# Chapter 2

## Interpolation Techniques

# Chapter 2

## Animation

**Animator specified**
- **interpolation**
- **key frame**

**Algorithmically controlled**
- **Physics-based**
- **Behavioral**

**Data-driven**
- **motion capture**

Chapter 2

# Motivation

**<u>Common problem:</u> given a set of points Smoothly (in time and space) move an object through the set of points**

**<u>Example additional temporal constraints:</u> From zero velocity at first point, smoothly accelerate until time t1, hold a constant velocity until time t2, then smoothly decelerate to a stop at the last point at time t3**

Chapter 2

# Motivation – solution steps

1. **Construct a space curve that interpolates the given points with piecewise first order continuity** — **p=P(u)**
2. **Construct an arc-length-parametric-value function for the curve** — **u=U(s)**
3. **Construct time-arc-length function according to given constraints** — **s=S(t)**

**p=P(U(S(t)))**

# Chapter 2

## Interpolating function

Interpolation v. approximation
Complexity: cubic
Continuity: first degree (tangential), i.e. $C^2$
Local v. global control: local
Information requirements: tangents needed?

Chapter 2

# Interpolation v. Approximation

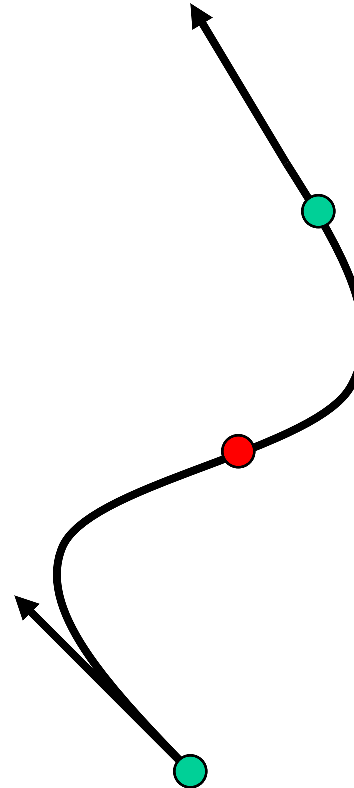
Chapter 2

# Complexity

**Low complexity**
**reduced computational cost**

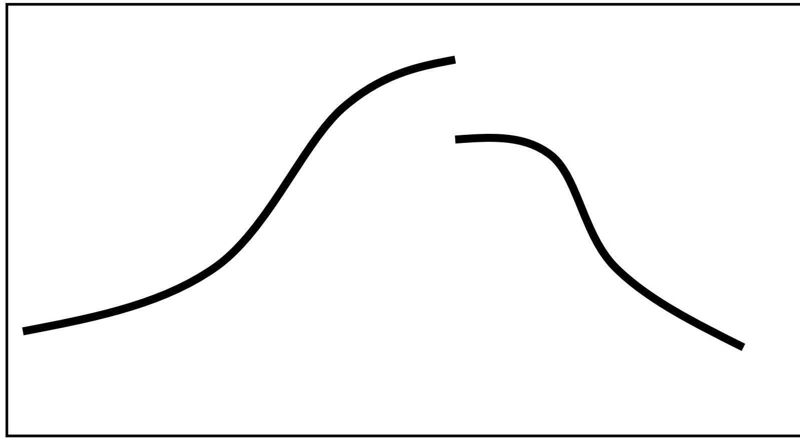
**Point of Inflection**
**Can match arbitrary tangents at end points**

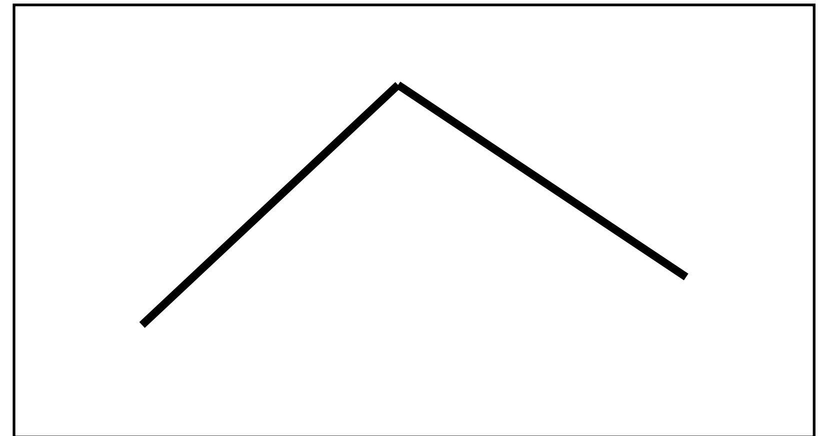
**Minimal requirement:**
**CUBIC polynomial**

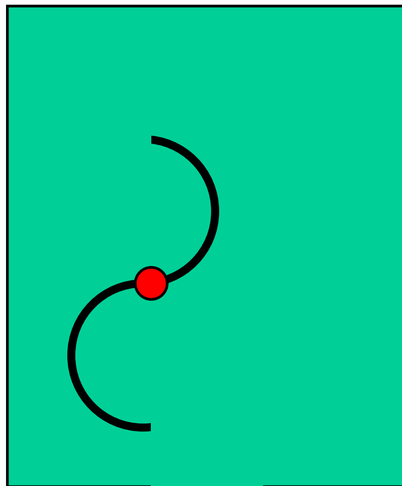


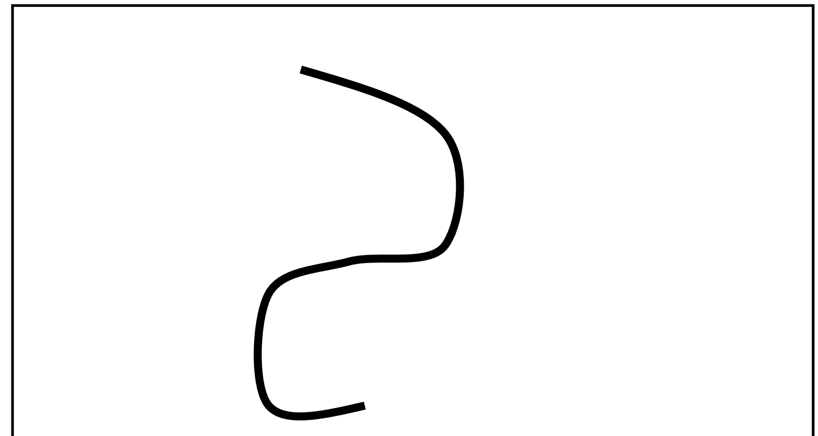
# Chapter 2

# Continuity

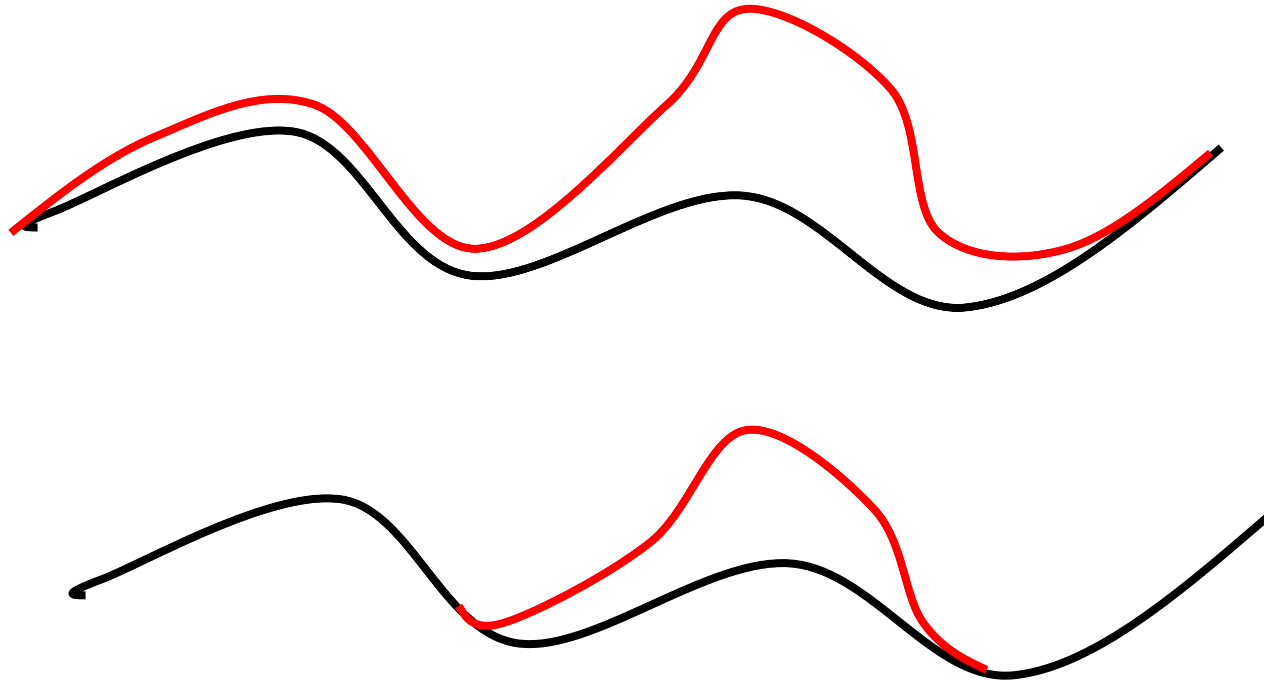








# Local v. Global Control

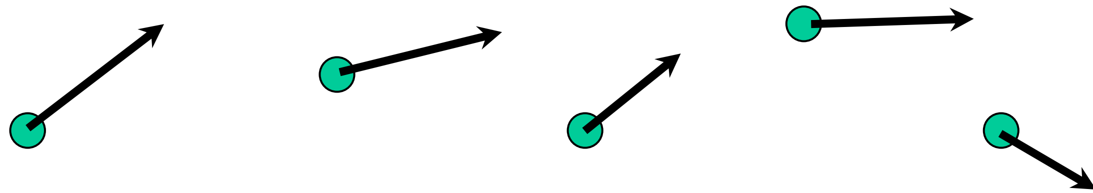


# Chapter 2

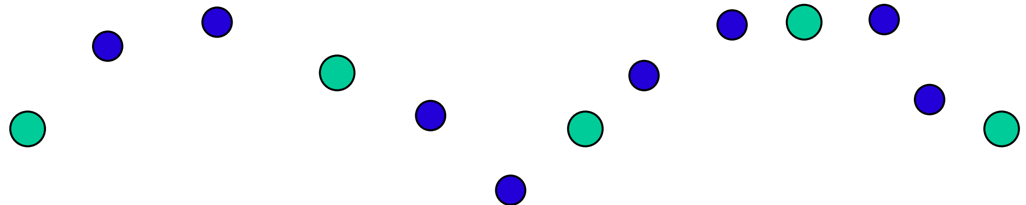
## Information requirements

just the points

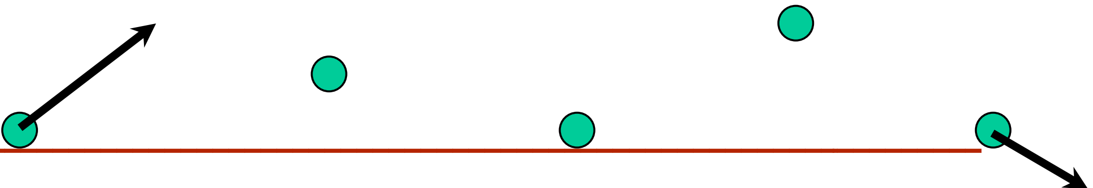
tangents



interior control points



just beginning and ending tangents



Chapter 2

# Curve Formulations

**Lagrange Polynomial**

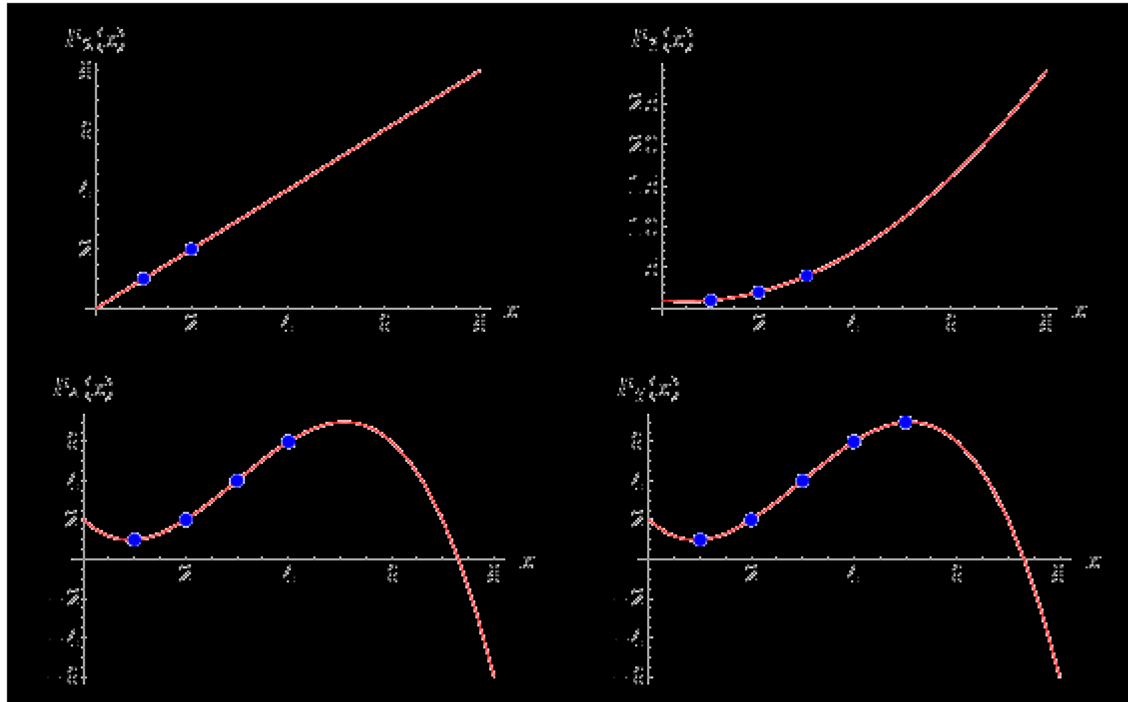
**<u>Piecewise cubic polynomials</u>**

**Hermite**
**Catmull-Rom**
**Blended Parabolas**
**Bezier**
**B-spline**
**Tension-Continuity-Bias**
**4-Point Form**

# Chapter 2

## Lagrange Polynomial

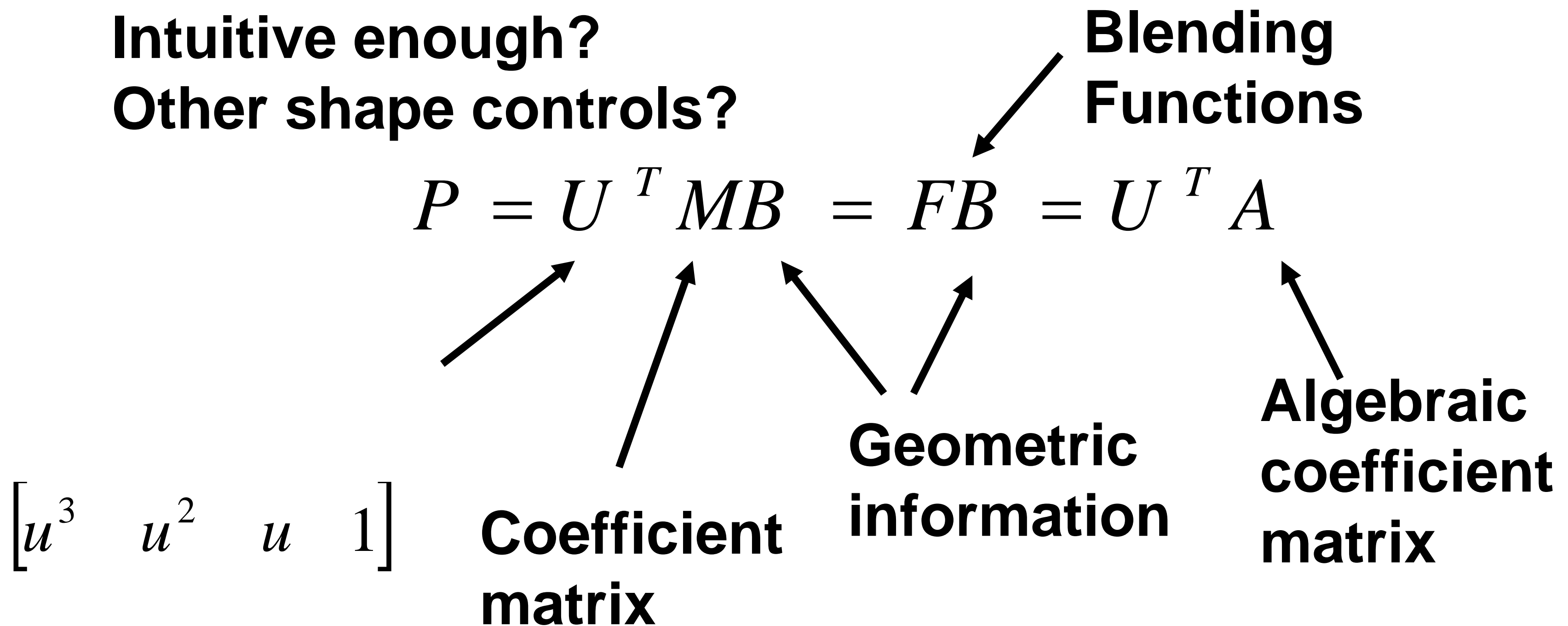
$$P_j(x) = y_j \prod_{\substack{k=1 \\ k \neq j}}^{x} \frac{x - x_k}{x_j - x_k}$$



# Chapter 2

# Polynomial Curve Formulations

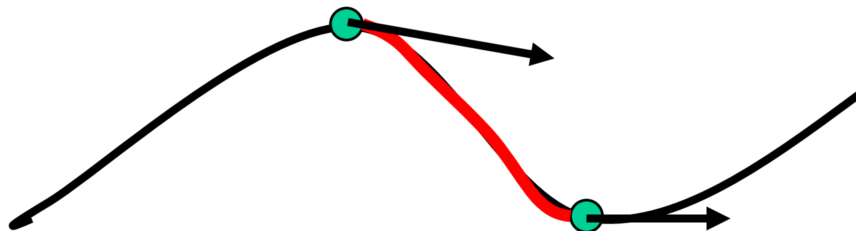
- Need to match real-world data v. design from scratch
- Information requirements: just points? tangents?
- Qualities of final curve?
- Intuitive enough?
- Other shape controls?



