

# Computer Animation

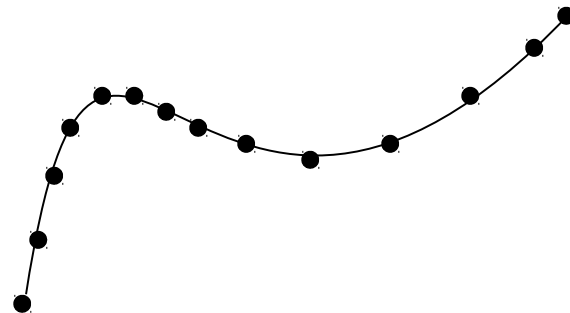
## 4-Motion Control

### SS 16

Prof. Dr. Charles A. Wüthrich,  
Fakultät Medien, Medieninformatik  
Bauhaus-Universität Weimar  
[caw AT medien.uni-weimar.de](mailto:caw AT medien.uni-weimar.de)

# Controlling motion along curves

- We all know now how to control the shape of the curve
- To an animator, it is equally important to know the speed at which a curve is traced by increasing parametric steps
- Obviously, since motion curves are of higher order, this relation is not straightforward
- Equal parameter intervals do not lead to arcs of equal length on the curve
- That is, speed is different at different points of the curve
- This can be overcome through a reparametrization of the curve



# Curve length

- There are different methods to compute such a reparametrization
- One can create a table of values so as to establish a relationship between arc length and parameter values
- In the first two methods, one creates a table of values to establish the relationship between parametric value and approximate arc length
- Once the table is built, one can use the table to approximate values of the parameter at steps of equal length along the curve



# Curve length

- The first method supersamples the curve, and then uses summed linear distance to compute the approximate arc length
- The second method uses Gauss quadrature to numerically estimate the arc length
- Both methods can use adaptive subdivision to control the error
- The third method analytically computes arc length. Unfortunately, it is not always possible to do so for all curves.

# Computing arc length

- To specify how fast the object moves in the environment, animators might want to specify the time at which positions along the curve are reached.
- This in general would be position and frame pairs.
- Or, maybe, the animator might want to specify velocities
- For example:
  - start at position A
  - accelerate till frame 20
  - move at constant speed till frame 35
  - Decelerate slowly till frame 60 and end at position B
- It is clear what we want: be able to control not only the curve (*space function*), but also the relationship between position and time (*distance-time function*).
- The distance we are traveling along the curve is called the *arc length* and will be denoted by  $s$ .

# Computing arc length

- Suppose that we are moving along the curve

$$P(u) = U^T M B$$

- The relation between parameter and arc length is not linear.
- When a unit change in parameter results in a unit change in curve length the curve is said to be *parametrized by arc length*

U

- How do I establish the relationship between parameter and arc length?
- What we want is to know the function  $s = G(u)$  which computes the length of the curve from its starting point for all values of the parameter  $u$
- If we have  $G$ , then knowing  $G^{-1}$  allows us to compute the parameter values corresponding to a certain length

# Arc length: analytic approach

- Obviously, the length of a curve between parameter values  $u_1$  and  $u_2$  is

$$s = \int_{[u_1, u_2]} |dP/du| du,$$

where

$$dP/du = ((dx(u)/du), (dy(u)/du), (dz(u)/du))$$

and

$$|dP/du| = \text{SQRT}((dx(u)/du)^2 + (dy(u)/du)^2 + (dz(u)/du)^2)$$

- For a cubic curve  $P(u) = au^3 + bu^2 + cu + d$  this will mean that the derivative of one of the 3 eq with respect to  $u$  is  $dx(u)/du = 3a_x u^2 + 2b_x u + c_x$  and under the SQRT one would have a curve of 4th degree  $Au^4 + Bu^3 + Cu^2 + Du + E$
- With a bit of computations one can compute then  $A, B, C, D$

# Arc length: Estimating through forward differences

- Suppose we have  $P(u)$ .
- One can compute a table of the distance of  $P(u)$  from the point  $P(0)$  at regular intervals:

$P(0), P(\Delta u), P(2\Delta u), \dots, P(1)$

that is, containing

$$P((i+1)\Delta u) - P(i \Delta u)$$

U

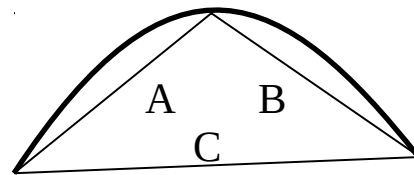
- One can interpolate these values first order (or higher order) to estimate the length of a segment in image space
- Conversely, one can use similar methods to deduce from the right hand column the corresponding value of  $u$
- Main problem with this approach is controlling the error

0	$ P(\Delta u) - P(0)  = G(\Delta u)$
$\Delta u$	$G(\Delta u) +  P(2\Delta u) - P(\Delta u)  = G(2 \Delta u)$
$2 \Delta u$	$G(2\Delta u) +  P(3\Delta u) - P(2\Delta u)  = G(3 \Delta u)$
...	...

