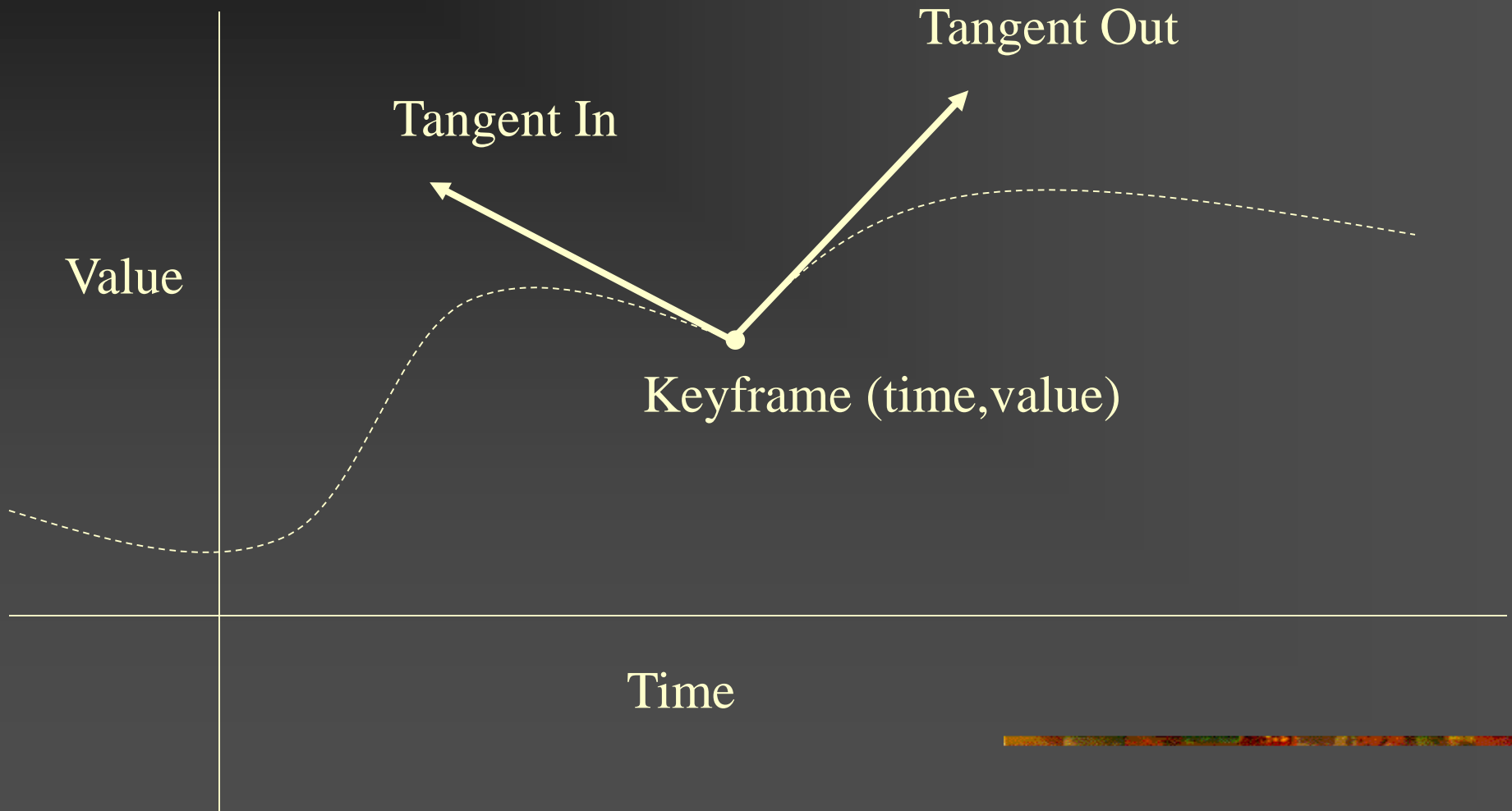
- A channel can be stored as a sequence of keyframes
  - Each keyframe has a time and a value and usually some information describing the tangents at that location
  - The curves of the individual *spans* between the keys are defined by 1-D interpolation (usually piecewise Hermite)
-

# Keyframe Channel





# Keyframe



# Keyframe Tangents

---

- Keyframes are usually drawn so that the incoming tangent points to the left (earlier in time)
  - The arrow drawn is just for visual representation and it should be remembered that if the two arrows are exactly opposite, that actually means the tangents are the same!
  - Also remember that we are only dealing with 1D curves now, so the tangent really just a slope
-