# Chapter 2

## Hermite

$$P = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2.0 & -2.0 & 1.0 & 1.0 \\ -3.0 & 3.0 & -2.0 & -1.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} p_i \\ p_{i+1} \\ p_i' \\ p_{i+1}' \end{bmatrix}$$

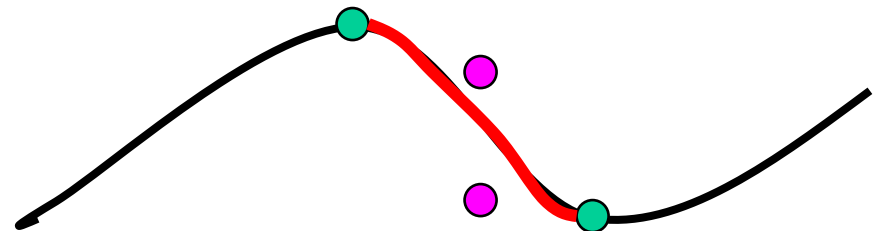


# Chapter 2

## Cubic Bezier

$$P = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2.0 & -2.0 & 1.0 & 1.0 \\ -3.0 & 3.0 & -2.0 & -1.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} p_i \\ p_{i+1} \\ p_i' \\ p_{i+1}' \end{bmatrix}$$

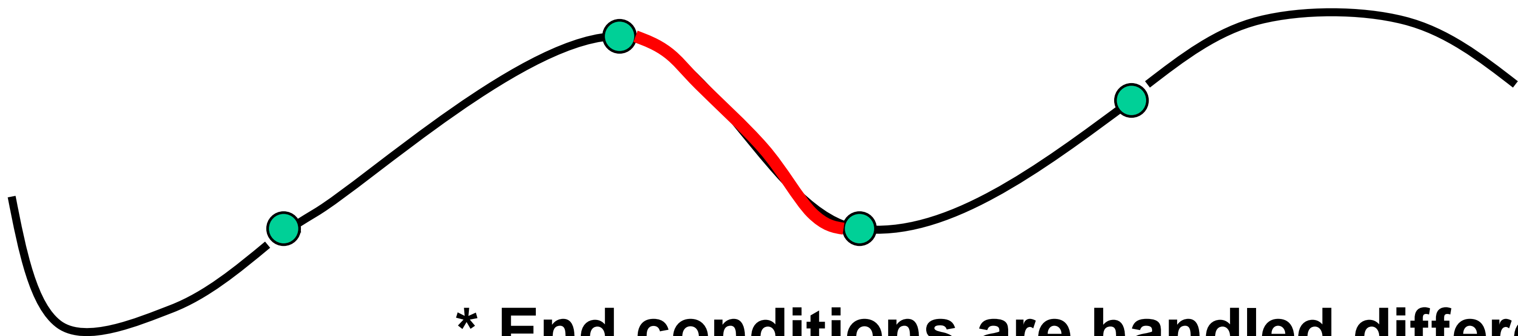
**Interior control points play the same role as the tangents of the Hermite formulation**



# Chapter 2

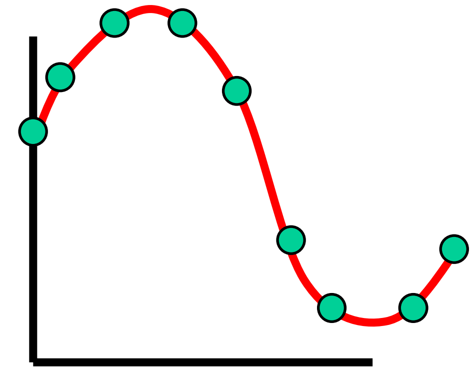
## Blended Parabolas/Catmull-Rom*

$$P = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{bmatrix}$$



*** End conditions are handled differently**

# Chapter 2



## Controlling Motion along p=P(u)

**Step 2. Reparameterization by arc length**

**u = U(s) where s is distance along the curve**

**Step 3. Speed control**

**for example, ease-in / ease-out**

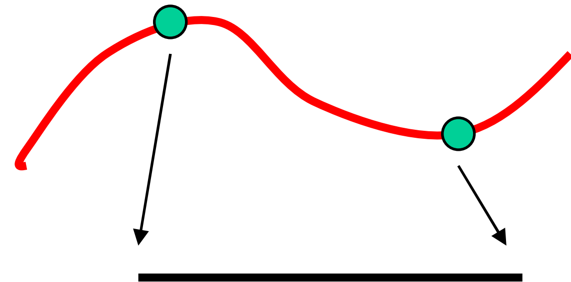
**s = ease(t) where t is time**

Chapter 2

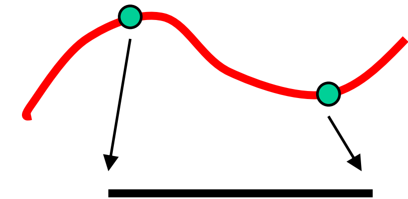
# Reparameterizing by Arc Length

**Analytic**
**Forward differencing**
- **Supersampling**
- **Adaptive approach**

**Numerically**
- **Adaptive Gaussian**



# Chapter 2
# Reparameterizing by Arc Length - analytic



$$P(u) = au^3 + bu^2 + cu + d$$

$$s = \int_{u_1}^{u_2} \left| dP / du \right| \quad du$$

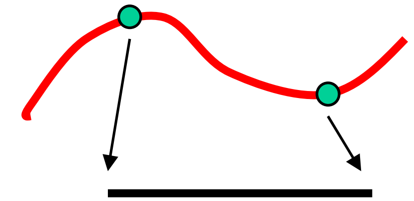
$$dP / du = \left( dx(u)/du \quad dy(u)/du \quad dz(u)/du \right)$$

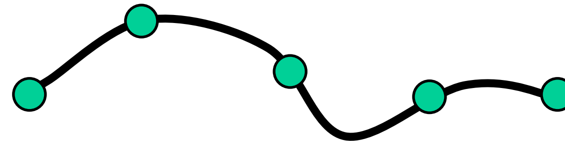
$$\left| dP / du \right| = \sqrt{\left( dx(u)/du \right)^2 + \left( dy(u)/du \right)^2 + \left( dx(u)/du \right)^2}$$

**Can't always be solved analytically for our curves**

Chapter 2

# Reparameterizing by Arc Length - supersample





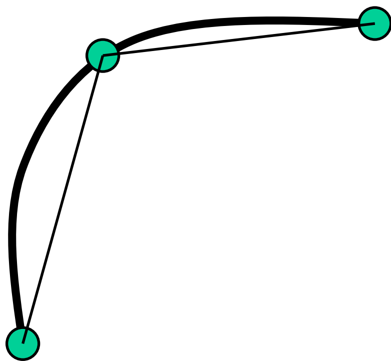
1. Calculate a bunch of points at small increments in u
2. Compute summed linear distances as approximation to arc length
3. Build table of (parametric value, arc length) pairs

**Notes**

1. Often useful to normalize total distance to 1.0
2. Often useful to normalize parametric value for multi-segment curve to 1.0

Chapter 2

# Build table of approx. lengths

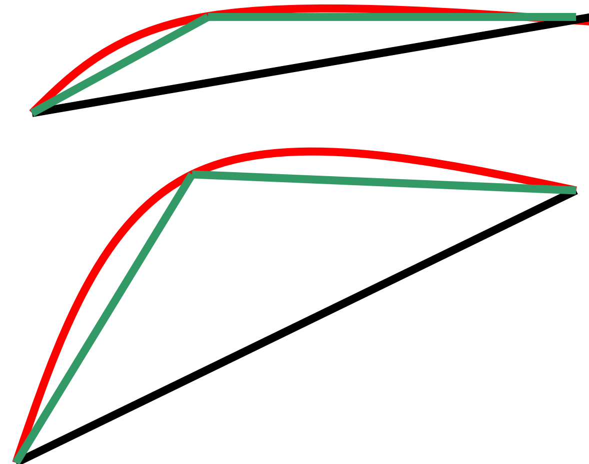


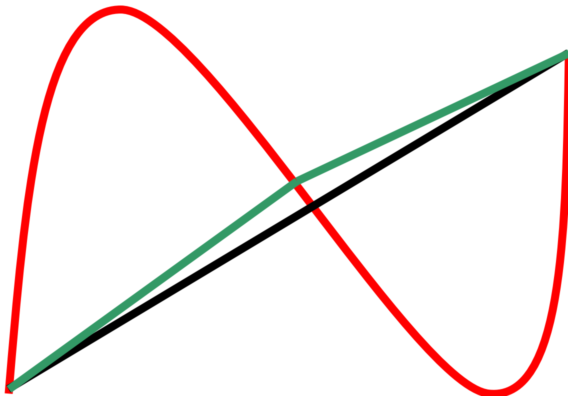
| index | u | Arc Length |
|---|---|---|
| 0 | 0.00 | 0.000 |
| 1 | 0.05 | 0.080 |
| 2 | 0.10 | 0.150 |
| 3 | 0.15 | 0.230 |
| ... | ... | ... |
| 20 | 1.00 | 1.000 |
| | | |

Chapter 2

# Adaptive Approach
# How fine to sample?

Compare successive approximations and see if they agree within some tolerance

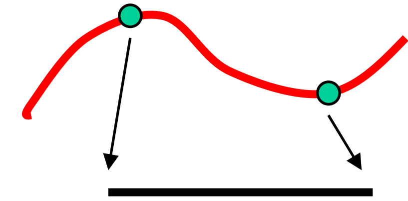




Test can fail – subdivide to predefined level, then start testing

# Chapter 2

# Reparameterizing by Arc Length - quadrature



**Reduce the number of evaluations using Gaussian quadrature**

$$\int_{-1}^{+1} f(u)du = \sum_i w_i f(u_i)$$

$$P(u) = au^3 + bu^2 + cu + d$$

$$\int_{-1}^{+1} \sqrt{Au^4 + Bu^3 + Cu^2 + Du + E}$$

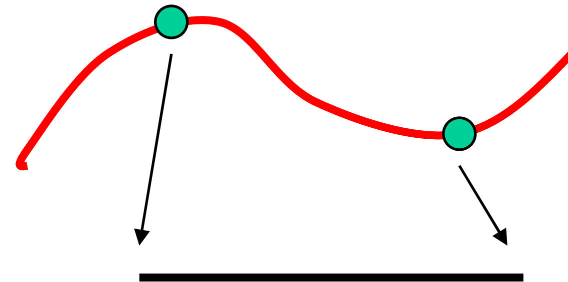
**Lookup tables of weights and parametric values**

**Can also take adaptive approach here**

Chapter 2

# Reparameterizing by Arc Length

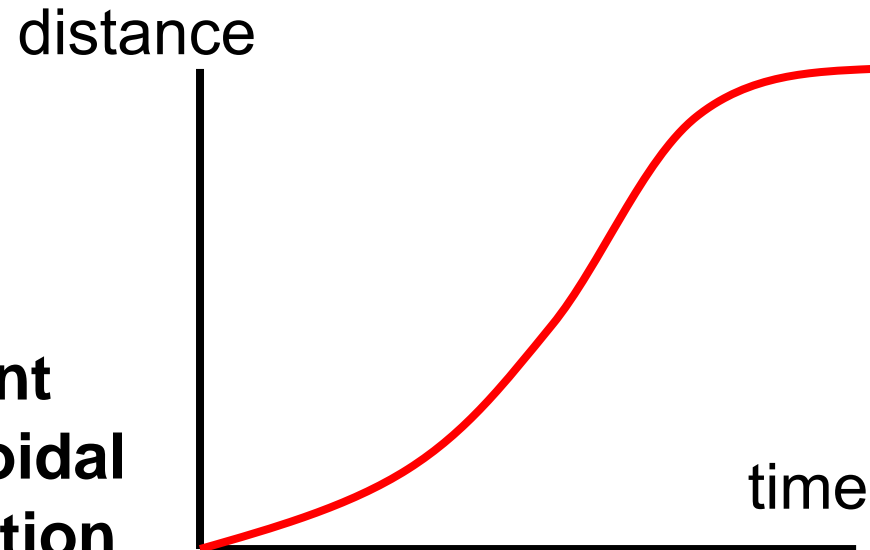
- Analytic
- Forward differencing
    - Supersampling
    - Adaptive approach
- Numerically
    - Adaptive Gaussian



Sufficient for many problems

Chapter 2

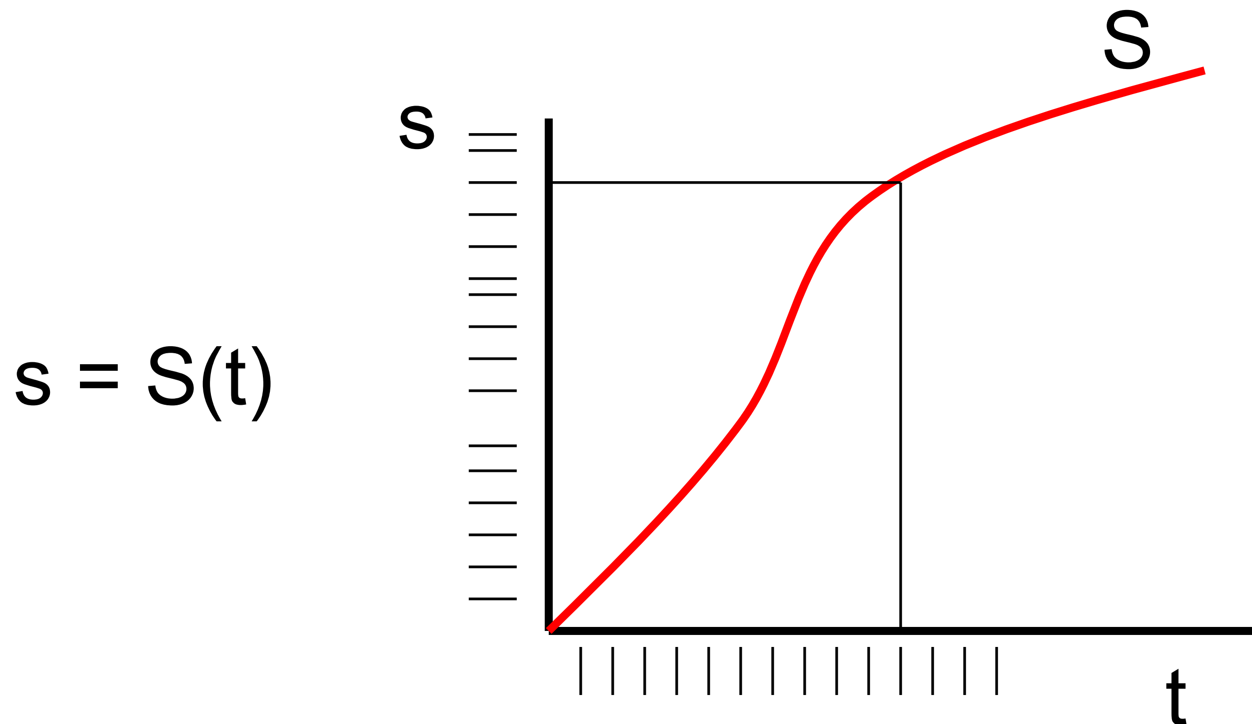
# Speed Control




- **Time-distance function**
  - **Ease-in**
    - **Cubic polynomial**
    - **Sinusoidal segment**
    - **Segmented sinusoidal**
    - **Constant acceleration**
  - **General distance-time functions**