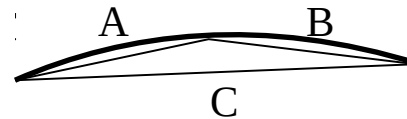


# Adaptive forward differences

- Since the relations between the variation of the parameter and the length of the curve is non-linear, the method of the last slide has problems when there is a big error
  - i.e. When the polyline implicitly used to estimate the parameter values inbetween table points is far from the actual curve
- This can be improved by computing the value of the midpoint of each interval between the table points.
  - if the sum of the sides A+B of the triangle is too different in length from the line joining the interval extre-mes C (over a threshold value), the midpoint is added to the list



Bad



Better

# Numerical meth.: Gaussian quadrature

- Another approach to computing length is based on numerics
- Computing the length of the curve implies computing the integral of the curve length
- Gaussian quadrature uses unevenly spaced intervals to achieve the greatest accuracy
- Gaussian quadrature computes

$$s = \int_{[0,1]} f(u) du \sim \sum_i w_i f(u_i)$$

U

- Since Gaussian quadr. is usually defined in the interval  $[0,1]$ , one has to reparametrize at first the original interval  $[a,b]$  we are considering

- This is achieved by using the new parameter  $t$  such that

$$t = (2u - a - b) / (b - a)$$

- Do not forget to apply the usual integral rules for changing parameters, that is adding the factor of parameter substitution to the integral

# Adaptive Gaussian quadrature

- If the curve derivative varies very fast in some areas, and less fast in other areas, the gaussian quadrature will either undersample part of the curve, or oversample
- In this case, a similar adaptive method to the one presented before can be used:
  - One subdivides intervals in half,
  - each half is evaluated using gaussian quadrature
  - The sum of the two halves is compared to the result of the whole interval.
  - If the difference is greater than a certain threshold, then the two halves are added to the sample points

# Finding u given s

- Suppose one wants to find the value of the parameter u at a given arc length s from the point  $R(u_1)$
- This equals to solving the equation
$$s - \text{LEN}(u_1, u) = 0$$
- Arc length is monotonic, so such a solution is unique as long as  $dR(u)/du$  is not 0

- Newton-Raphson integration can be used: generate the sequence  $\{p_n\}$

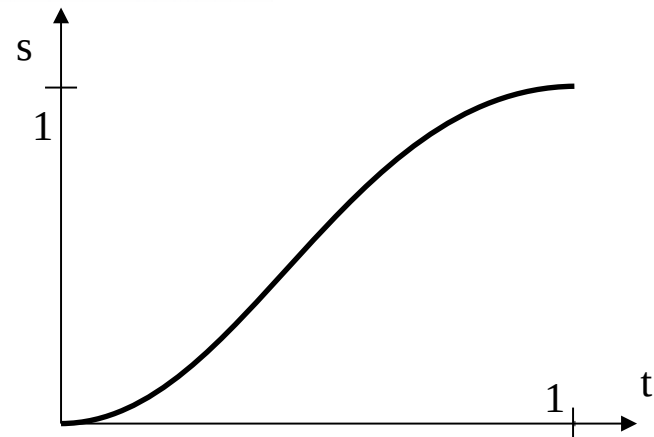
$$p_n = p_{n-1} - f(p_{n-1}) / f'(p_{n-1})$$

where

- f is  $s - \text{LEN}(u_1, P_{n-1}) = 0$  and can be evaluated at  $p_{n-1}$  using techniques of last slide
- $f'$  is  $dP/du$  evaluated at  $p_{n-1}$
- This eliminates the need for quadrature, and is faster
- But can have two problems:
  - Some  $p_k$  might not be on the curve, thus also  $p_{k+1}, p_{k+2}, \dots$  will not
  - When the derivative approaches 0 we divide by zero
- Use subdivision instead