# 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

# Animating with Keyframes

---

- Keyframed channels form the foundation for animating properties (DOFs) in many commercial animation systems
  - Different systems use different variations on the exact math but most are based on some sort of cubic Hermite curves
-

# Curve Fitting

---

- Keyframes can be generated automatically from sampled data such as motion capture
- This process is called 'curve fitting', as it involves finding curves that fit the data reasonably well
- Fitting algorithms allow the user to specify tolerances that define the acceptable quality of the fit
- This allows two way conversion between keyframe and raw formats, although the data might get slightly distorted with each translation

# Keyframe Data Structure

```
class Keyframe {  
    float Time;  
    float Value;  
    float TangentIn, TangentOut;  
    char RuleIn, RuleOut;    // Tangent rules  
    float A, B, C, D;        // Cubic coefficients  
}
```

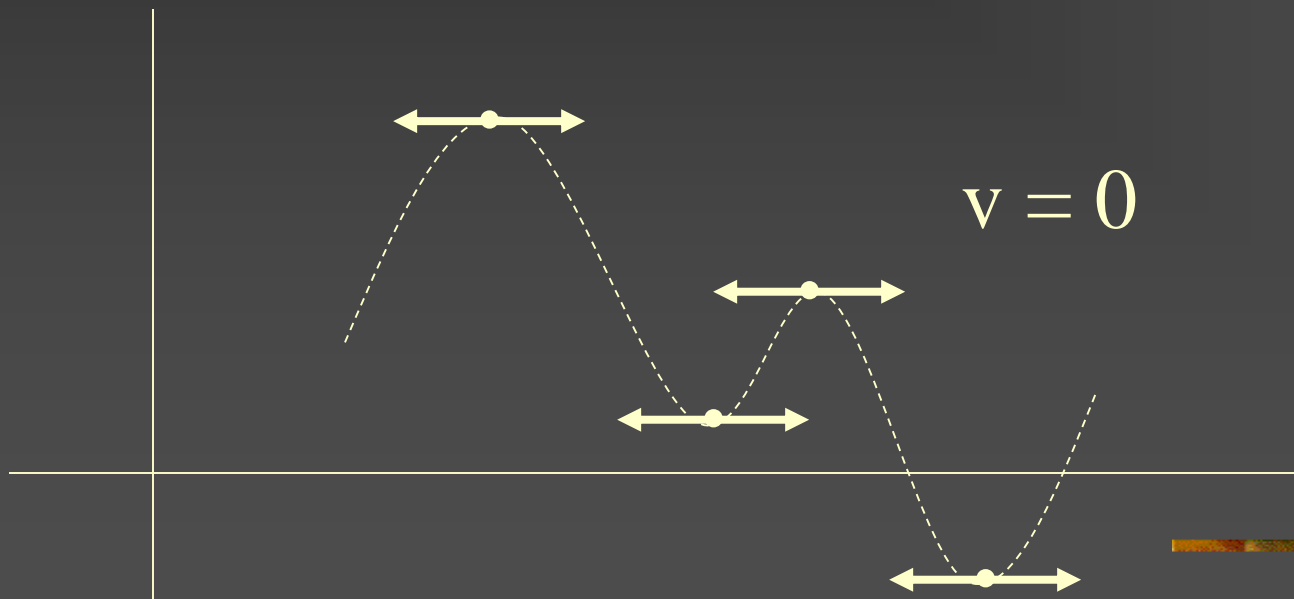
- Data Structures:
  - Linked list
  - Doubly linked list
  - Array

# 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)
- Note: I use ' $v$ ' for tangents (velocity) instead of ' $t$ ' which is used for time