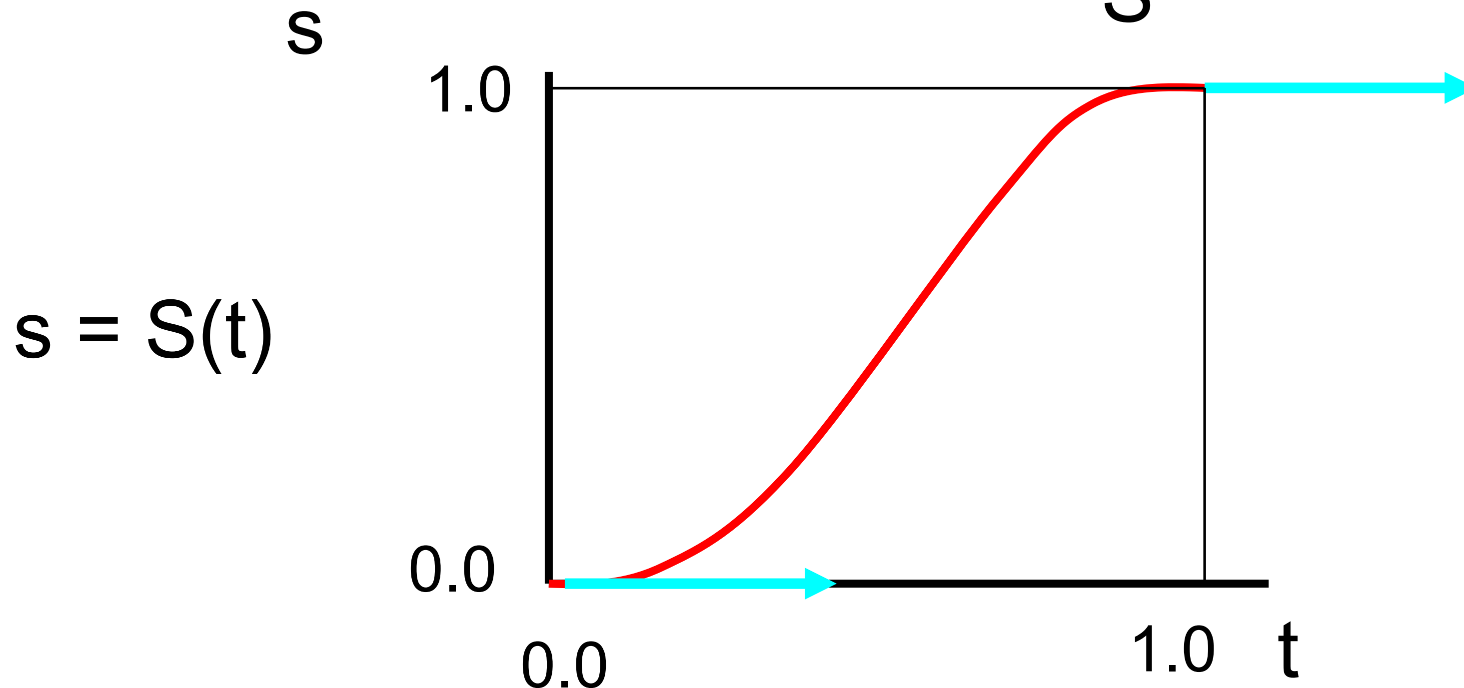
## Chapter 2

# Time Distance Function



# Chapter 2

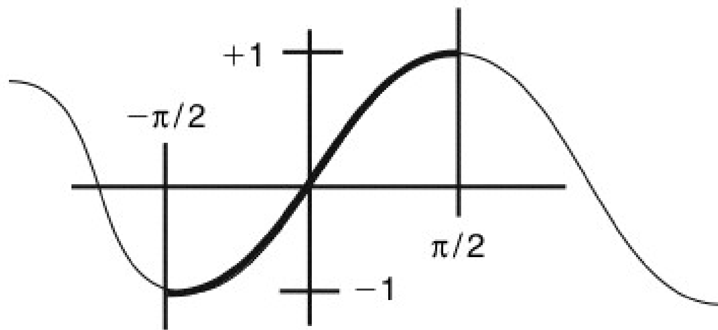
## Ease-in/Ease-out Function



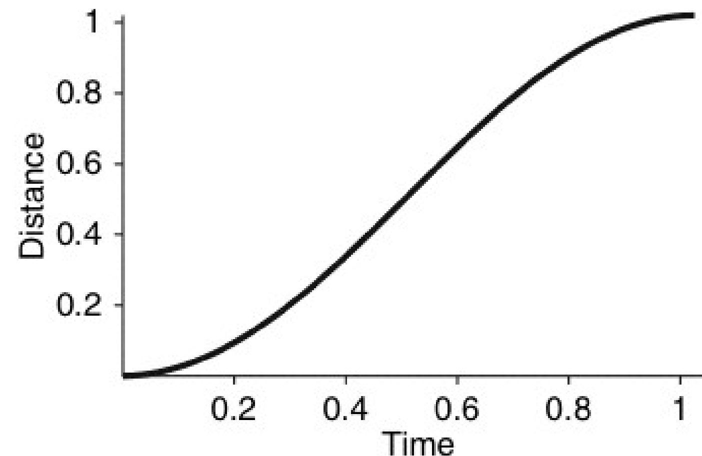

**Normalize distance and time to 1.0 to facilitate reuse**

# Chapter 2

## Ease-in: Sinusoidal




Sine curve segment to use as ease-in/ease-out control

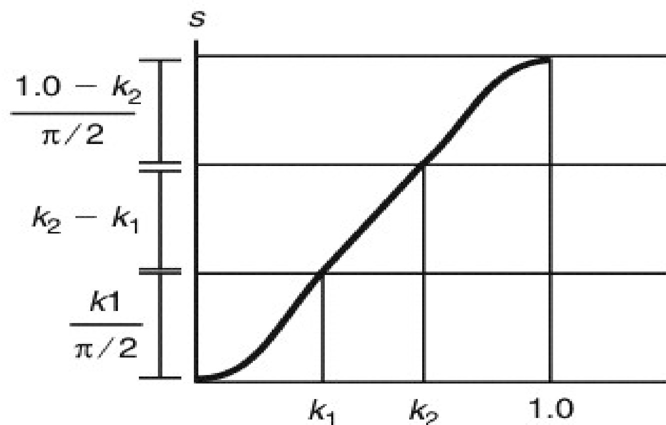



Sine curve segment mapped to useful values

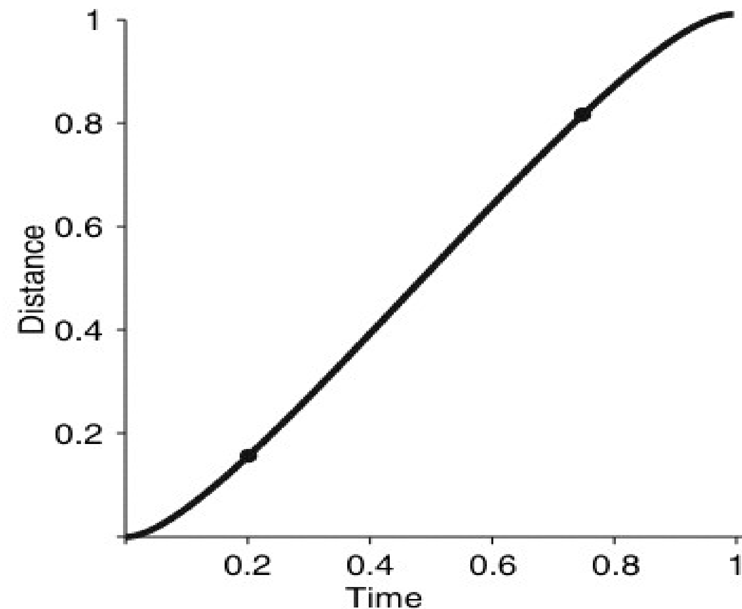
$$s = ease(t) = \left(\sin(t\pi - \pi/2\right) + 1)/2$$

Chapter 2

# Ease-in: Piecewise Sinusoidal




Ease-in/ease-out curve as it is initially pieced together




Curve segments scaled into useful values with points marking segment junctions

# Chapter 2

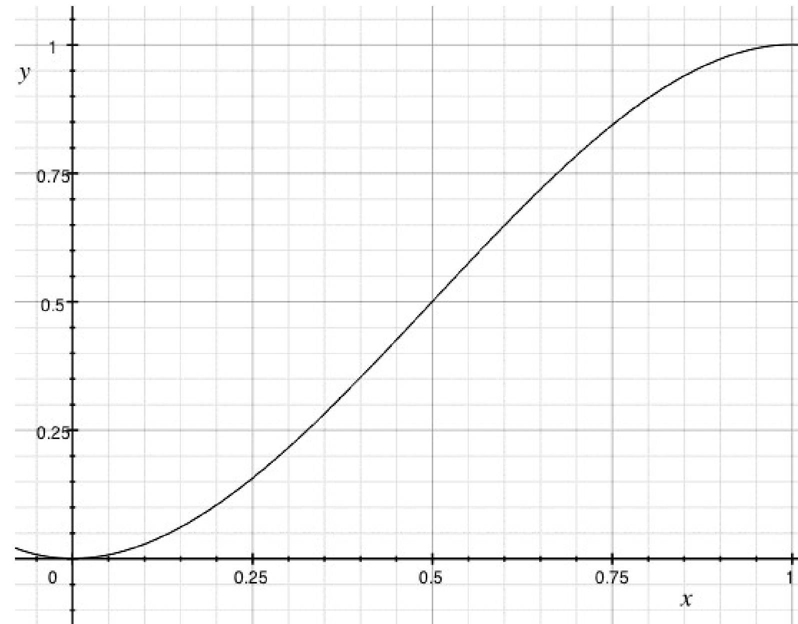
## Ease-in: Piecewise Sinusoidal

$$ease(t) = \begin{cases} (k_1 \dfrac{2}{\pi}(\sin(\dfrac{t\pi}{2k_1} - \dfrac{\pi}{2}))/f & t <= k_1 \\ (\dfrac{k_1}{\pi/2} + t - k1)/f & k_1 < t <= k_2 \\ (\dfrac{k_1}{\pi/2} + k_2 - k_1 + (1-k_2)\dfrac{2}{\pi}\sin(\dfrac{\pi(t-k_2)}{2(1-k_2)}))/f & k_2 < t \end{cases}$$

**Provides linear (constant velocity) middle segment**

Chapter 2

# Ease-in: Single Cubic

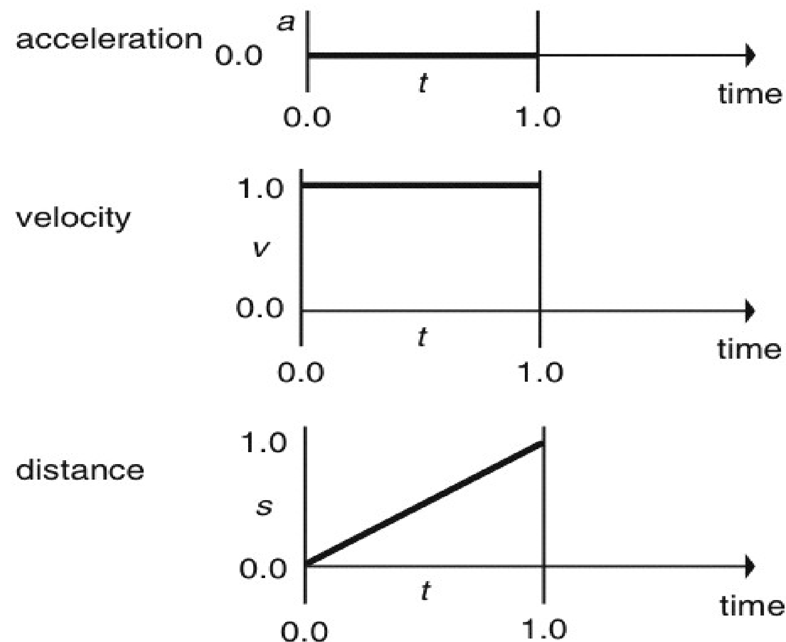



$$s = ease(t) = -2t^3 + 3t^2$$

**Drawback: no segment of constant speed**

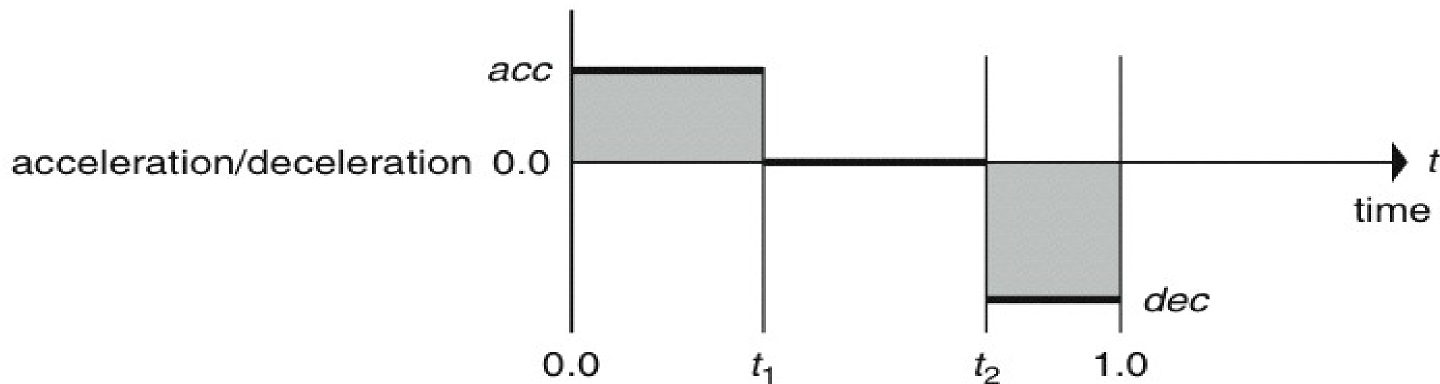
Chapter 2

# Ease-in: Constant Acceleration



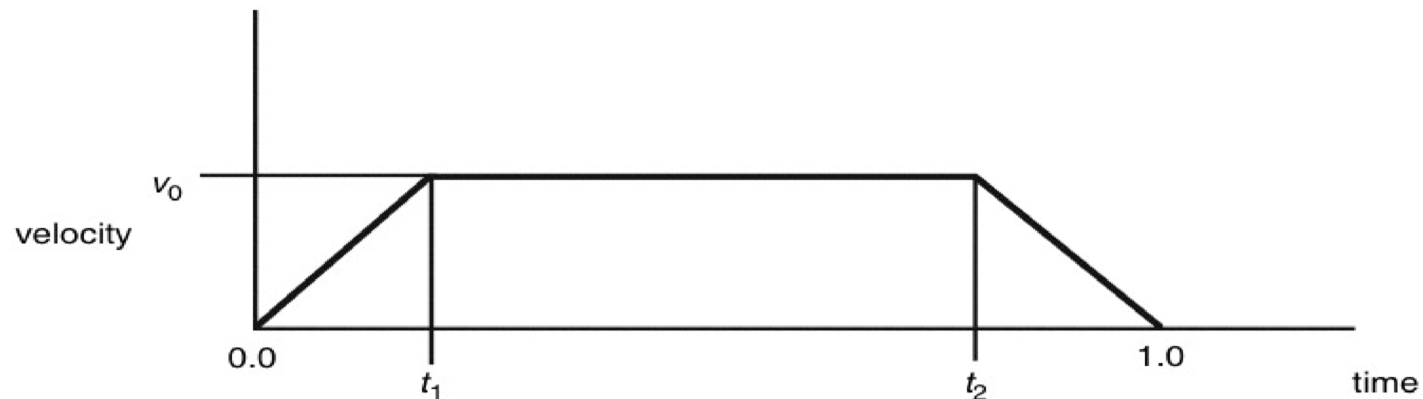
Chapter 2

# Ease-in: Constant Acceleration



# Chapter 2

## Ease-in: Constant Acceleration




$$v = v_0 \cdot \frac{t}{t_1} \qquad 0.0 < t < t_1$$

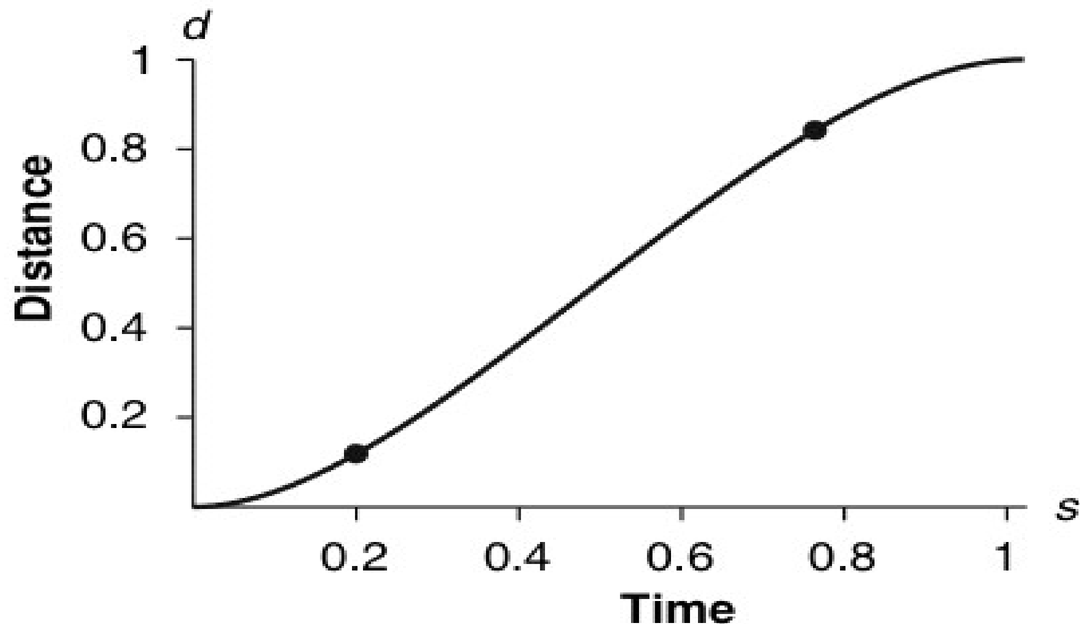
$$v = v_0 \qquad t_1 < t < t_2$$

$$v = v_0 \cdot \left(1.0 - \frac{t - t_2}{1 - t_2}\right) \qquad t_2 < t < 1.0$$

Chapter 2

# Ease-in: Constant Acceleration

**Integration of acceleration gives us desired function:**



Chapter 2

# Arbitrary Speed Control

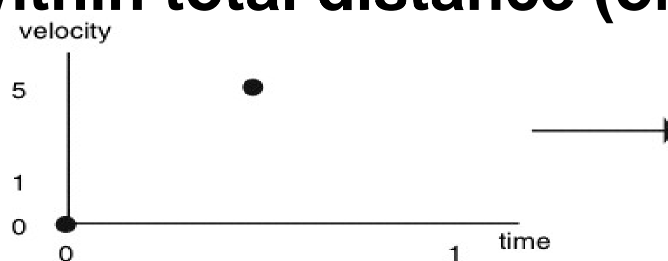
**Animators can work in:**

- **Distance-time space curves**
- **Velocity-time space curves**
- **Acceleration-time space curves**
- **Set time-distance constraints**
- **etc.**