# Speed control

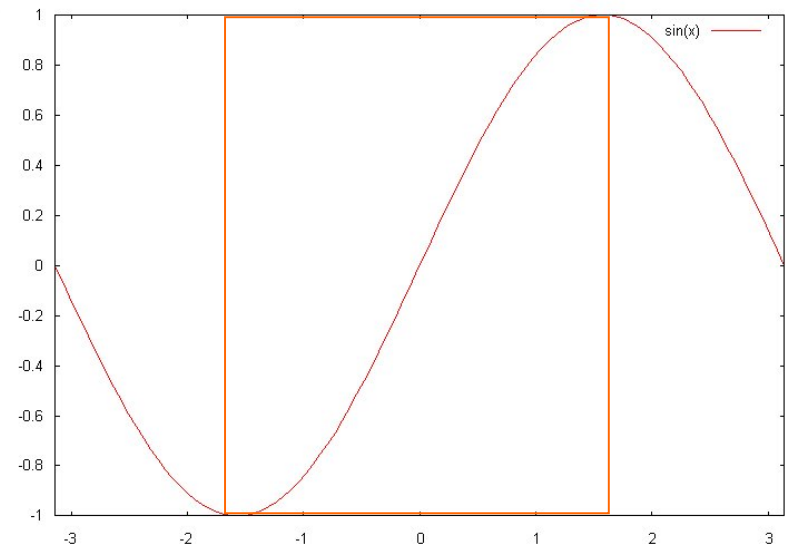
- On a arc-length parametrized curve, it is possible to control speed
- Simplest (and dullest) control: constant speed (equal space  $s$  in equal time  $t$ )
- Easiest speed control is *ease-in/ease-out*:
  - From standstill, accelerate until maximum speed
  - Decelerates and stop
- Speed along a curve can be controlled by varying arc length at something else than a linear function of  $t$ .
- The speed variations are seeable in the distance-time curve, which plots the space traversed  $s$  against the time  $t$ .
- Here is an example of a distance-time curve for ease-in



# Speed control: ease in/ease out

- There are different ways of mathematically achieve ease in/ease out
  - The first one is to use the sinus between  $-\pi/2$  and  $\pi/2$  and scaling the parameter to cover  $[0,1]$
  - $S(t)=(1/2)(\sin(\pi t-\pi/2)+1)$
- This curve can be split and joined with a straight line (take care of continuity at the splits) to add a period of constant speed

U



# Speed control: constant acceleration

- The computational cost of the sinus function is high.
- A better method is to use physics for the calculations:  $s=vt$ , and  $v=at$
- This obtains a parabolic ease-in function thus  $s=at^2$
- Similarly for deceleration one can use a constant (limited) deceleration until the object stops
- To describe the distance-time function of such a movement the following equations are used

- In formulas:

$$d=\frac{1}{2}at^2 \quad 0 < t < t_1$$

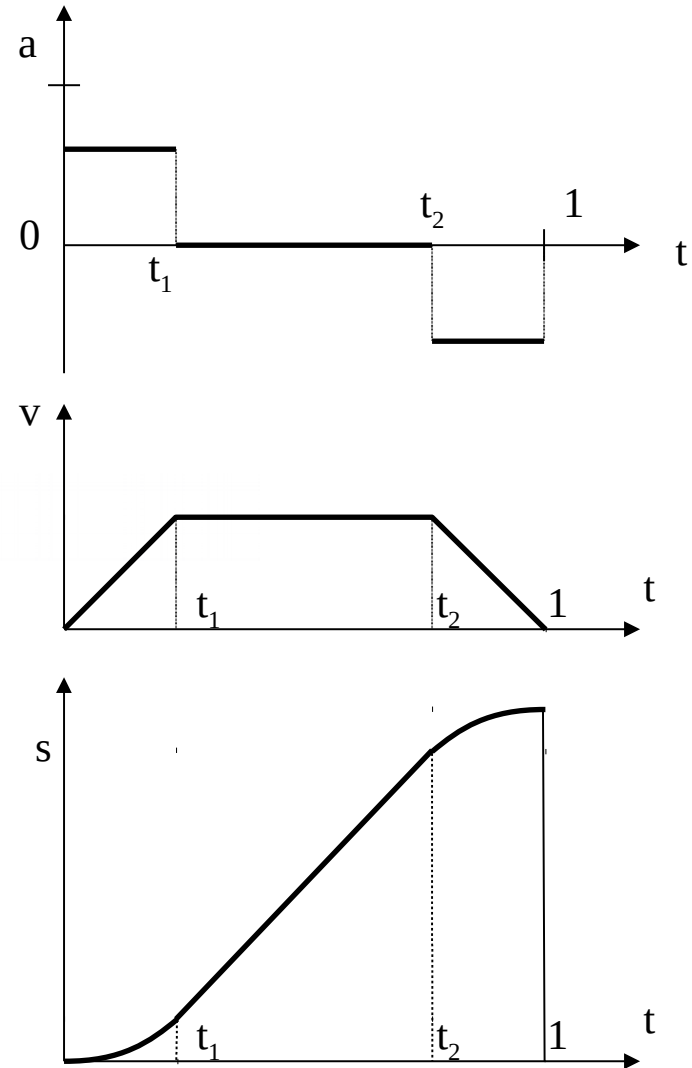
$$d=\frac{1}{2}at_1^2 + v_0(t-t_1) \quad t_1 < t < t_2$$

$$d=\frac{1}{2}at_1^2 + v_0(t-t_1) + \frac{v_0 - \frac{1}{2}at_2}{1-t_2}(t-t_2) \quad t_2 < t < 1$$