# Flat Tangents

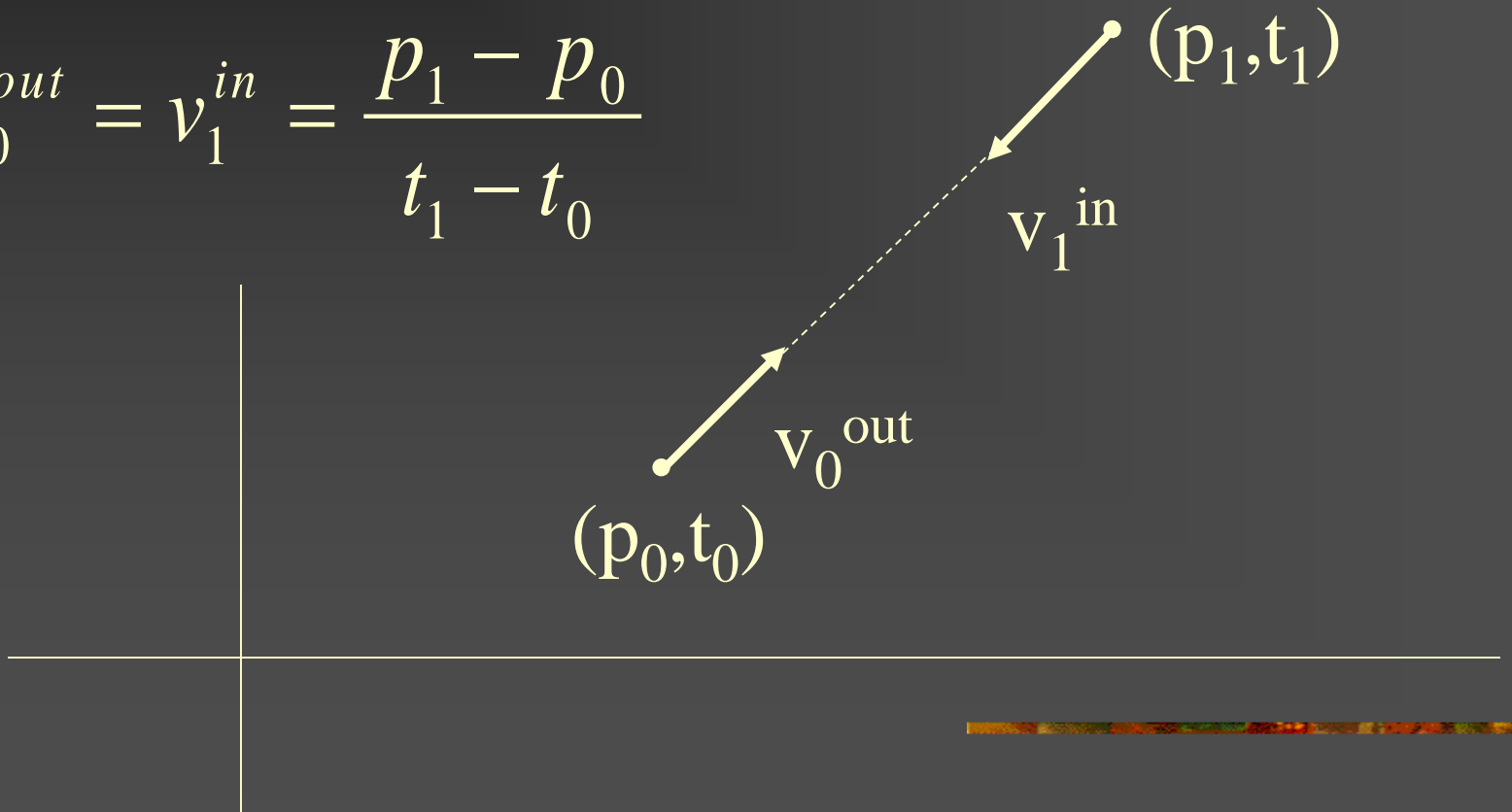
- Flat tangents are particularly useful for making 'slow in' and 'slow out' motions (acceleration from a stop and deceleration to a stop)