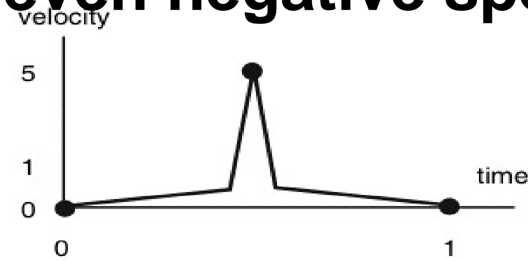
Chapter 2

# Curve fitting to distance-time pairs

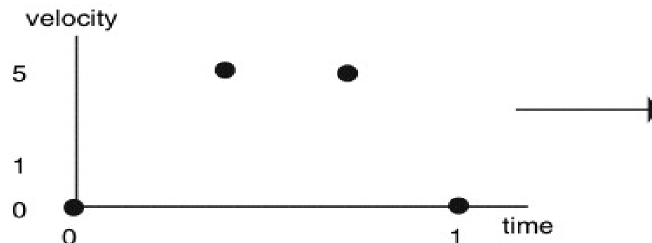
**Alternative: specify speed of camera at key points**
**This may result in undesirable spikes to stay within total distance (or even negative speed)**



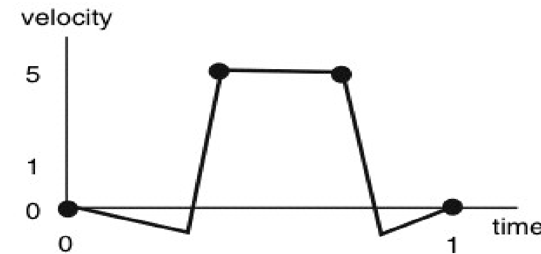
User-specified velocities



Possible solution to enforce total distance covered equal to one



User-specified velocities

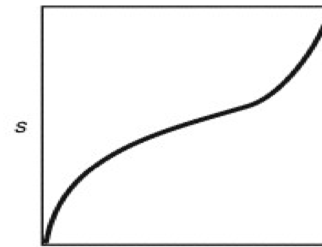


Possible solution to enforce total distance covered (signed area under the curve) equal to one. Negative velocity corresponds to

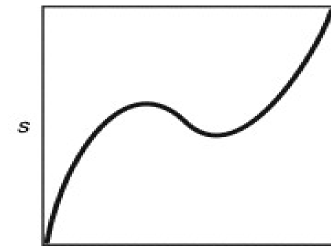
# Chapter 2

## Working with time-distance curves

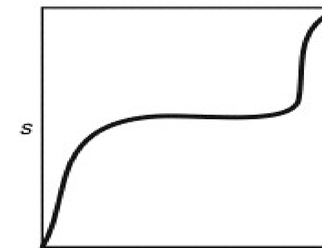
**Define a curve to parameterize according to distance which sometimes can be more intuitive**



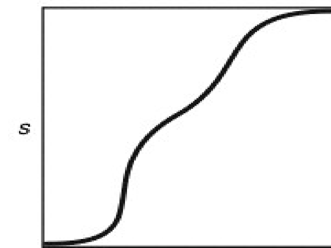
Starts and ends abruptly

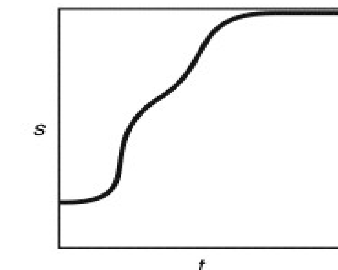


Backs up

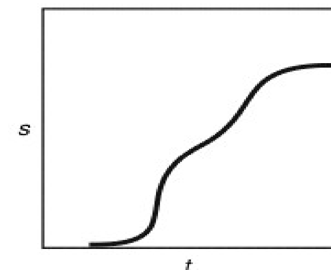


Stalls



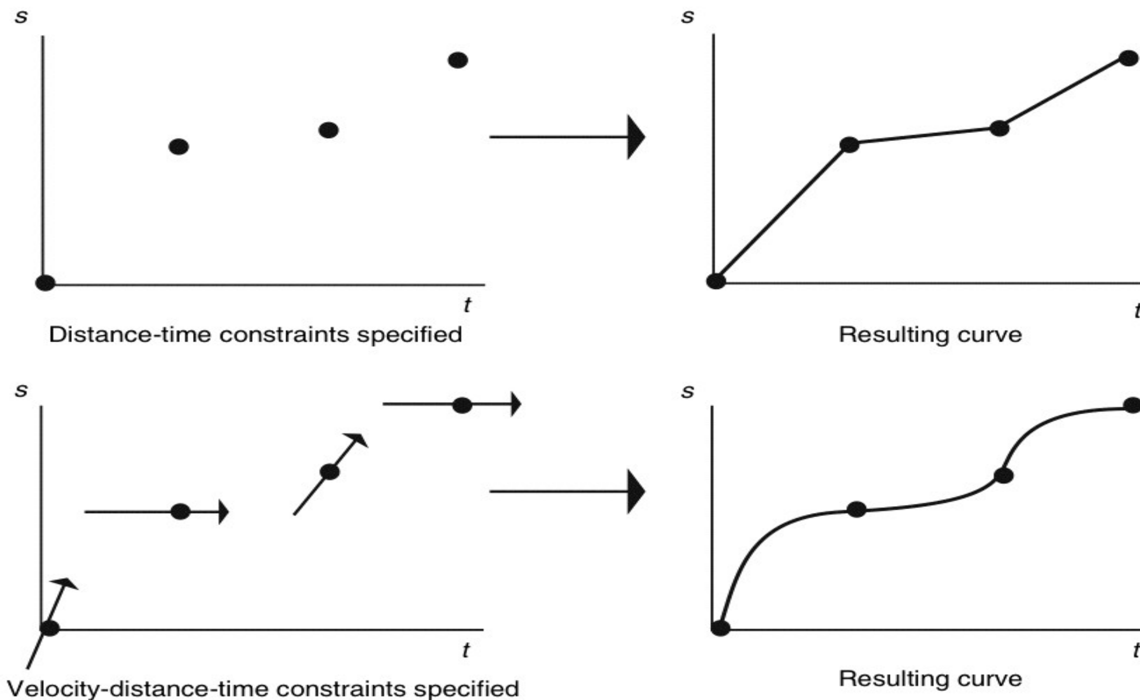
Starts partway along the curve and gets to the end before $t = 1.0$



Waits a while before starting and does not reach the end

Chapter 2

# Interpolating distance-time pairs



# Chapter 2

Now that we have the camera path defined:

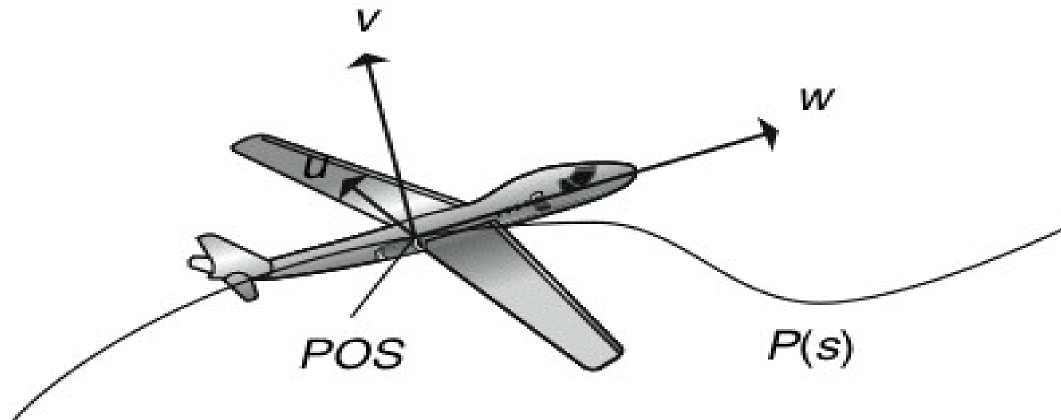
Is this all we need?

Looking at the function gluLookAt already tells us that we need more information in form of the up vector and the view direction or center point

Chapter 2

# Frenet Frame – control orientation

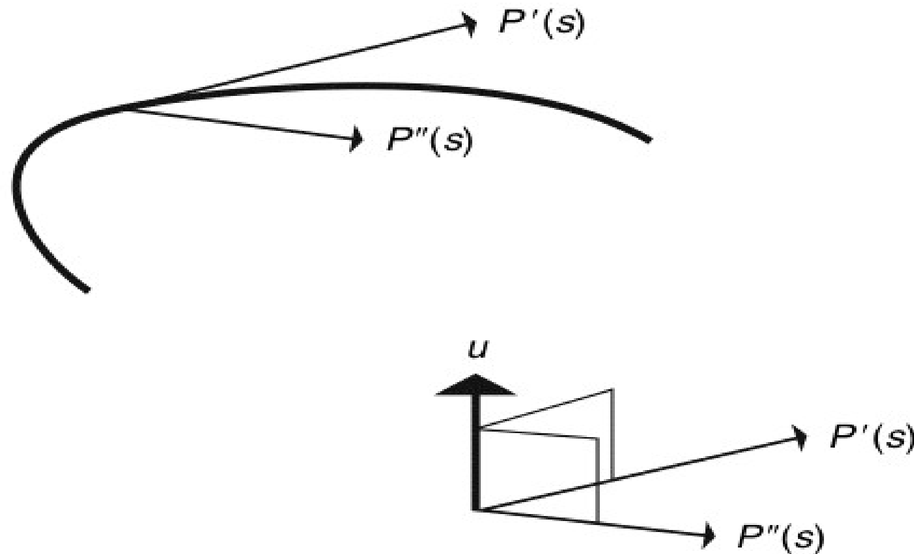
**For a camera, we also need to specify the orientation, i.e. view direction and up vector**




**Local coordinate system for the animated object**

# Chapter 2

## Frenet Frame
## tangent & curvature vector



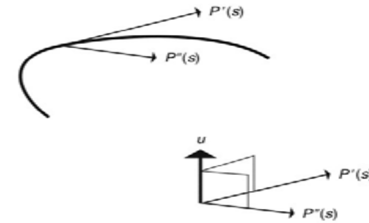
Chapter 2

# Frenet Frame tangent & curvature vector

$$P(u) = UMB$$

$$P'(u) =$$

$$P''(u) =$$




$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

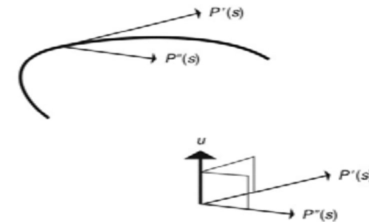
# Chapter 2

## Frenet Frame
## tangent & curvature vector

$$P(u) = UMB$$

$$P'(u) = U'MB$$

$$P''(u) = U''MB$$




$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

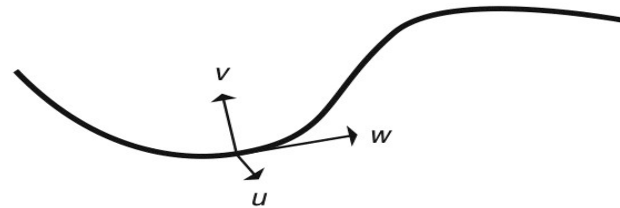
$$U' = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix}$$

$$U'' = \begin{bmatrix} 6u & 2 & 0 & 0 \end{bmatrix}$$

Chapter 2

# Frenet Frame local coordinate system

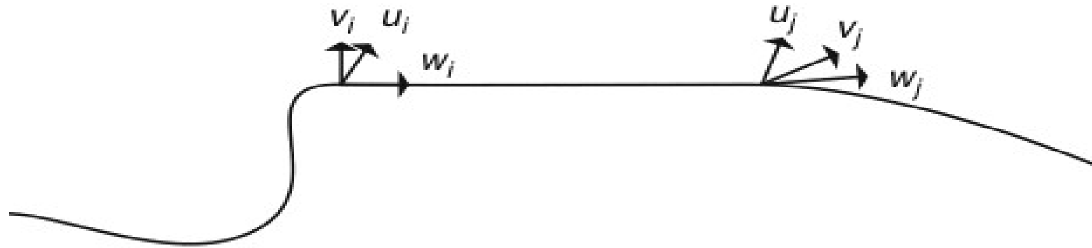
- Directly control orientation of object/camera



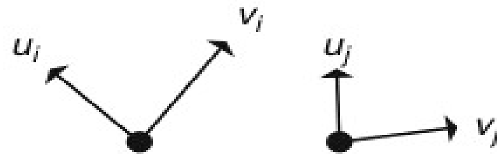

- Use for direction and bank into turn, especially for ground-planar curves (e.g. roads)

Chapter 2

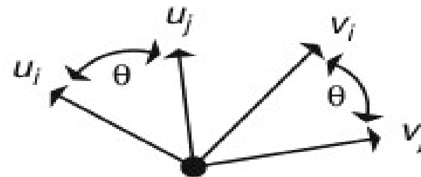
# Frenet Frame - undefined




Frenet frames on the boundary of an undefined Frenet frame segment becuse of zero curvature.




The two frames sighted down the (common) *w* vector.

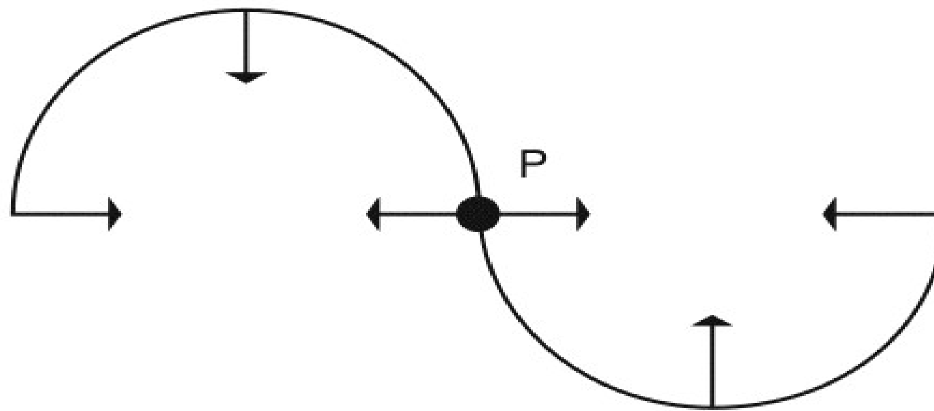



The two frames superimposed to identify angular difference.

Solution: interpolate between known vectors

Chapter 2

# Frenet Frame - discontinuity



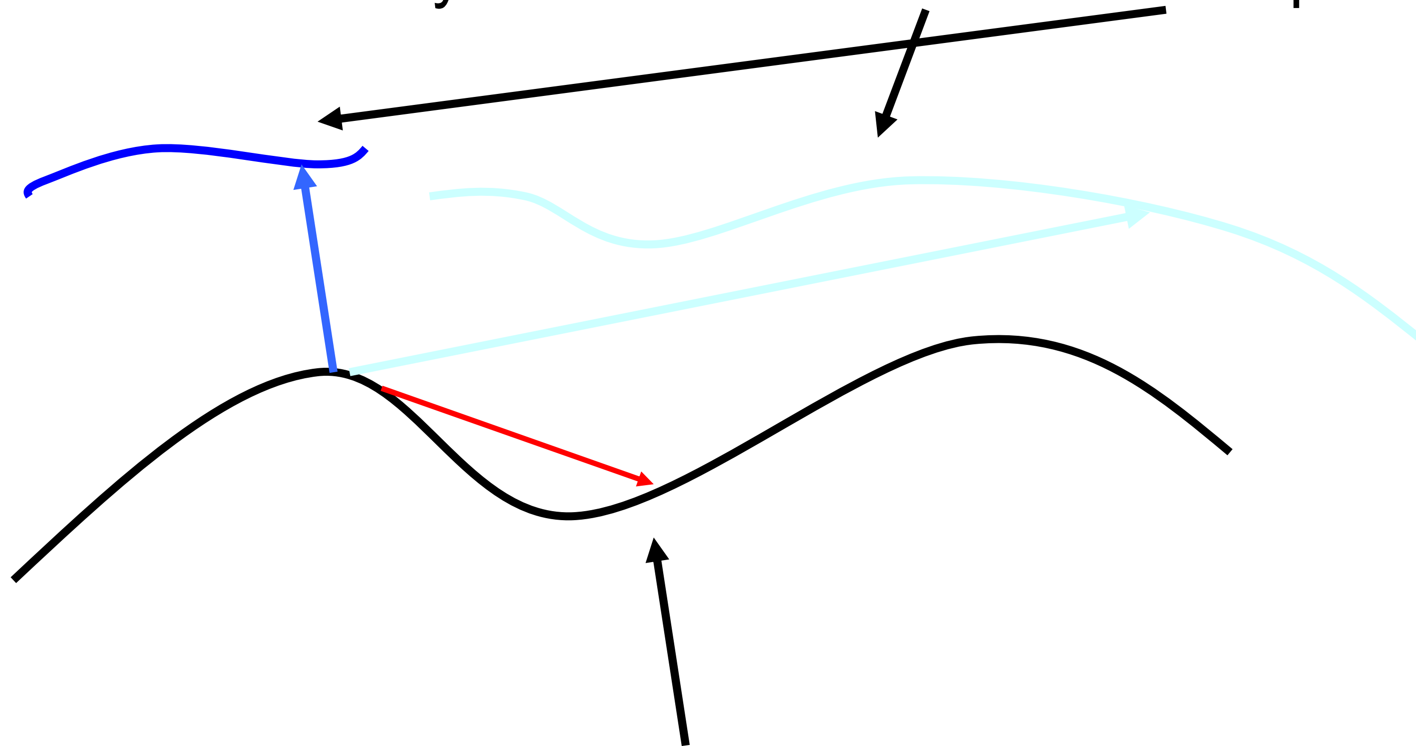

Problem: second derivative switches direction, hence flipping the camera around

In some applications, e.g. driving along a road, it may be more practical to just look farther ahead on the curve.