- Whereby  $v_0$  is the velocity when acceleration ends

# Speed control: constant acceleration

- $a = a_0$   $0 < t < t_1$   
 $a = 0$   $t_1 < t < t_2$   
 $a = -a_0$   $t_2 < t < 1$
- $v = v_0 t / t_1$   $0 < t < t_1$   
 $v = v_0$   $t_1 < t < t_2$   
 $a = v_0(1 - (t - t_2) / (1 - t_2))$   $t_2 < t < 1$
- The formulas look really complicated, but there are different ways to plot this to make it understandable





# General distance-time functions

- Many interesting aspects come up when allowing the user to control motion
- The more influence a user is given, the more problems come up
- Suppose the user defines some velocities at some points:
  - The rest of the velocity curve has to be fitted to these „fixed“ values
  - Sometimes leading to unwanted effects (reverse velocity to fit the time constraints)
- More intuitive is to control on the space-time curve
  - This because it allows to control velocities as a tangent, and to adapt the rest of the curve accordingly
- Motion control often requires specifying positions at specific times
  - The motion is specified as a series of constraints at a specific time, formally, a t-uple  $\langle t_i, s_i, v_i, a_i, \dots \rangle$
  - higher order approximation is needed for smooth movement

# Curve fitting

- If the animator specifies certain constraints then the time parametrized curve can be computed using these constraints as control points
- Suppose constraints are of the form  $(P_i, t_i)$  ( $i=1, \dots, j$ )

U

- It only requires to compute the curve passing through these points, i.e.

$$P(t) = \sum_{i=1}^n B_i N_{i,k}(t)$$

with  $2 \leq k \leq n+1 \leq j$

- In matrix form  $P = NB$
- Inverting this equation leads to find the control point values for the curve

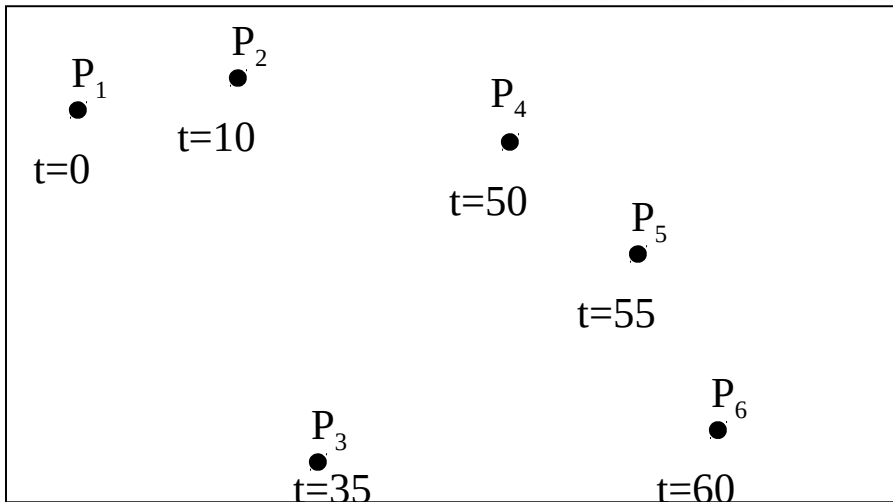
# Curve Fitting to position-time pairs

- Suppose the user gives the following positions and the corresponding times
- One can fit a B-spline curve to the values  $(P_i, t_i)$  ( $i=1, \dots, j$ ):
  - That is, take the general eq. of B-splines and make it pass through points
  - Find corresp. control points.
- Computing the curve passing through these points means computing  $P(t) = \sum_{i=1, \dots, n} B_i N_{i,k}(t)$  with  $2 \leq k \leq n+1 \leq j$
- In matrix form  $P = NB$ ,
- Inverting this equation leads to find the control point values for the curve:  $B = N^{-1}P$
- This is done through the pseudoinverse:  

$$P = NB$$

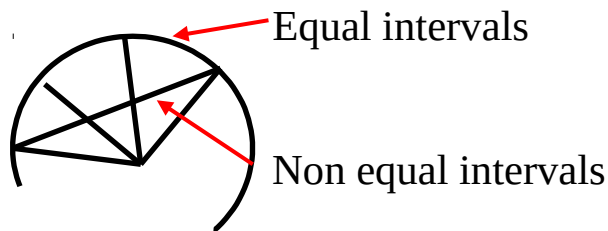
$$N^T P = N^T N B$$

$$[N^T N]^{-1} N^T P = B$$
- Remember the tradeoff: the higher the order, the higher the wiggling