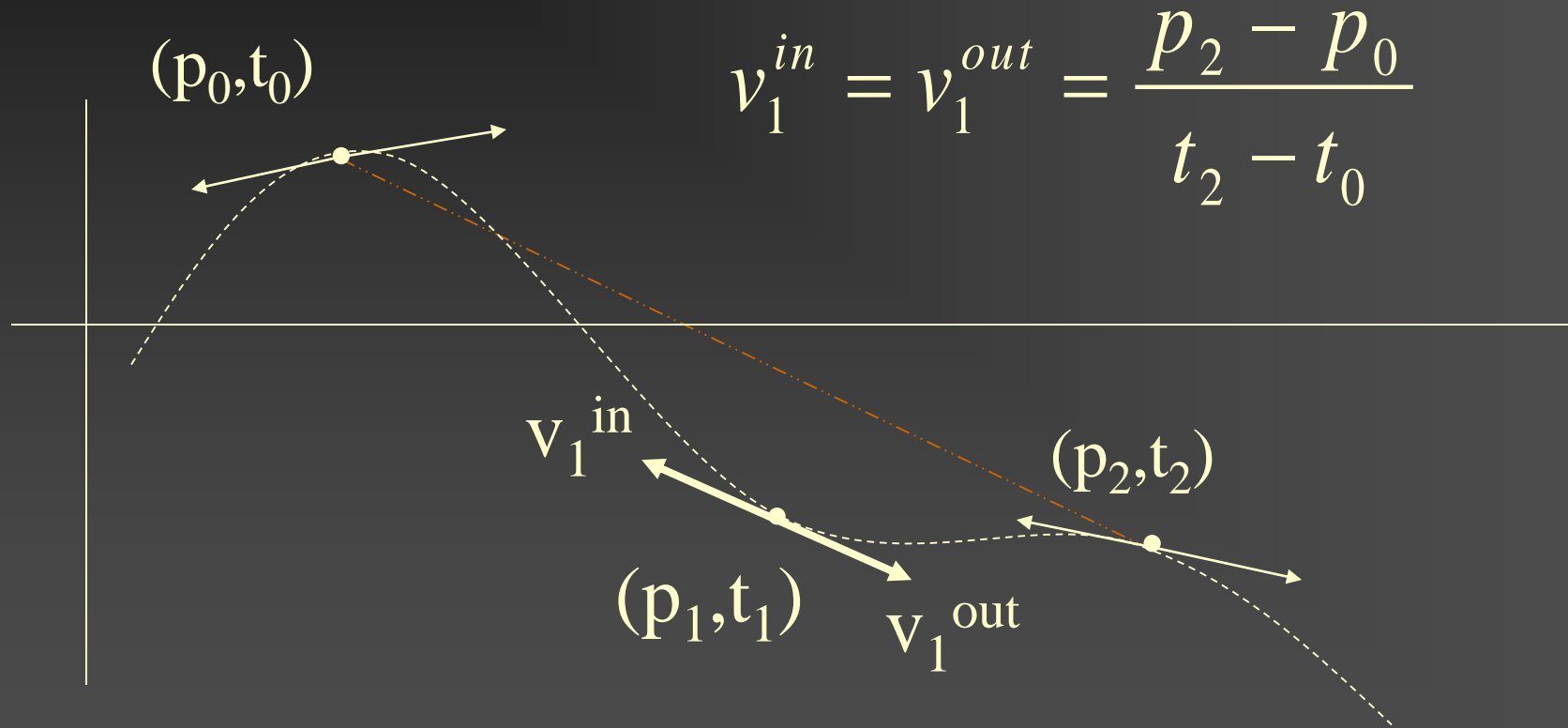


# Linear Tangents

$$v_0^{out} = v_1^{in} = \frac{p_1 - p_0}{t_1 - t_0}$$



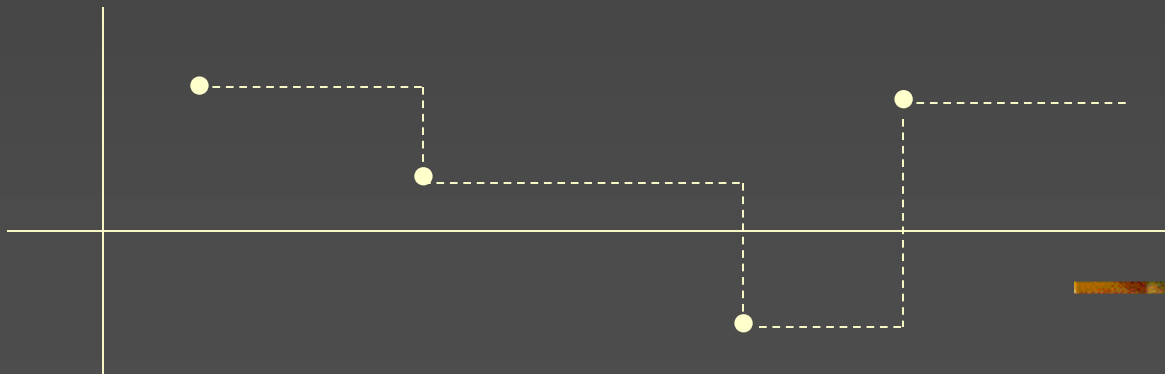
# Smooth Tangents



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

# Step Tangent

- Occasionally, one comes across the 'step' tangent rule
- This is a special case that just forces the entire span to a constant
- This requires hacking the cubic coefficients ( $a=b=c=0$ ,  $d=p_0$ )
- It can only be specified on the outgoing tangent and it nullifies whatever rule is on the next incoming tangent