Chapter 2

# Other ways to control orientation

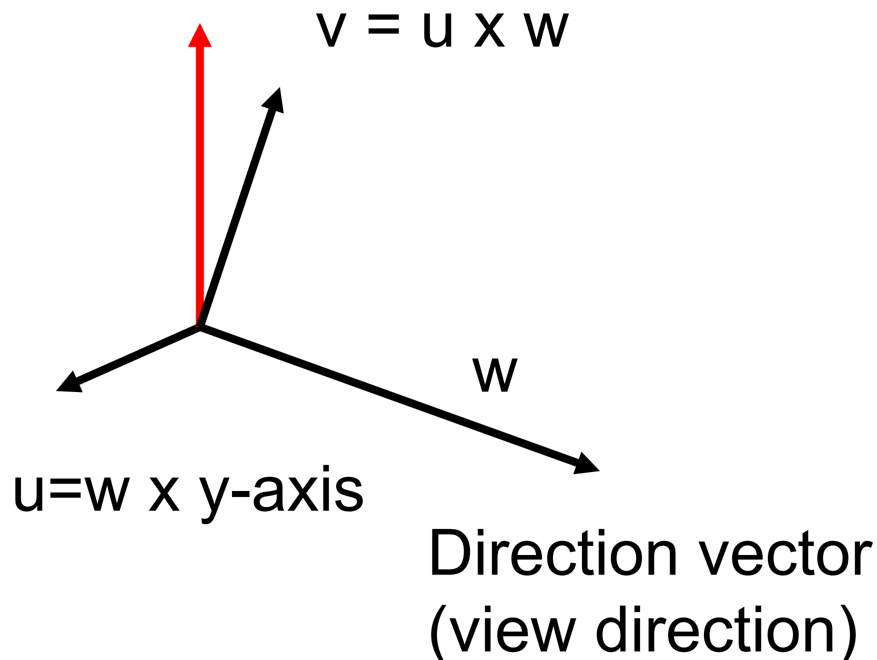
Use auxiliary curve to define direction or up vector



Use point P(s+ds) for direction

# Chapter 2

## Direction & Up vector




Direction vector
(view direction)

To keep 'head up', use y-axis to compute up vector v perpendicular to direction vector

Chapter 2

# Direction & Up vector

If an initial up vector is provided you can adjust the local coordinate system with every change of position:

$u = w \times v_{old}$

$v = u \times w$

This leads to an updated up vector v and avoids unintentional flipping of the camera.

Chapter 2

# Orientation interpolation

**Preliminary note:**

1. **Remember that** $Rot_q(v) \equiv Rot_{kq}(v)$
2. **Affects of scale are divided out by the inverse appearing in quaternion rotation**
3. **When interpolating quaternions, use UNIT quaternions – otherwise magnitudes can interfere with spacing of results of interpolation**
4. **Unit quaternions can be interpreted as points on a 4-D unit sphere**

Chapter 2

# Orientation interpolation

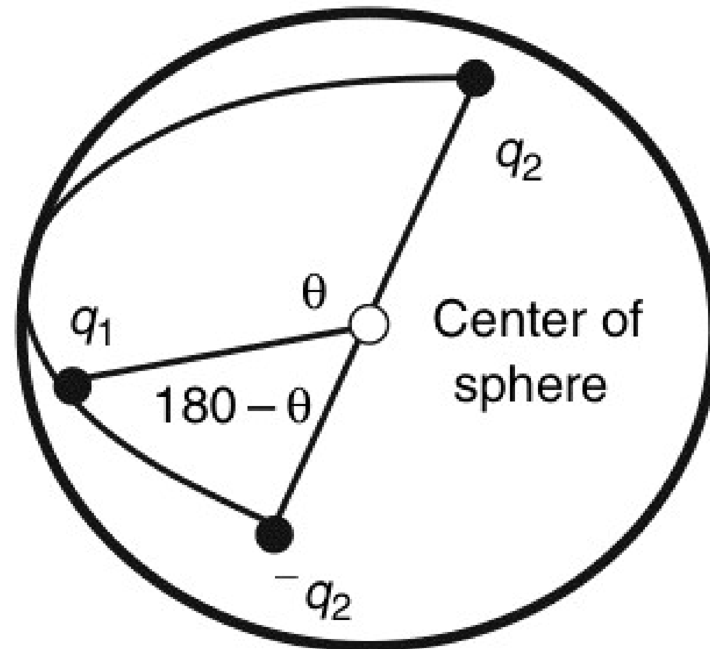
**Quaternions can be interpolated to produce in-between orientations:**

$$q = (1 - k)q_1 + kq_2$$

**2 problems analogous to issues when interpolating positions:**

1. **How to take equi-distant steps along orientation path?**
2. **How to pass through orientations smoothly (1st order continuous)**
3. **And another particular to quaternions: with dual unit quaternion representations, which to use?**

# Dual representation




$$Rot_q(v) = Rot_{kq}(v)$$

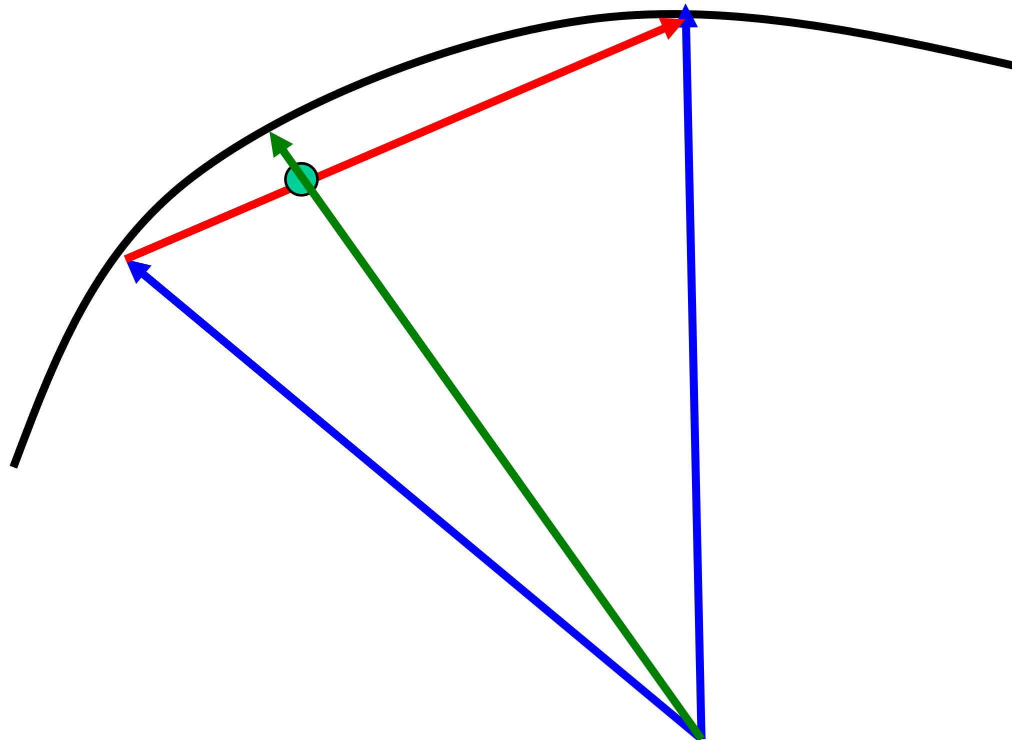
**Dual unit quaternion representations**

**For Interpolation between $q_1$ and $q_2$, compute cosine between $q_1$ and $q_2$ and between $q_1$ and $-q_2$; choose smallest angle**

Chapter 2

# Interpolating quaternions

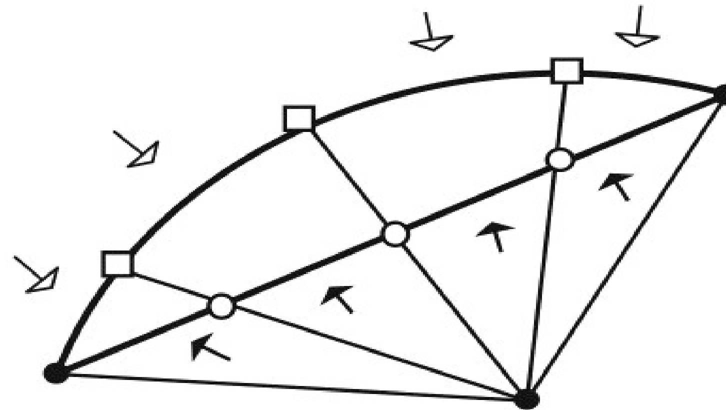
Unit quaternions form set of points on 4D sphere



**Linearly interpolating unit quaternions: not equally spaced**

Chapter 2

# Interpolating quaternions in great arc => equal spacing



Chapter 2

# Interpolating quaternions

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

where $q_1 \cdot q_2 = \cos\theta$

**'slerp', sphereical linear interpolation is a function of**

- **the beginning quaternion orientation, q1**
- **the ending quaternion orientation, q2**
- **the interpolant, u (interpolation parameter)**

**Note: resulting quaternions may have to be normalized**

Chapter 2

# Smooth Orientation interpolation

**When interpolating between series of orientations, slerping suffers from the same problem as linear interpolation between points in Euclidean space.**

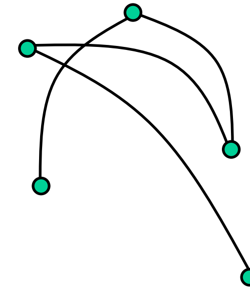
**Interpolate along great arc (in 4-space) using cubic Bezier on sphere**

1. **Select representation to use from duals**
2. **Construct interior control points for cubic Bezier**
3. **use deCasteljau construction of cubic Bezier**

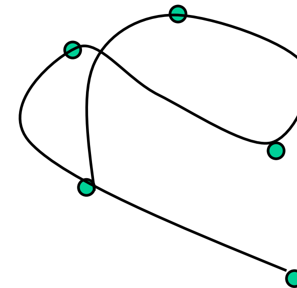
# Chapter 2

## Smooth quaternion interpolation

**Similar to first order continuity desires with positional interpolation**



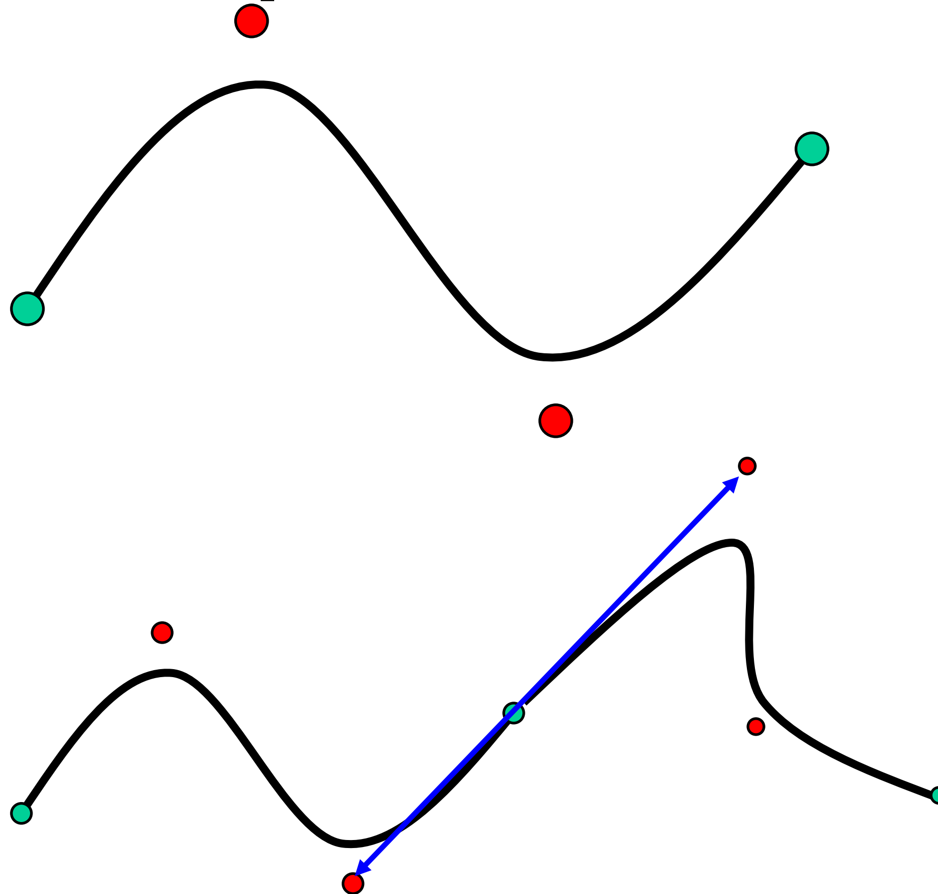
**How to smoothly interpolate through orientations $q_1$, $q_2$, $q_3$,…$q_n$**



**Bezier interpolation – geometric construction**

# Chapter 2

## Bezier interpolation



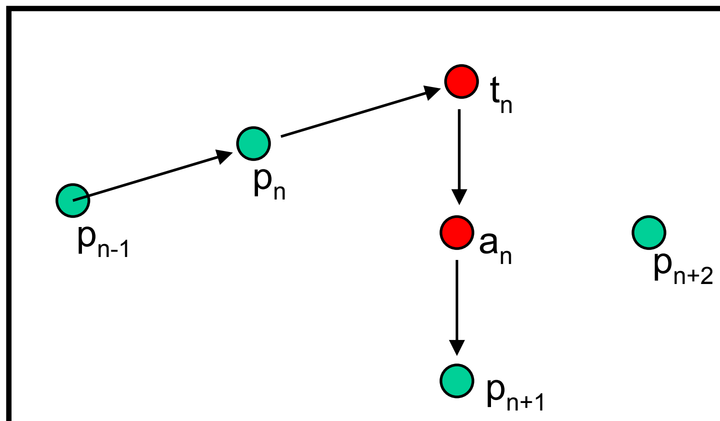
Chapter 2

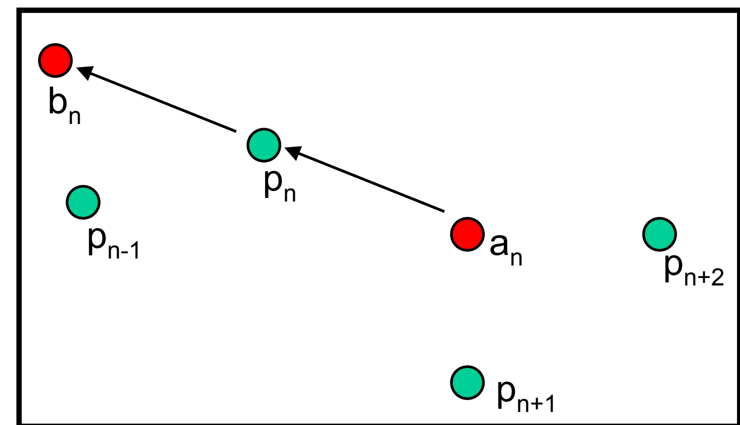
# Bezier interpolation

## Construct interior control points

Create interior control points $a_n$ based on $p_{n-1}$, $p_n$, and $p_{n+1}$: $a_n = 0.5 * (p_n + (p_n - p_{n-1}) + p_{n+1})$

Compute remaining interior points $b_n$ using $a_n$ and $p_n$: $b_n = p_n + (p_n - a_n)$



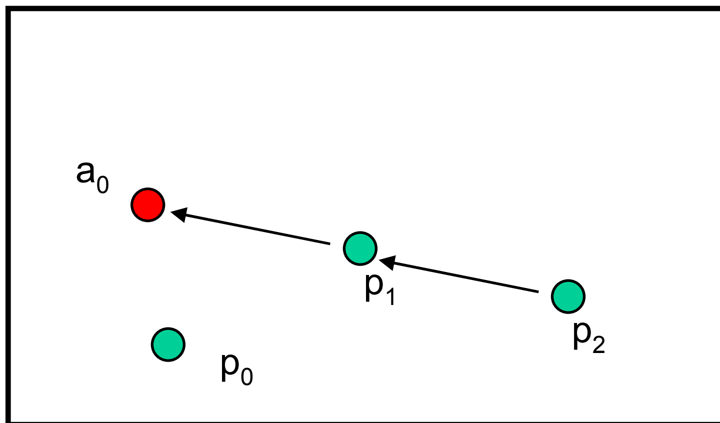
Chapter 2

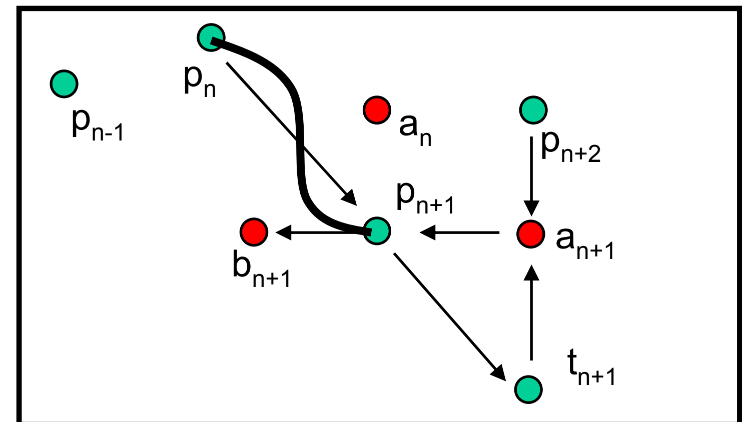
# Bezier interpolation

Construct interior control points

At the end points, we do not have the previous control points: $a_0 = p_1 + (p_1 - p_2)$

This then defines cubic Bezier segments consisting of the control points $p_n$, $a_n$ ,$b_{n+1}$, $p_{n+1}$.