# Interpolation of quaternion rotations

- A major reason for choosing quaternions is that they can be easily interpolated
- Quaternion form can be interpolated to produce good intermediate orientations
- This does not work easily with direct interpolation
- Unit quaternions are used to represent orientation, and can be seen as point of on the unit sphere in 4-dimensional space
- To interpolate between two unit quaternions, one can linearly interpolate
- But this will not produce constant speed rotation, because a path on a sphere is not the same as a path on a plane (which is what linear interpol. follows)
- Equal speed interpolations can be computed by interpolating directly on the path on the sphere



# Interpolation of quaternion rotations

- The problem (of course) is how to do that
- Remember:  $q=[s,v]$  and  $-q=[-s,-v]$  represent the same orientation
- So interpolation from  $q_1$  to  $q_2$  can be also carried between  $q_1$  and  $-q_2$ .
- The difference is that one path will be longer
- The shorter one is the one distinguished by the smallest angle
- One can compute the cosine of the angle between  $q_1$  and  $q_2$ :

$$\cos\theta = q_1 \cdot q_2 = s_1 \cdot s_2 + v_1 \cdot v_2$$

- If it is positive, then shortest path is from  $q_1$  to  $q_2$
- Else shortest path is from  $q_1$  to  $-q_2$


# Interpolation of quaternion rotations

- So, the spherical linear interpolation (SLERP) between  $q_1$  and  $q_2$  with parameter  $u \in [0,1]$  is

$$\text{SLERP}(q_1, q_2, u) = \left( \frac{\sin((1-u)\theta)}{\sin\theta} \right) q_1 + \left( \frac{\sin(u\theta)}{\sin\theta} \right) q_2$$

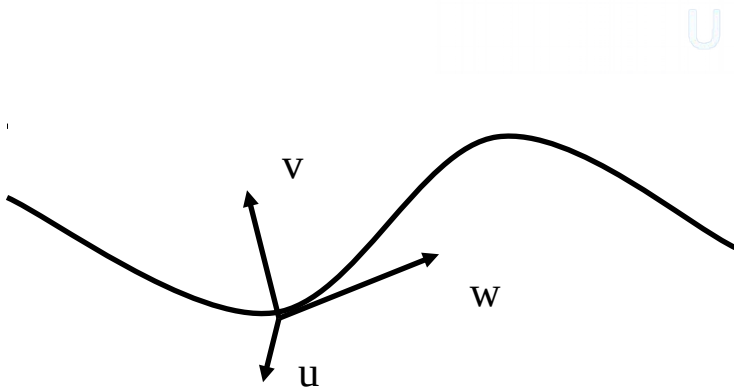
- Note that this does not generate a unit quaternion, so one has to normalize the result
- Notice that in the case  $u=1/2$ , SLERP is easy to compute except for a scaling factor
- Finally notice that if a chain of SLERPs is performed, it will perform similarly to linear interpolation (i.e. with rough changes)
- Higher order interpolations, based on Bezier curves, have been developed, but are beyond the purpose of this lesson

# Following a path

- Animating an object to move along a path is quite natural and common
- Not only following the path is needed: also moving the orientation
- Typically, one would have a local coordinate system associated with the object 
- Let the coordinates be  $(u, v, w)$ , and suppose they are right handed
- Suppose the origin of the coordinate system follows the curve  $P(s)$ , and that the movement of  $P(s)$  is specified
- Call POS the current position
- One can view the  $u, v, w$  coordinates as a view vector, an up vector and a vector perpendicular to  $u$  and  $v$
- This is similar to camera definition in Computer Graphics