# 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

# Cubic Equation (1 dimensional)

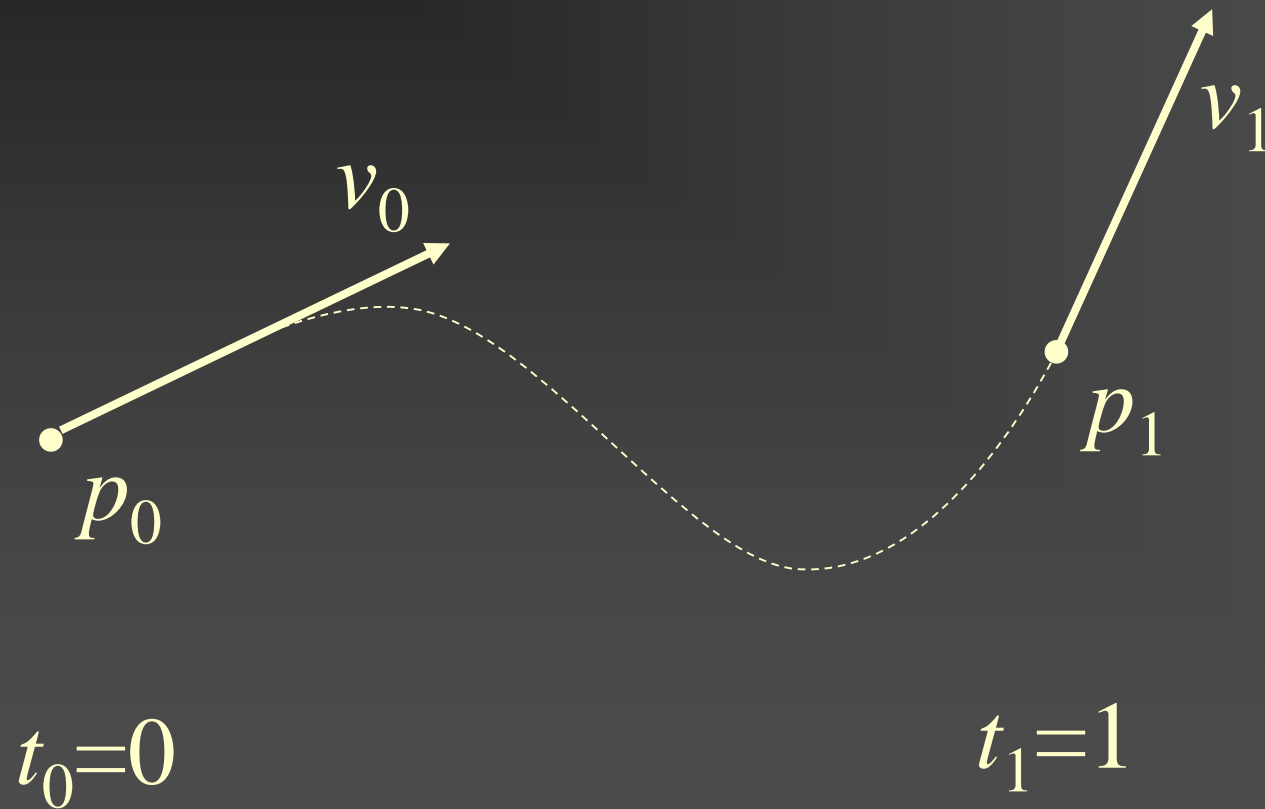
$$f(t) = at^3 + bt^2 + ct + d$$

$$\frac{df}{dt} = 3at^2 + 2bt + c$$

$$f(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$\frac{df}{dt} = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

# Hermite Curve (1D)



# Hermite Curves

- We want the value of the curve at  $t=0$  to be  $f(0)=p_0$ , and at  $t=1$ , we want  $f(1)=p_1$
  - We want the derivative of the curve at  $t=0$  to be  $v_0$ , and  $v_1$  at  $t=1$
- 

$$f(0) = p_0 = a0^3 + b0^2 + c0 + d = d$$

$$f(1) = p_1 = a1^3 + b1^2 + c1 + d = a + b + c + d$$

$$f'(0) = v_0 = 3a0^2 + 2b0 + c = c$$

$$f'(1) = v_1 = 3a1^2 + 2b1 + c = 3a + 2b + c$$

# Hermite Curves

$$p_0 = d$$

$$p_1 = a + b + c + d$$

$$v_0 = c$$

$$v_1 = 3a + 2b + c$$

$$\begin{bmatrix} p_0 \\ p_1 \\ v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$



# Matrix Form of Hermite Curve

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} p_0 \\ p_1 \\ v_0 \\ v_1 \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ v_0 \\ v_1 \end{bmatrix}$$

# Matrix Form of Hermite Curve

$$f(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ v_0 \\ v_1 \end{bmatrix}$$

$$f(t) = \mathbf{t} \cdot \mathbf{B}_{Hrm} \cdot \mathbf{g}_{Hrm}$$

$$f(t) = \mathbf{t} \cdot \mathbf{c}$$

- Remember, this assumes that  $t$  varies from 0 to 1

# Inverse Linear Interpolation

- If  $t_0$  is the time at the first key and  $t_1$  is the time of the second key, a linear interpolation of those times by parameter  $u$  would be:

$$t = Lerp(u, t_0, t_1) = (1 - u)t_0 + ut_1$$

- The inverse of this operation gives us:

$$u = InvLerp(t, t_0, t_1) = \frac{t - t_0}{t_1 - t_0}$$

- This gives us a 0...1 value on the span where we now will evaluate the cubic equation
- Note:  $1/(t_1 - t_0)$  can be precomputed for each span