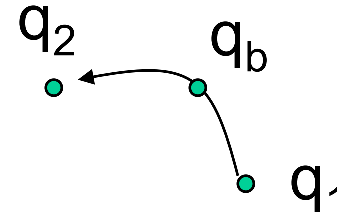



# Chapter 2

## Quaternion operators

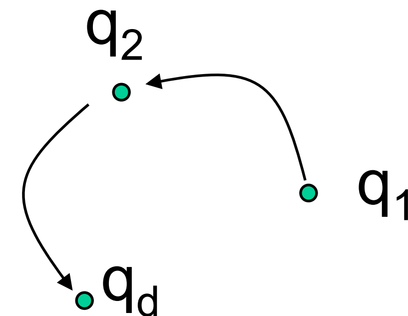
**bisect($q_1$, $q_2$)**

Similar to forming a vector between 2 points, form the rotation between 2 orientations



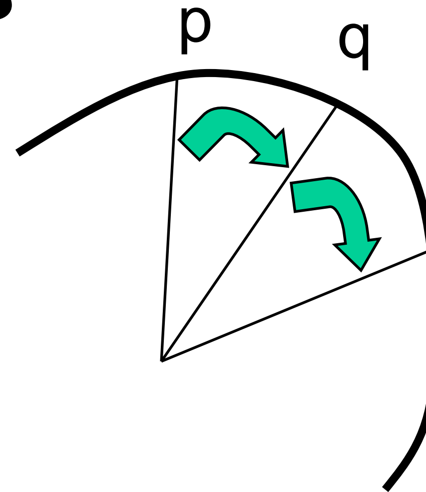

**double($q_1$, $q_2$)**

Given 2 orientations, form result of applying rotation between the two to 2nd orientation



# Chapter 2

## Quaternion operators

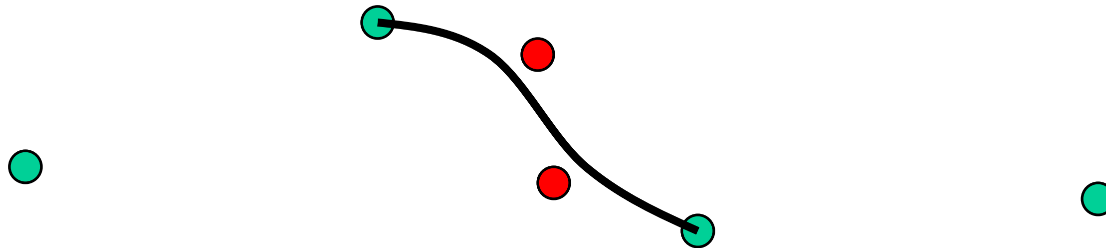



$$double(p,q) = 2(p \cdot q)q - p$$

Bisect 2 orientations: $\text{bisect}\ (q_1, q_2) = \dfrac{q_1 + q_2}{\|q_1 + q_2\|}$

Chapter 2

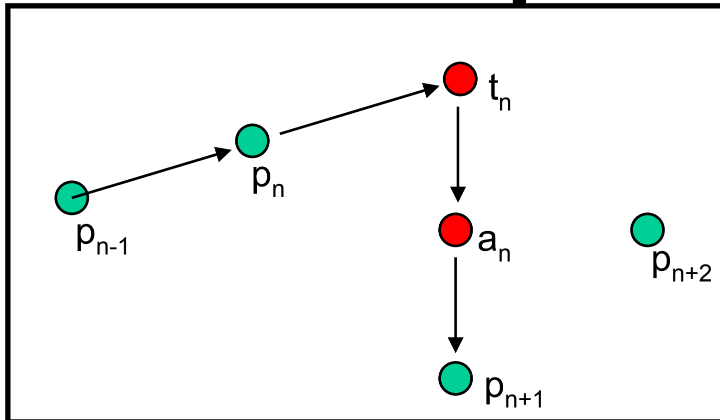
# Bezier interpolation



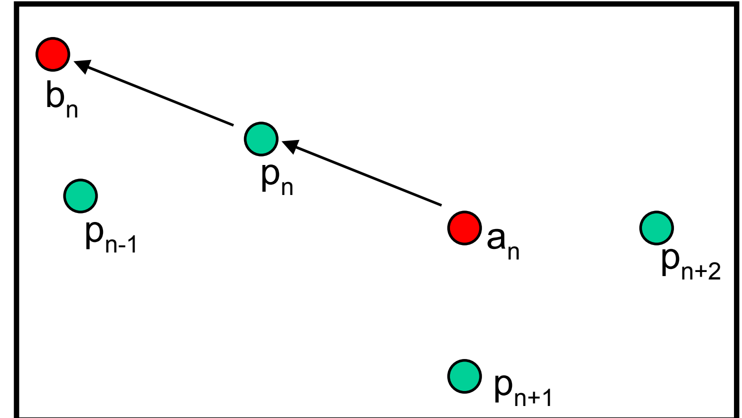
**Need quaternion-friendly operators to form interior control points**

# Chapter 2
## Bezier interpolation

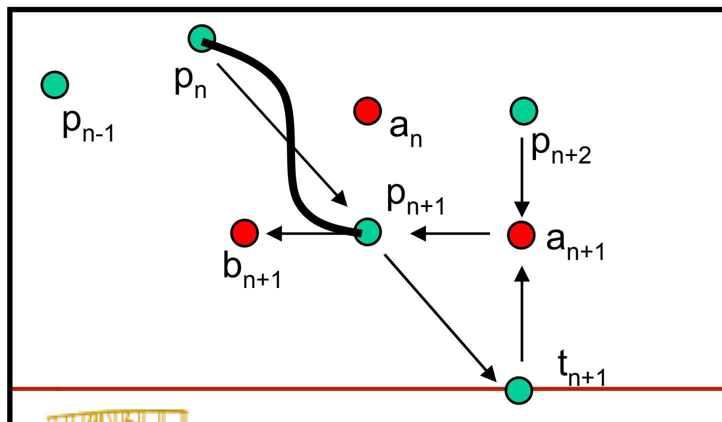
Construct interior control points




$$a_n = \text{bisect}(\text{double}(q_{n-1}, q_n), q_{n+1})$$




$$b_n = double(a_n, q_n)$$




Bezier segment:

$q_n, a_n, b_{n+1}, q_{n+1}$

Chapter 2

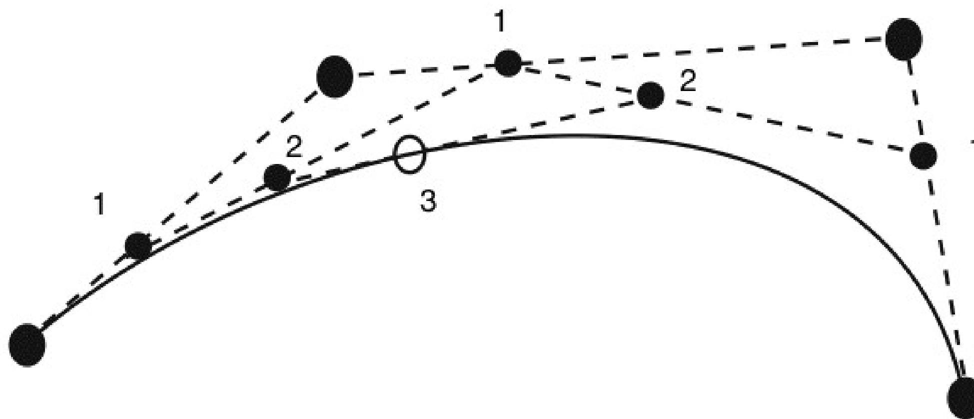
# Bezier construction using quaternion operators

**Need quaternion-friendly operations to interpolate cubic Bezier curve using ‘quaternion’ points**

**de Casteljau geometric construction algorithm**

Chapter 2

# Bezier construction using quaternion operators




$t_1$=slerp($q_n$, $a_n$,1/3)
$t_2$=slerp($a_n$, $b_{n+1}$,1/3)
$t_3$=slerp($b_{n+1}$, $q_{n+1}$,1/3)

$t_{12}$=slerp($t_1$, $t_2$,1/3)
$t_{23}$=slerp($t_{12}$, $t_{23}$,1/3)

q=slerp($t_{12}$, $t_{23}$,1/3)

Chapter 2

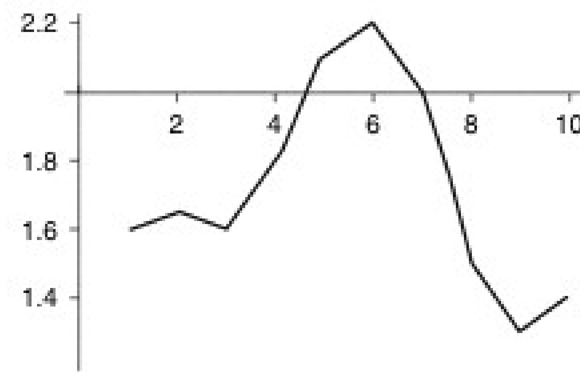
# Working with paths

For cases in which the points making up a path are generated by a digitizing process, the resulting curve can be too jerky because of noise or imprecision. To remove the jerkiness, the coordinate values of the data can be smoothed by one of several approaches:

**Smoothing a path**
**Determining a path along a surface**
**Finding downhill direction**

Chapter 2

# Smoothing data

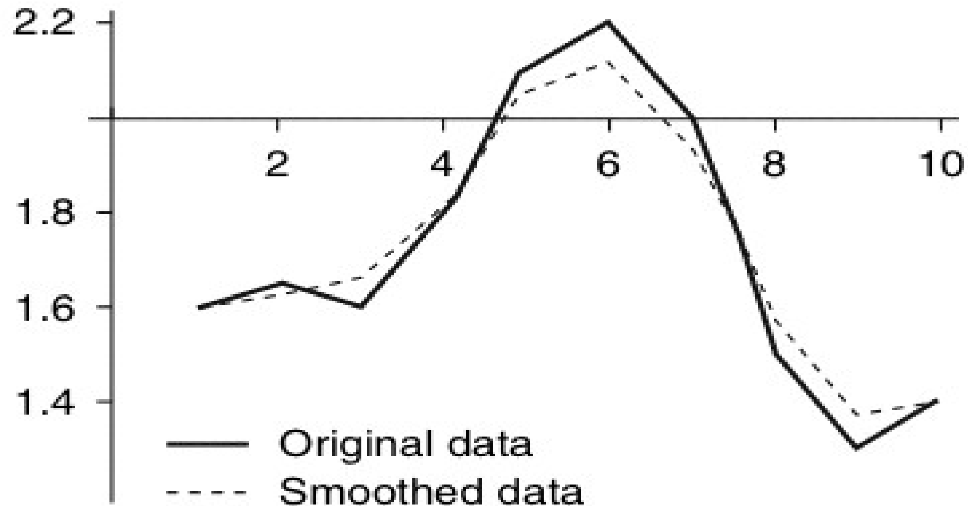



Averaging adjacent points:

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

Chapter 2

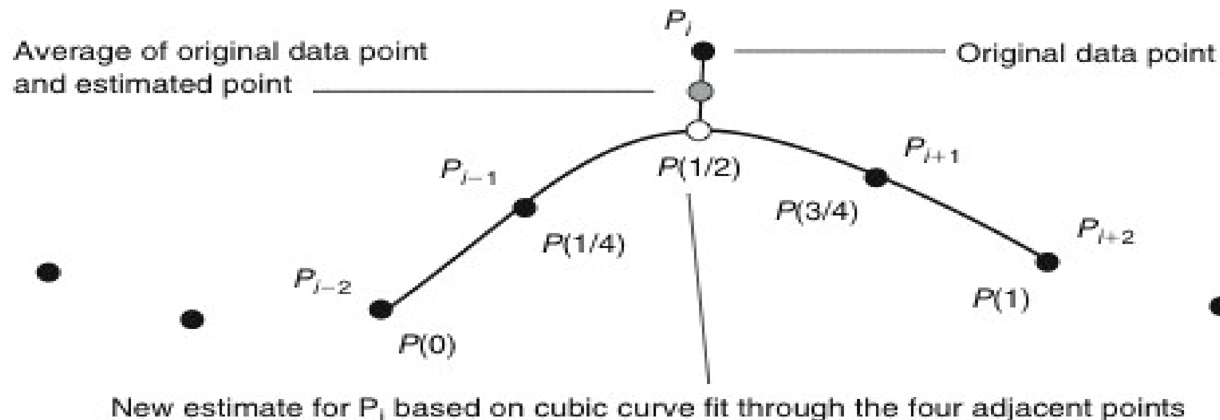
# Smoothing data (result)



# Chapter 2

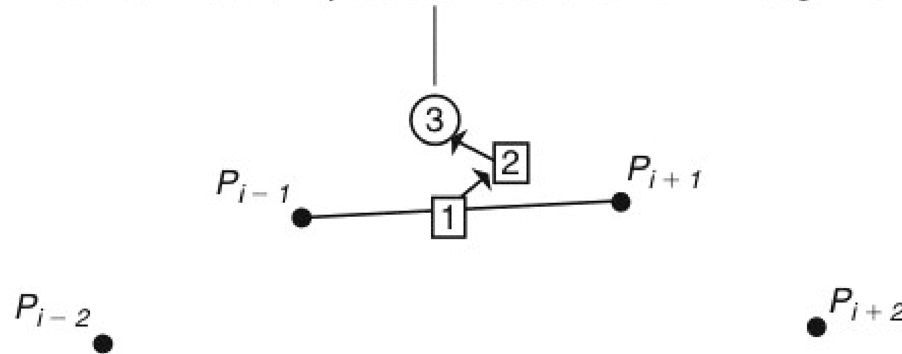
## Smoothing data

Use the surrounding four points to compute a cubic polynomial. Compute estimated point and average with actual point:



## Chapter 2

# Smoothing data

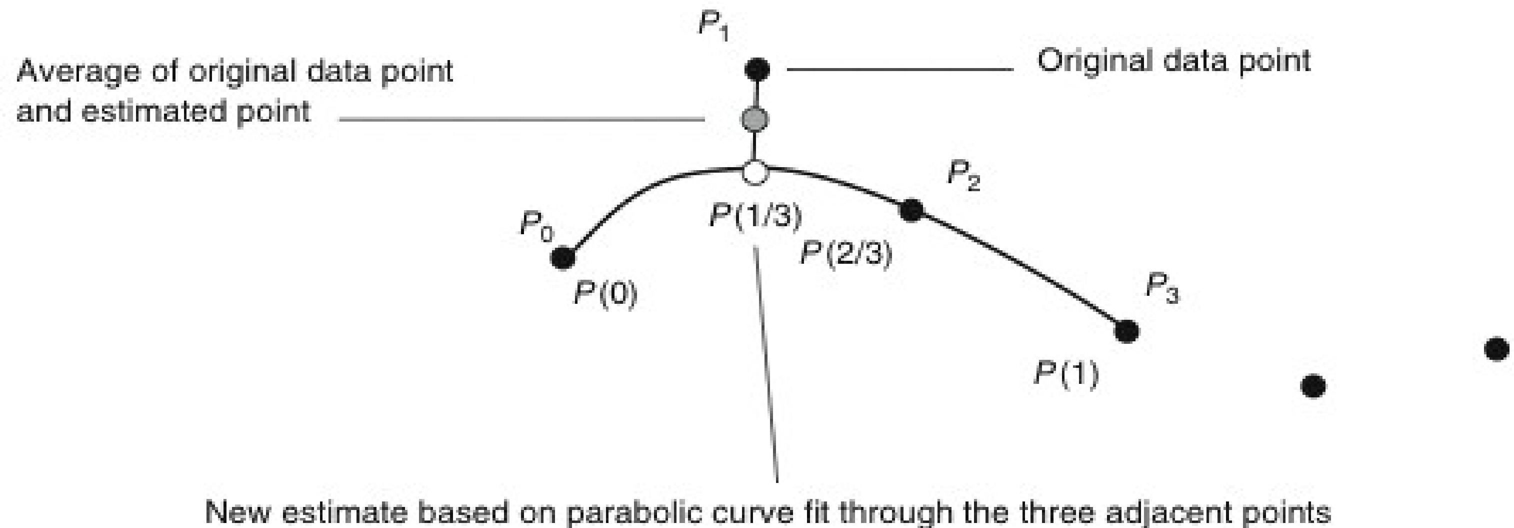



1. Average $P_{i-1}$ and $P_{i+1}$
2. Add *1/6* of the vector from $P_{i-2}$ to $P_{i-1}$
3. Add *1/6* of the vector from $P_{i+2}$ to $P_{i+1}$ to get the new estimated point