# Following a path: Frenet Frame

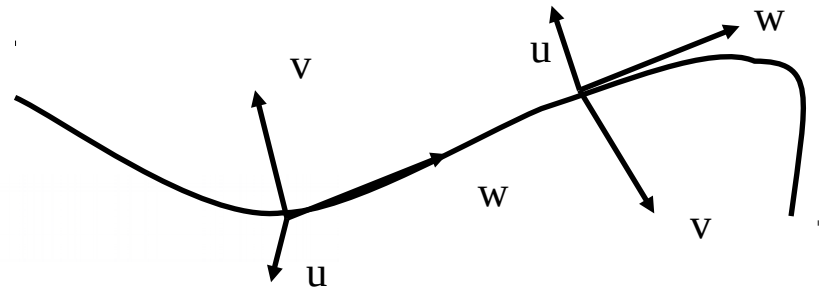
- The orientation of the camera system can be made dependent from the properties of the curve  $P(s)$
- A Frenet frame is given by the following axes definitions
  - $w$  follows the tangent of the curve (its first derivative  $P'(s)$ )
  - $v$  is orthogonal to  $w$  and in the direction of the second order derivative ( $P''(s)$ )
  - $u$  is the cross product of  $w$  and  $v$
- In symbols:
  - $w = P'(s)$
  - $u = (P'(s) \times P''(s))$
  - $v = w \times u$





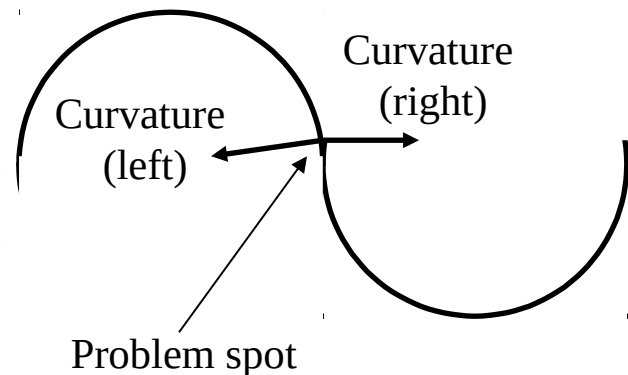
# Following a path: Frenet Frame

- Frenet frames are quite nice, but bear some flaws
- When the curve has no curvature, its second order derivative is zero. Here the Frenet frame is undefined
  - This problem can be solved by interpolating the Frenet frames at the start and end of the rectilinear trait
  - Since the tangent vector must be the same at the extremities, it is only a rotation that has to be interpolated



# Following a path: Frenet Frame

- A more complicated problem occurs at discontinuities in the curvature vector
- For example, when the path follows first a circle, and then a second circle
- At the problem point, the curvature will switch to pointing from one circle center to the other one
- Here, the Frenet frame is defined everywhere but is discontinuous
- Here, the object will rotate wildly along the path with „instant switches“



# Following a path: Frenet Frame

- The worst problem is that the path following is not so natural:
  - when we view at something, we do not look along the tangent
  - When we move, we anticipate curves
- Similar effect to your car light not following the road
- Also, one might want to make the object bend towards the interior to „anticipate the force“
- .... or, opposite, to let it bend out to give the effect of a force acting on the object



# Camera Path Following: Center of Interest

- A more natural way of specifying the orientation of a camera is to use the center of interest (COI)
  - One can view towards a fixed point
  - Or alternatively the center of an object
- Good method for a camera circling some arena of action
- The center of interest is specified, and so the view vector  $w = \text{COI} - \text{POS}$
- This leaves one degree of freedom in camera specification
- One simple way is to set the view vector  $v$  as viewing „up“, i.e. perpendicular to  $w$  and lying in the  $wy$  plane
$$w = \text{COI} - \text{POS}$$
$$u = w \times y$$
$$v = u \times w$$
- This works quite well for a camera moving along a path and focussing to a single object.
- When it gets very close to the object, this results in drastic changes (fly-near effect)
- This is not always bad!!!

# Camera Path Following: Center of Interest

- There are variations to specifying a fixed point
- One can for example specify various points on the camera path itself
- The up vector
  - is usually specified as lying in the  $wy$  plane
- But one can also allow the user to input
  - Either a tilting value with respect to the default up vector
  - Or the up vector on a whole
- Following a points on the path is relatively easy:
  - If  $P(s)$  describes the position on the curve, then  $P(s+\delta s)$ , with  $\delta s > 0$ , specifies its position in the future
  - It is advisable to choose points at equidistances on the curve, so as to make changes not that noticeable
  - Alternatively, one can take the baricenter of some future points to avoid too much hopping
- The real flaw of this method is the fact that camera views look jerky

# Camera Path Following: Center of Interest

- A better method is to use instead of some function of the position path, a different function altogether for the POI
- Let  $P(s)$  be the curve of the camera path, and  $C(s)$  the curve of the COI (obviously the animator specifies this)
- Similarly, and up vector path must be specified  $U(s)$ , so that the general up direction is  $U(s) - P(s)$
- The resulting coordinates for the camera will then become
$$w = C(s) - P(s)$$
$$u = w \times (U(s) - P(s))$$
$$v = u \times w$$
- This gives maximum control, but is also difficult to control.
- An easy way of specifying  $C(s)$  is to use fixed positions, with ease-in/ease-out moves between the different fixed points

# Smoothing paths

- There are several ways to smooth a path if it has been generated by a sample process, such as a motion capturing system
- This path acquisition method is getting more and more frequent and inexpensive
- However, data here can be prone to noise or imprecision, depending on the input method



Courtesy Animazoo Ltd.