# Evaluating Cubic Spans

---

- Tangents are generally expressed as a slope of value/time
  - To normalize the spans to the 0...1 range, we need to correct the tangents
  - So we must scale them by  $(t_1 - t_0)$
-

# Precomputing Constants

- For each span we pre-compute the cubic coefficients:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ (t_1 - t_0)v_0 \\ (t_1 - t_0)v_1 \end{bmatrix}$$

# Computing Cubic Coefficients

- Note: My matrix34 class won't do this properly!
- Actually, all of the 1's and 0's in the matrix make it pretty easy to multiply it out by hand

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ (t_1 - t_0)v_0 \\ (t_1 - t_0)v_1 \end{bmatrix}$$

# 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  $u$ )

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

$$x = au^3 + bu^2 + cu + d = d + u(c + u(b + u(a)))$$

# Channel::Precompute()

---

- The two main setup computations a keyframe channel needs to perform are:
    - Compute tangents from rules
    - Compute cubic coefficients from tangents & other data
  - This can be done in two separate passes through the keys or combined into one pass (but keep in mind there is some slightly tricky dependencies on the order that data must be processed if done in one pass)
-



# Extrapolation Modes

- Channels can specify 'extrapolation modes' to define how the curve is extrapolated before  $t_{\min}$  and after  $t_{\max}$
- 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)

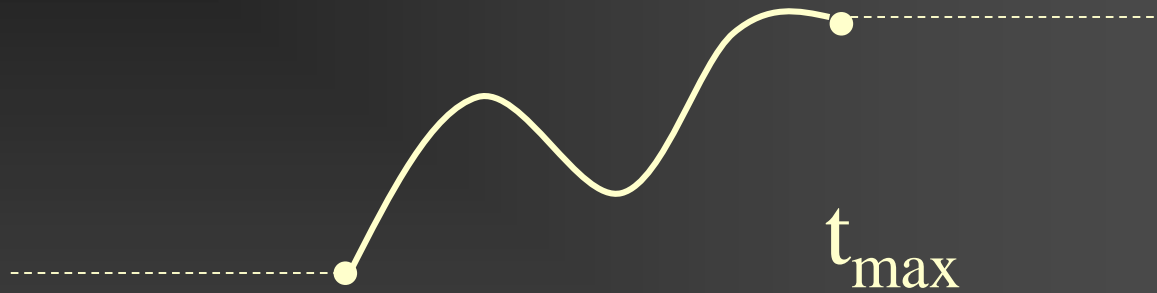
# Extrapolation

---

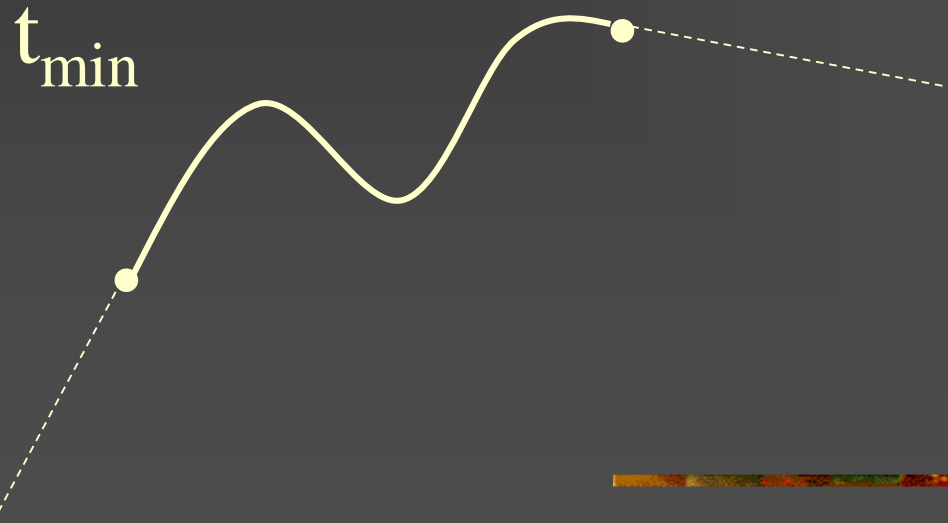
- Note that extrapolation applies to the entire channel and not to individual keys
  - In fact, extrapolation is not directly tied to keyframing and can be used for any method of channel storage (raw...)
-