Chapter 2

# Smoothing data

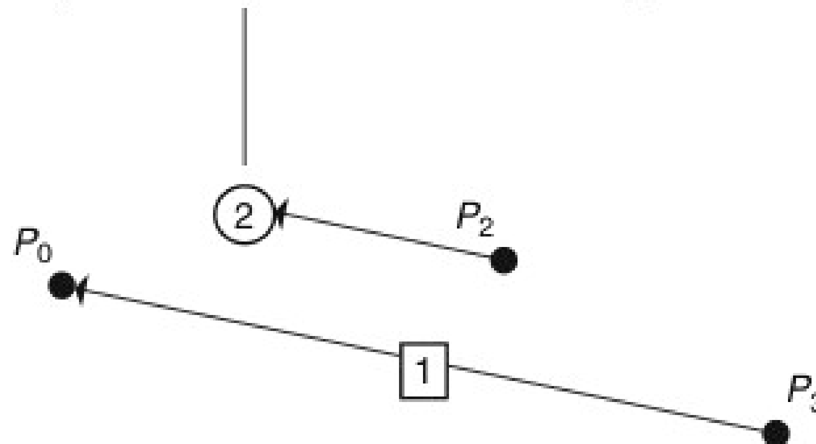
Average estimated point with original data point:



# Chapter 2

## Smoothing data

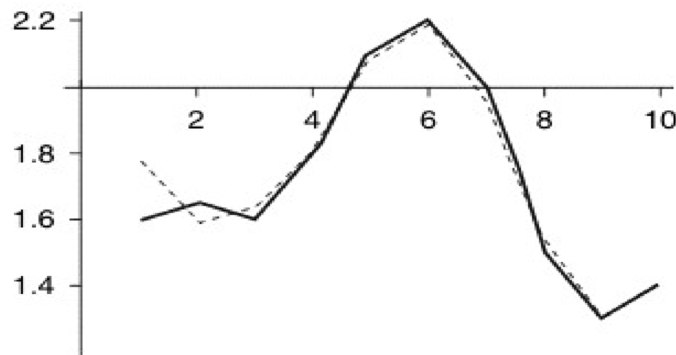
At the end points, there are not enough data points so that we have to use a quadratic curve to determine the estimated point: $P_1` = P_2 + (1/3)(P_0 - P_3)$

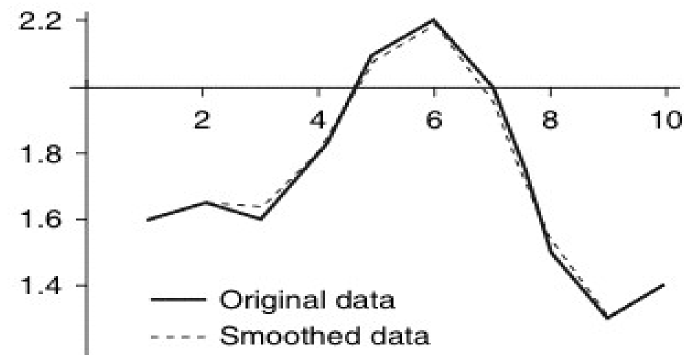


## Chapter 2

# Smoothing data

Similarly, the end points can be smoothed using parabolic interpolation: $P_0` = P_3 + 3(P_1 - P_2)$



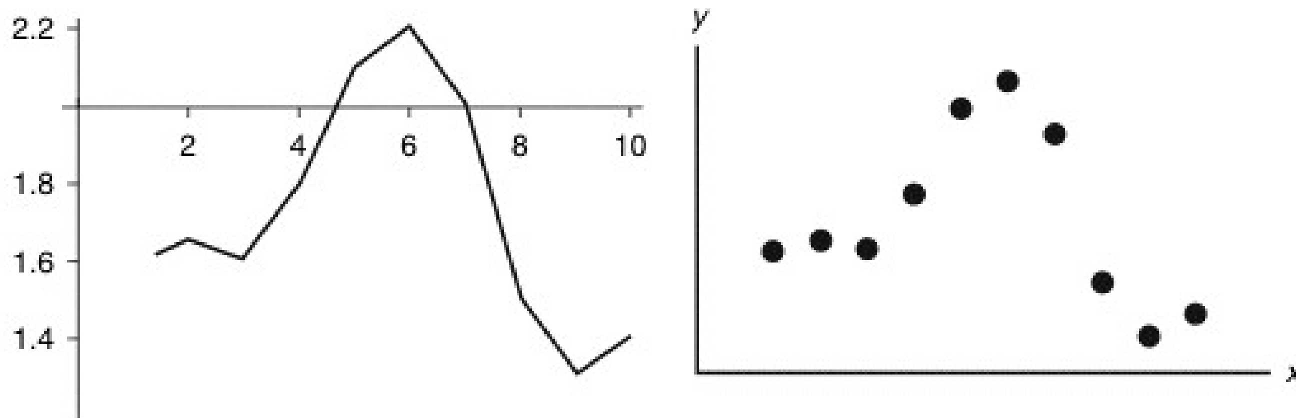

Cubic smoothing with parabolic end condition

Cubic smoothing without parabolic end condition

# Chapter 2

## Smoothing data

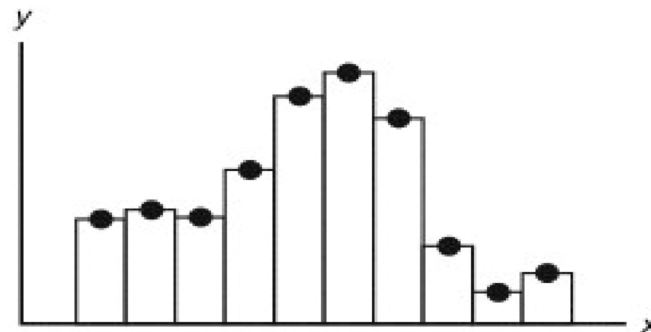
When the data to be smoothed can be viewed as a value of a function $y_i = f(x_i)$, the data can be smoothed by convolution:



Chapter 2

# Smoothing data

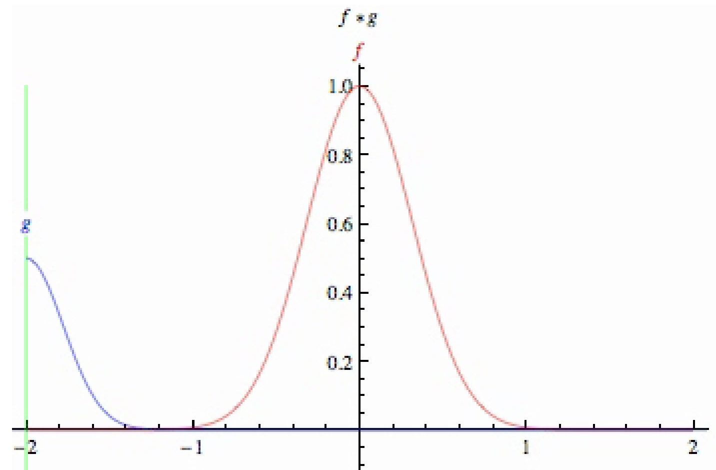
Thus, the data is interpreted as discrete data points comparable to a 1-D image:



# Chapter 2

Mathematically, convolution is an integral that expresses the amount of overlap of one function *g* as it is shifted over another function *f*. It therefore "blends" one function with another:

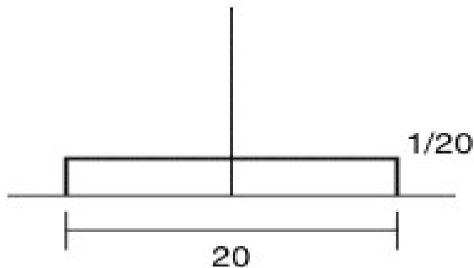
$$[f * g](t) = \int_0^t f(\tau) g(t-\tau) d\tau$$

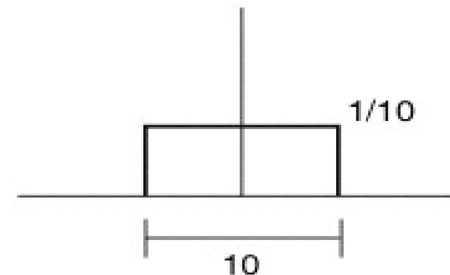


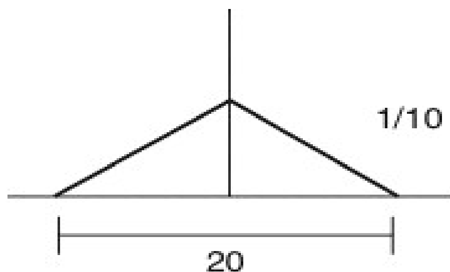
# Chapter 2

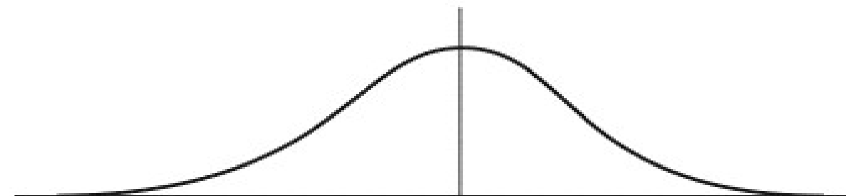
## Smoothing data

Different filter kernels can then be applied:



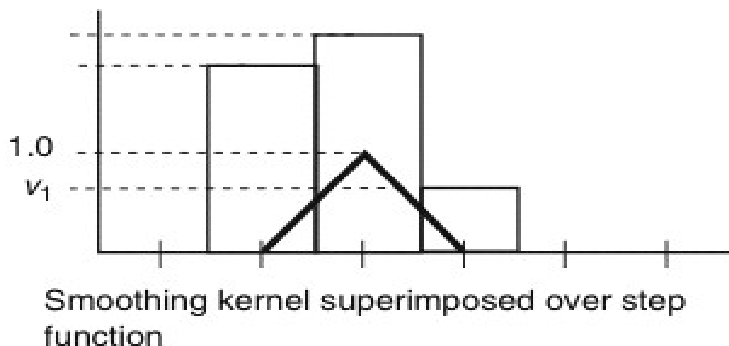



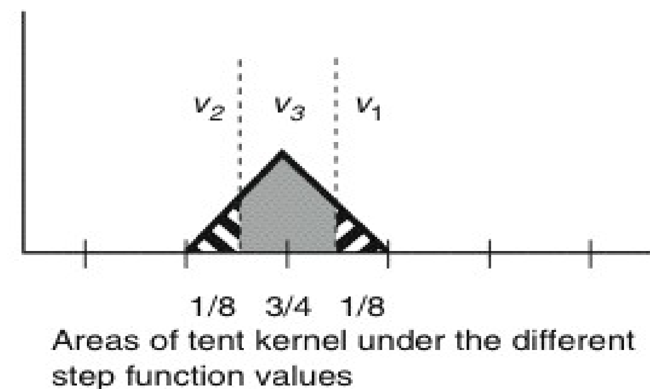
Gaussian $\frac{1}{a\sqrt{2\pi}} e^{-(x-b)^2/(2a^2)}$

Chapter 2

# Smoothing data




Smoothing kernel superimposed over step function



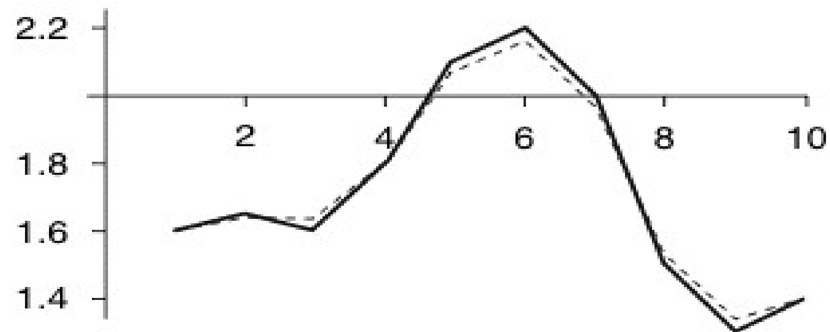

Areas of tent kernel under the different step function values

$$V = \frac{1}{8}v_1 + \frac{3}{4}v_2 + \frac{1}{8}v_3$$

Computation of value smoothed by applying area weights to step function values

# Chapter 2

## Smoothing data



# Chapter 2

## Path finding

If one object is to move across the surface of another object, then a path across the surface must be determined. If start and destination points are known, it can be computationally expensive to find the shortest path between the points. However, it is not often necessary to find the absolute shortest path. Various alternatives exist for determining suboptimal, yet more-or-less direct paths.

# Chapter 2

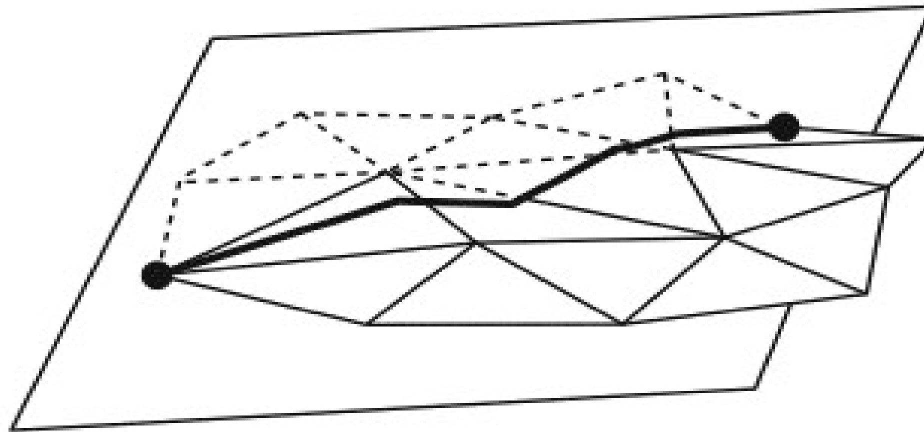
## Path finding

An easy way to determine a path along a polygonal surface mesh is to determine a plane that contains the two points and is generally perpendicular t the surface.

*Generally perpendicular* can be defined, for example, as the average of the two vertex normals that the path is being formed between. The intersection of the plane with the faces making up the surface mesh will define a path between the two points.

Chapter 2

# Path finding



This approach results in a reasonable path which is not necessarily the shortest path. Example: sphere.

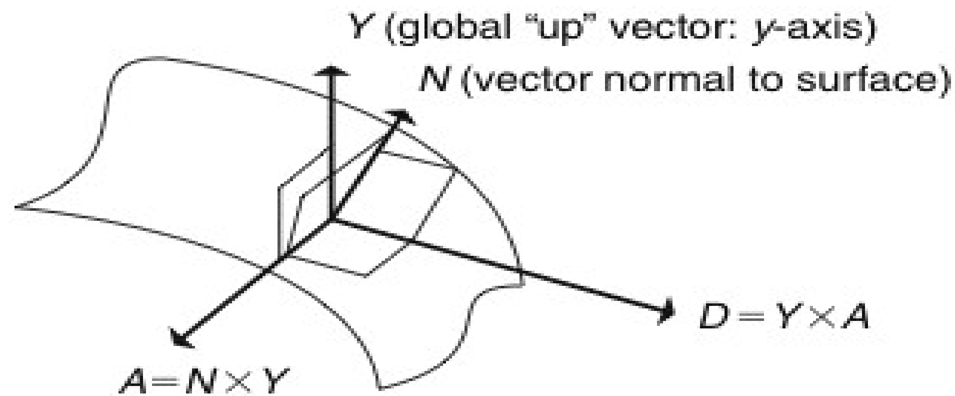
Chapter 2

# Path finding – downhill

If a path downhill from an initial point on the surface is desired, then the surface normal and global up vector can be used to determine the downhill vector. The cross product of the normal and global up vector defines a vector that lies on the surface perpendicular to the downhill direction. So the cross product of this vector and the normal vector defines the downhill (and uphill) vector on a plane. This same approach works with curved surfaces to produce the instantaneous downhill vector.

# Chapter 2

## Path finding - downhill



# Chapter 2

## Interpolation-Based Animation

The techniques described so far describe the basics of interpolating values. The remainder of this chapter addresses how to use those basics to facilitate the production of computer animation. Procedures and algorithms are used in which the animator has very specific expectations about the motion that will be produced on a frame-by-frame basis.