# Smoothing paths: linear interpolation

- The simplest way of smoothing the data is to average neighbouring data point.
- Suppose we have the chain of points  $\{P_i\}_{i=0,N}$
- In the simplest form, one averages  $P_i$  as the average itself and of  $P_{i-1}$  and  $P_{i+1}$ .
- Obviously, here the „spikes“ are flattened, so applying this method many times makes little sense

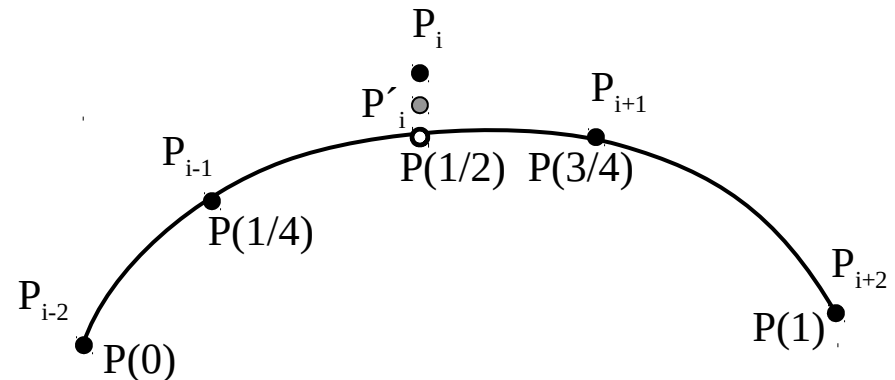
$$P'_i = \frac{P_i + \frac{P_{i-1} + P_{i+1}}{2}}{2} = \frac{1}{4}P_{i-1} + \frac{1}{2}P_i + \frac{1}{4}P_{i+1}$$



# Smoothing paths: cubic interpolation

- A second method use the four adjacent points  $P_{i-2}, P_{i-1}, P_{i+1}, P_{i+2}$  on either side to fit a cubic curve that is then evaluated at the midpoint.
- This midpoint is averaged with the original point to obtain the smoothed point
- Remembering that a 3rd order curve was  $P(u)=au^3+bu^2+cu+d$

- One obtains  
 $P_{i-2}=P(0)=d$   
 $P_{i-1}=P(1/4)=a(1/64)+b(1/16)+c/4+d$   
 $P_{i+1}=P(3/4)=a(27/64)+b(9/16)+3c/4+d$   
 $P_{i+2}=P(1)=a+b+c+d$



# Smoothing paths: cubic interpolation

- For the last points, a parabolic arc can be computed to fit the second and forelast points
- Notice that here the curve will be of the form  $au^2+bu+c$  , and the equation turns into

$$P'_1 = P_2 + 1/3(P_0 - P_3)$$

and similarly for the last three points

# Smoothing paths: convolution kernels

- If the data can be viewed as a data function  $y_i=f(x_i)$  then convolution can be used to smooth the data
- Convolution with the convolution kernel  $g(u)$  defined in the interval  $[-s,s]$  is in fact computing
- The resulting integral can be computed directly or approximated by discrete means

$$P(x) = \int_{[-s,s]} f(x+u) g(u) du$$

# Smoothing paths: B-spline approximation

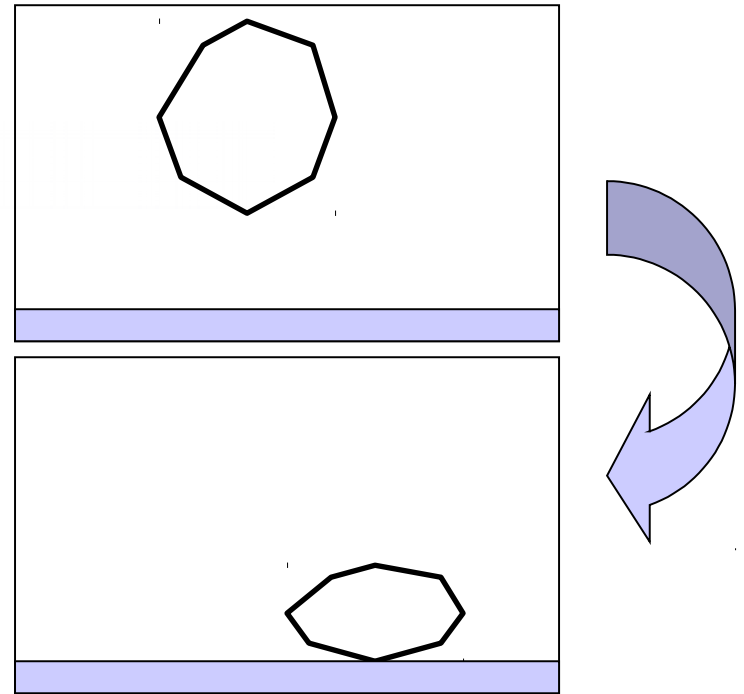
- If the path does not necessarily have to pass through the sample points, one can use approximation methods we saw before
- Particularly B-splines are well adapted for the defining a path tacked from real data