# Extrapolation

■ Flat:

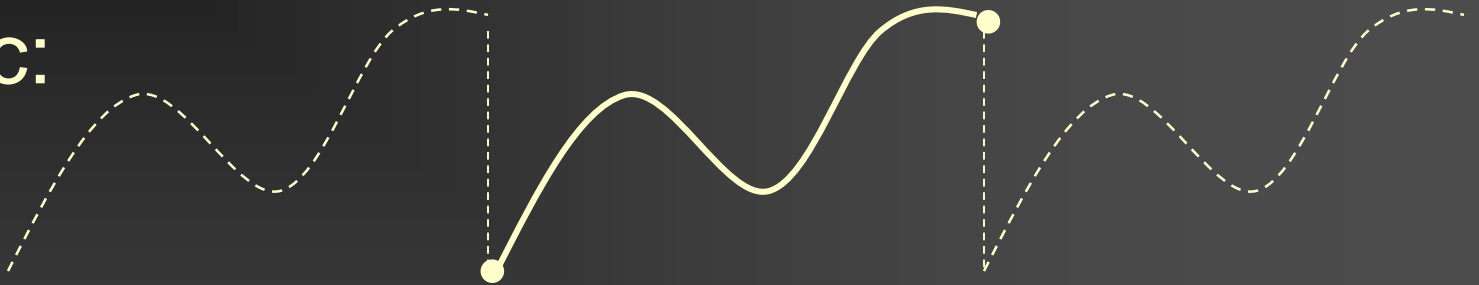


■ Linear:

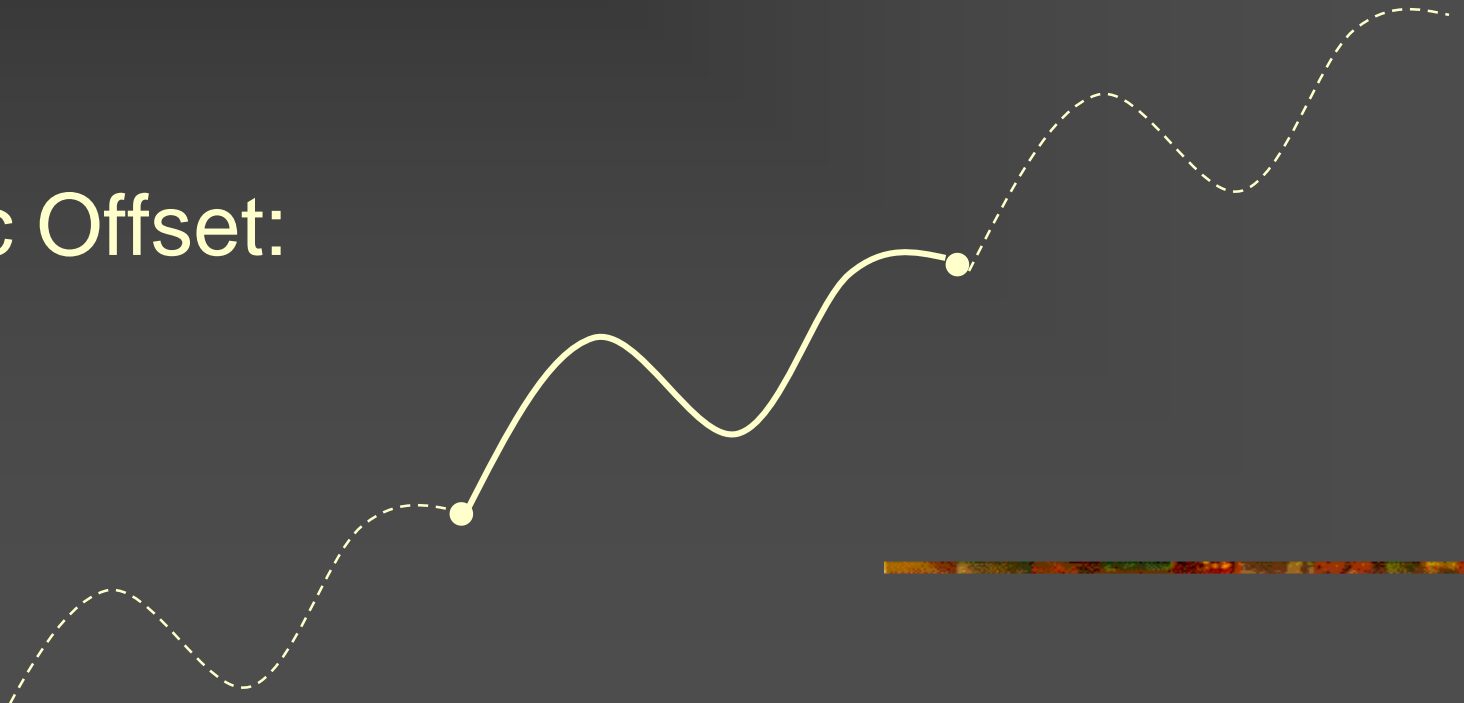


# Extrapolation

■ Cyclic:

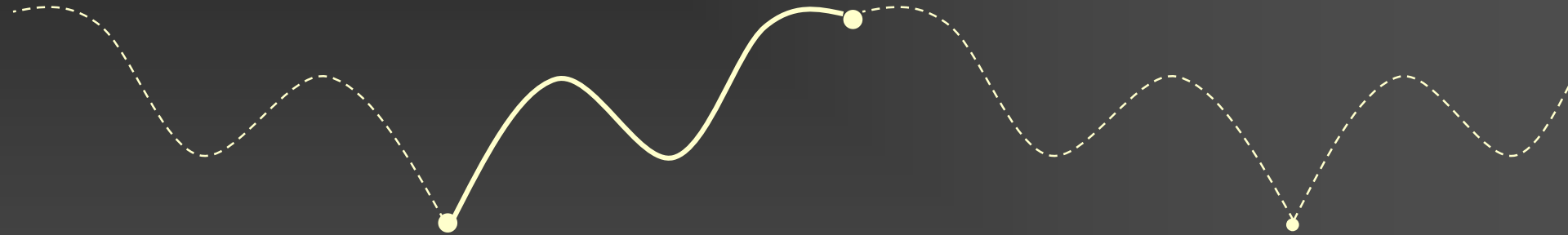


■ Cyclic Offset:



# Extrapolation

- Bounce:



# Keyframe Evaluation

- The main runtime function for a channel is something like:

```
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)

# Channel::Evaluate()

---

- 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
-



# Finding the Span: Binary Search

- A very reasonable way to find the key is by a binary search. This allows pretty fast ( $\log N$ ) access time with no additional storage cost (assuming keys are stored in an array (rather than a list))
- Binary search is sometimes called 'divide and conquer' or 'bisection'
- For even faster access, one could use hashing algorithms, but that is probably not necessary, as they require additional storage and most real channel accesses can take advantage of coherence (sequential access)

# Finding the Span: Linear Search

---

- One can always just loop through the keys from the beginning and look for the proper span
  - This is an acceptable place to start, as it is important to get things working properly before focusing on optimization
  - It may also be a reasonable option for interactive editing tools that would require key frames to be stored in a linked list
  - Of course, a bisection algorithm can probably be written in less than a dozen lines of code...
-

# Sequential Access

---

- If a character is playing back an animation, then it will be accessing the channel data sequentially
  - Doing a binary search for each channel evaluation for each frame is not efficient for this
  - If we keep track of the most recently accessed key for each channel, then it is extremely likely that the next access will require either the same key or the very next one
  - This makes sequential access of keyframes potentially very fast
  - However there is a catch...
-

# Sequential Access

- Consider a case where we have a video game with 20 bad guys running around
- They all need to access the same animation data (which should only be stored once obviously)
- However, they might each be accessing the channels with a different 'time'
- Therefore, the higher level code that plays animations needs to keep track of the most recent keys rather than the simpler solution of just having each channel just store a pointer to its most recent key
- Thus, the animation player class needs to do considerable bookkeeping, as it will need to track a most recent key for every channel in the animation

# High Performance Channels

- If coefficients are stored, we can evaluate the cubic equation with 4 additions and 4 multiplies
- In fact,  $a, b, c,$  &  $d$  can actually be precomputed to include the correction for  $1/(t_1 - t_0)$  so that the cubic can be directly solved for the original  $t$ . This reduces it to  $3+$  and  $3^*$
- In other words, evaluating the cubic is practically instantaneous, while jumping around through memory trying to locate the span is far worse
- If we can take advantage of sequential access (which we usually can), we can reduce the span location to a very small number of operations

# Robustness

---

- The channel should always return some reasonable value regardless of what time  $t$  was passed in
    - If there are no keys in the channel, it should just return 0
    - If there is just 1 key, it should return the value of that key
    - If there are more than 1 key, it should evaluate the curve or use an extrapolation rule if  $t$  is outside of the range
    - At a minimum, the 'constant' extrapolation rule should be used, which just returns the value of the first (or last) key if  $t$  is before (or after) the keyframe range
  - When creating new keys or modifying the time of a key, one needs to verify that its time stays between the key before and after it
-