# Path along a surface

- If an object needs to follow a surface when it moves, then a path on the surface itself has to be found
- If we know start and endpoints, then this is simple:
  - trace a plane „perpendicular“ to the surface
  - Compute the intersection plane-surface
- Alternatively, other methods can be used, for example if one wants to follow the „valleys“ on the surface
- Here „greedy“ methods can be used, or methods that compute the normal to the surface and follow it

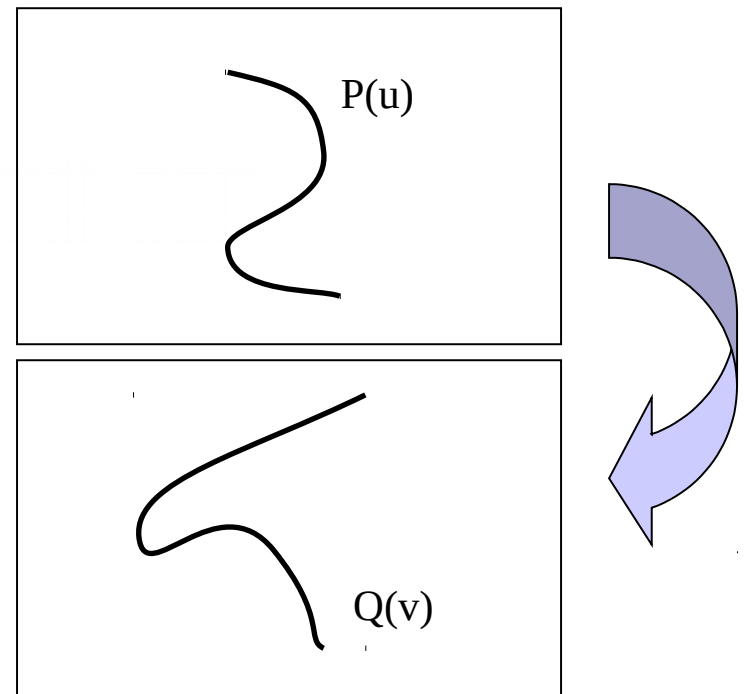
# Keyframe systems

- Early computer animation systems were keyframe systems
- Most were 2D too, and implemented keyframe animations made by hand
- In computer animation a key frame is a variable set by the user at specific timepoints
- The system interpolates intermediate frames from the key frames
- The interpolation is quite straightforward if the shapes to be interpolated have the same number of controlling points



# Keyframe systems

- In this case, linear interpolation can be used to produce the inbetween frames
- However, this is not the general case
- The general problem is: given two curves in 2D, how do I transform them into each other?
- If both curves are of the same type (eg Bezier of 3rd degree) then one can interpolate between control points
- Another method is to use interpolating functions to generate the same numbers of points on both lines, and then interpolate these points
- However, this does not allow sufficient control



# Keyframe systems

- Reeves proposed a method based on surface patch technology to solve the problem of interpolating a curve in time
- Basically, one defines a patch in 3D to join the curves and allow the time parameter to be interpolated
- Sample points are taken on the patch to define the intermediate curves (=curves at inbetweens)



# Animation languages

- In recent times, scripting languages have been developed to support animation systems
- Most animation languages are not easy to understand, and are close to hardcore programming
- A typical animation language is Renderman, or Alias/wavefront's MEL
- Their big advantage is control

# Animation languages

- Some effort has been put to accomodate unskilled artistic animators without scripting capabilities
- Simpler scripting languages such as ANIMA II have been developed
- Recently, actor based languages have appeared
- This is a novel approach but still at its infancy
- The idea is to have objects (=actors) and the instantiation of their variables representing the moving parameters
- Finally, the development of avatars has generated the need for some form of interaction with the animated models.



Copyright (c) 1988 ILM

+++ Ende - The end - Finis - Fin - Fine +++ Ende - The end - Finis - Fin - Fine +++