## Chapter 2

# Interpolation based animation

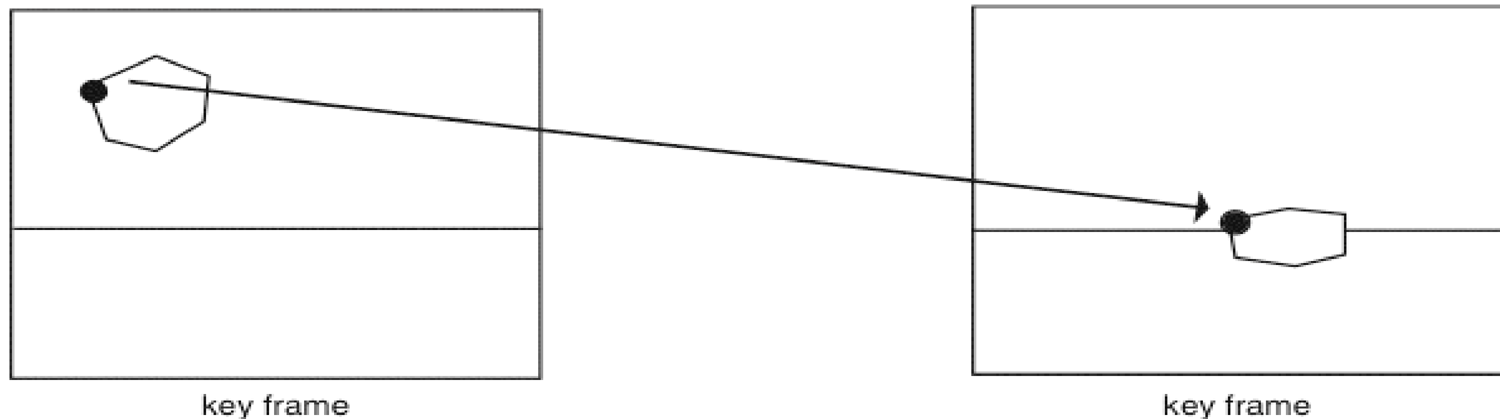
**Key-frame systems – in general**

**Interpolating shapes**

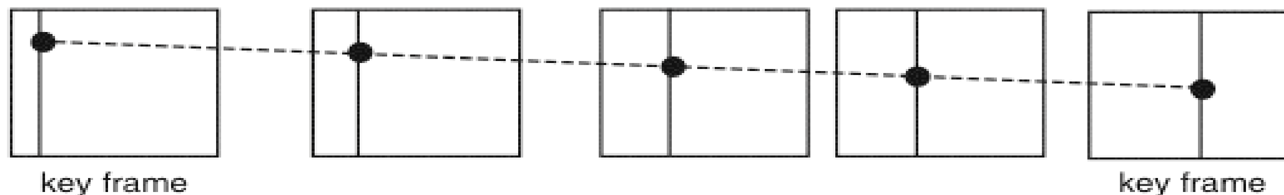
- **Deforming a single shape**
- **3D interpolation between two shapes**
- **Morphing – deforming an image**

Chapter 2

# Keyframing – interpolating values




Simple key frames in which each curve of a frame has the same number of points as its counterpart in the other frame

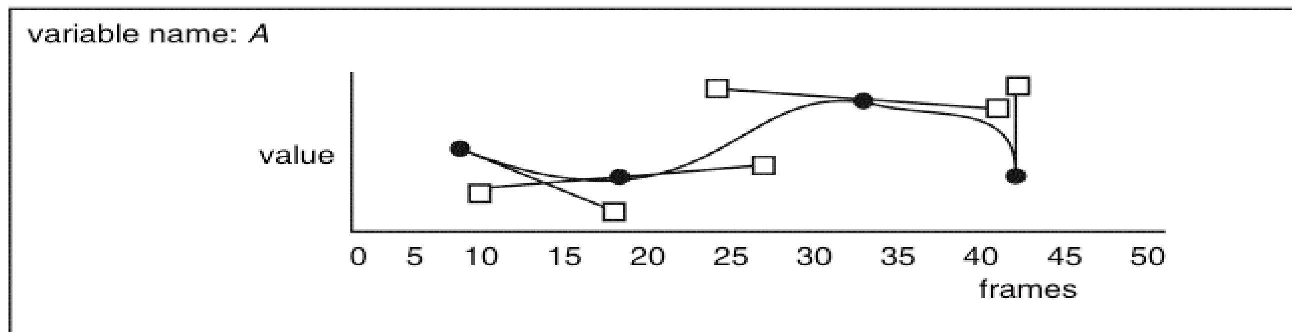



Keys and three intermediate frames with linear interpolation of a single point (with reference showing the progression of the interpolation in *x* and *y*)

# Chapter 2

## Keyframing

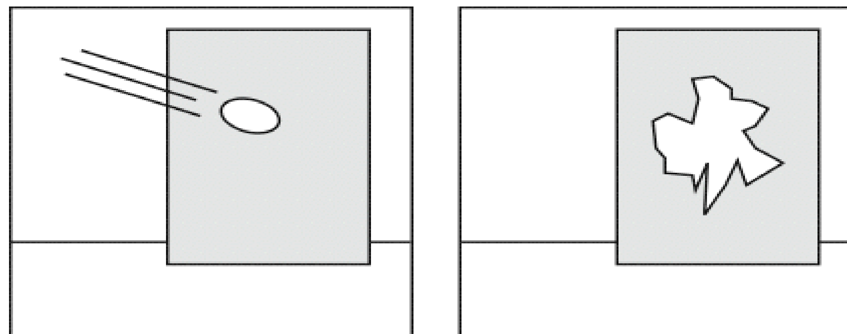
In computer animation, the term key frame has been generalized to apply to any variable whose value is set at specific *key frames* and from which values for the intermediate frames are interpolated according to some prescribed procedure (interpolation scheme can include points, tangents, …). These variables have been referred to as *articulation variables (avars)* and the systems to *track based*.



# Chapter 2

## Keyframing curves

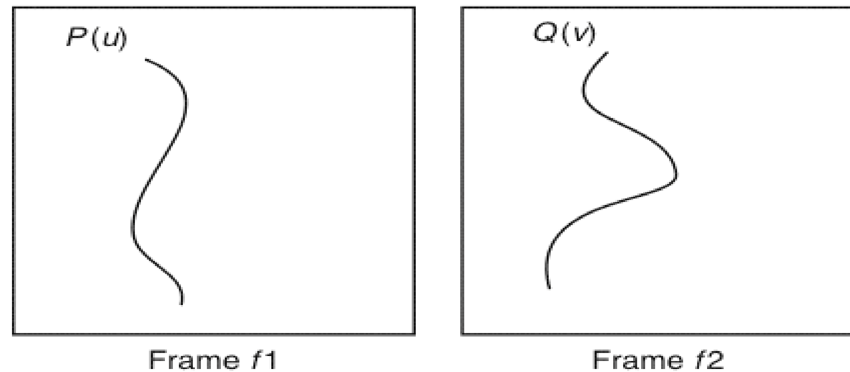
Point-by-point correspondence information is usually not known, and even if it is, the resulting interpolation is not necessarily what the user wants. The best one can expect is for the curve-to-curve correspondence to be given. The problem is, given two arbitrary curves in key frames, to interpolate a curve as it "should" appear in intermediate frames. For example, observe the egg splatting against the wall:



# Chapter 2

## Time-Curve interpolation

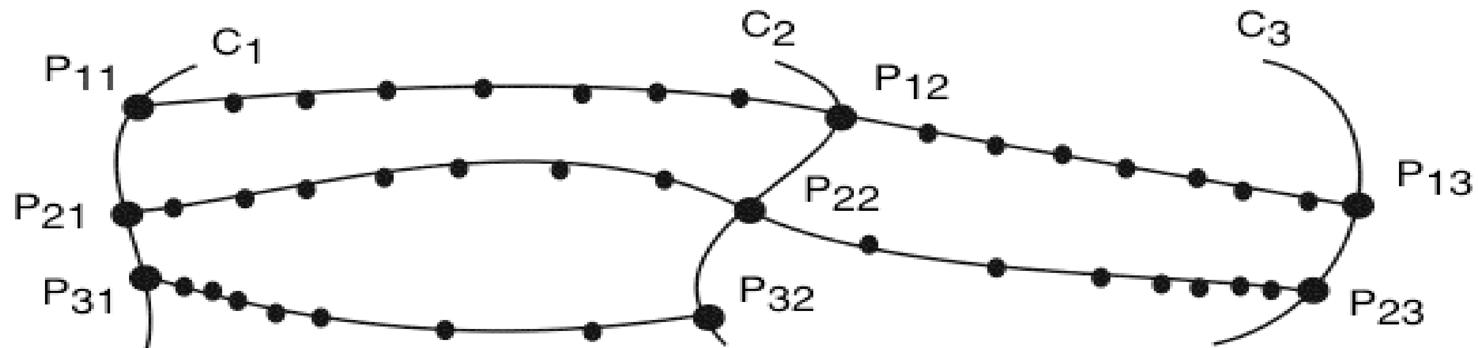
**Implement using surface patch technology**



# Chapter 2

## Time-Curve interpolation

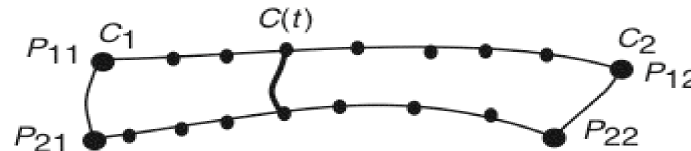
### Establish point correspondence



# Chapter 2

## Time-Curve interpolation

Define time – space-curve “patches”




**Interpolate in one dimension for curve (spatially)**
**Interpolate in other dimension temporally**

Chapter 2

# Object interpolation

Correspondence problem
Interpolation problem

1. Modify shape of object interpolate vertices of different shapes

2. Interpolate one object into second object

3. Interpolate one image into second image

# Chapter 2

## Deforming Objects

Deforming an object shape and transforming one shape into another is a visually powerful animation technique. It adds the notion of malleability and density. Flexible body animation makes the objects in an animation seem much more expressive and alive. There are physically based approaches that simulate the reaction of objects undergoing forces. However, many animators want more precise control over the shape on an object than that provided by simulations and/or do not want the computational expense of the simulating physical process.

# Chapter 2

## Deforming Objects

Instead, the animator may want to deform the object directly and define key shapes. Shape definitions that share the same edge connectivity can be interpolated on a vertex-to-vertex basis in order to smoothly change from one shape to the other. A sequence of key shapes can be interpolated over time to produce flexible body animation. Multivariate interpolation can be used to blend among a number of different shapes. The various shapes are referred to as **blend shapes** or **morph targets** and is a commonly used technique in facial animation.

Chapter 2

# Object Modification

Different techniques are available

Modify the vertices directly → Vertex warping

OR

Modify the space the vertices lie in →
- 2D grid-based deforming
- Skeletal bending
- Global transforms
- Free Form Deformations

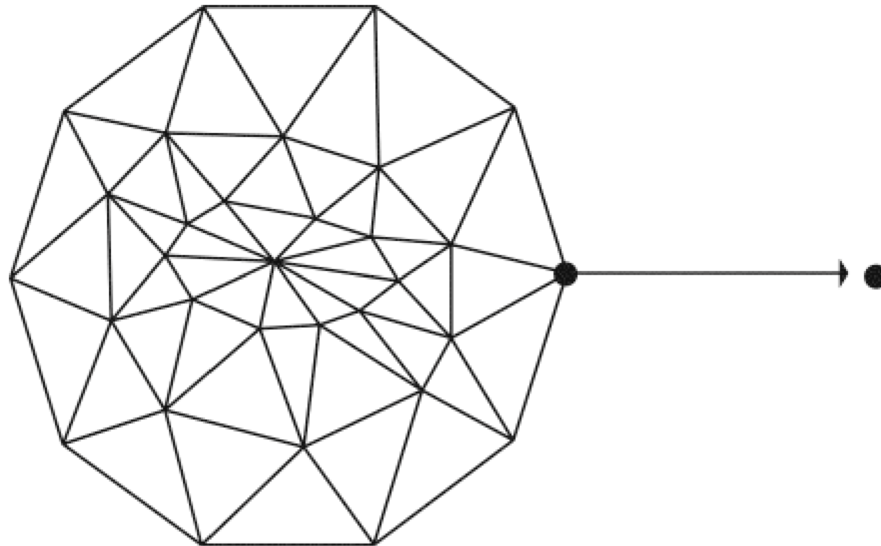
# Chapter 2

## Warping

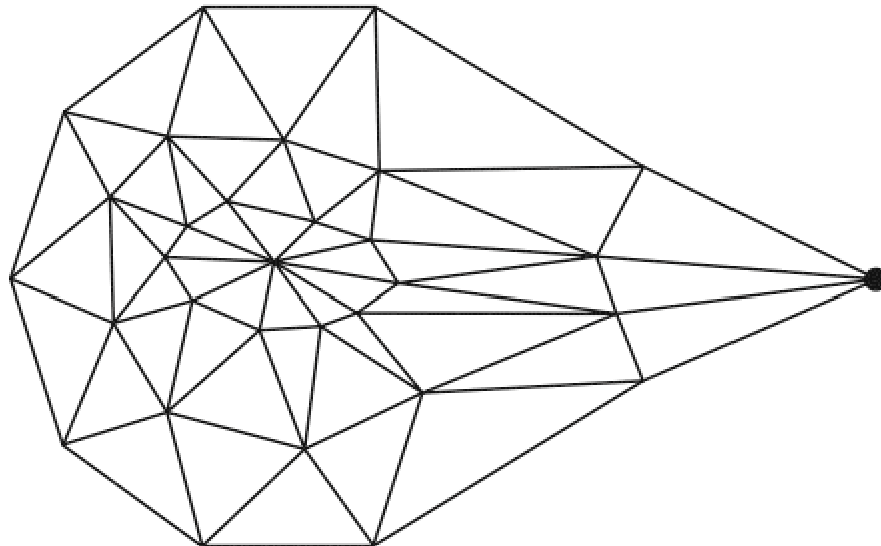
A particularly simple way to modify the shape of an object is to displace one or more of its vertices. To do this on a per-vertex basis can be tedious for a large number of vertices. Simply grouping a number of vertices together and displacing them uniformly can be effective in modifying the shape of an object but is too restrictive in the shapes that can easily be created. An effective improvement is to allow the user to displace a vertex (the seed vertex) or group of vertices of the object and propagate the displacement to adjacent vertices along the surface while attenuating the amount of displacement.

# Warping

Displacement can be attenuated as a function of distance between seed vertex and vertex to be displace.



Displacement of seed vertex



Attenuated displacement propagated to adjacent vertices

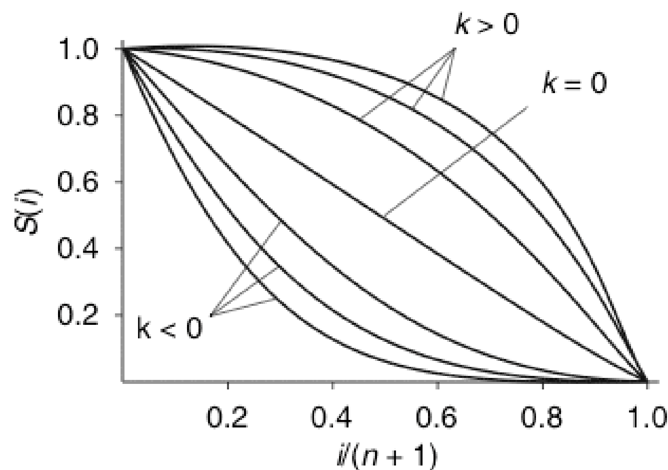
# Chapter 2

## Warping

Attenuation is typically a function of the distance metric. The minimum of connecting edges is used for the distance metric and the user specifies the maximum range of effect to be vertices within $n$ edges of the seed vertex. A scale factor is applied to the displacement vector according to the user-selected integer value of $k$ as shown on the next slide.

Chapter 2

# Power functions
# For attenuating warping effects




$$S(i) = 1.0 - \left(\frac{i}{n+1}\right)^{k+1} \qquad k \geq 0$$

$$S(i) = \left(1.0 - \frac{i}{n+1}\right)^{-k+1} \qquad k < 0$$

# Chapter 2

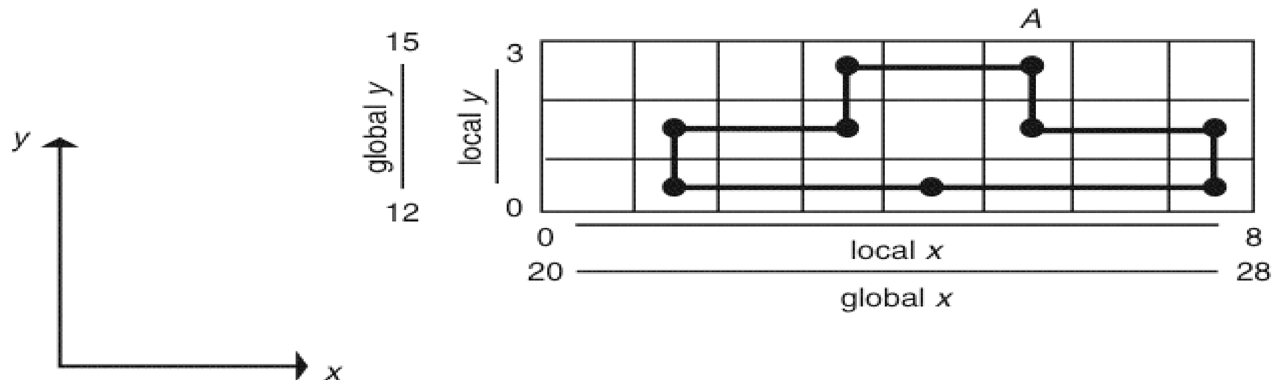
## Warping

These attenuation functions are easy to compute and provide sufficient flexibility for many desired effects. When $k$ equals zero it corresponds to a linear attenuation, while values of $k$ less than zero create a more elastic impression. Values of $k$ greater than zero create the effect of more rigid displacements.

# Chapter 2

## 2D grid-based deforming

The local coordinate system is a two-dimensional grid in which an object is placed. The grid is initially aligned with the global axes so that the mapping from local to global coordinates consists of a scale and a translate.

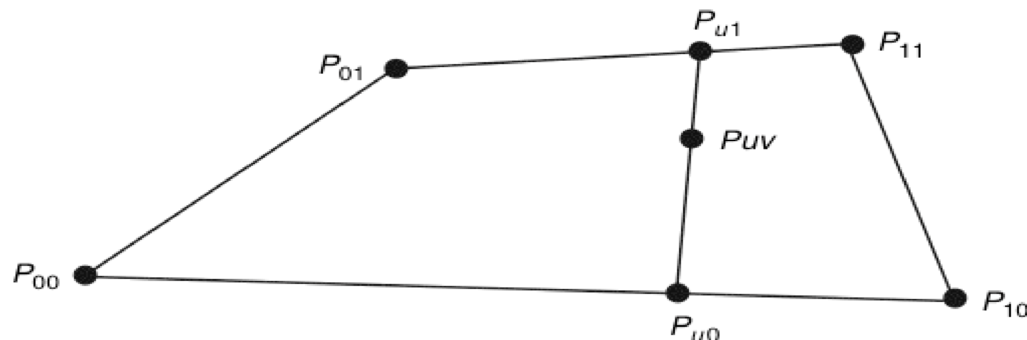


# Chapter 2

## 2D grid-based deforming

The grid is then distorted by the user moving the vertices of the grid so that the local space is distorted. The vertices of the object are then relocated in the distorted grid by bilinear interpolation relative to the cell of the grid in which the vertex is located.

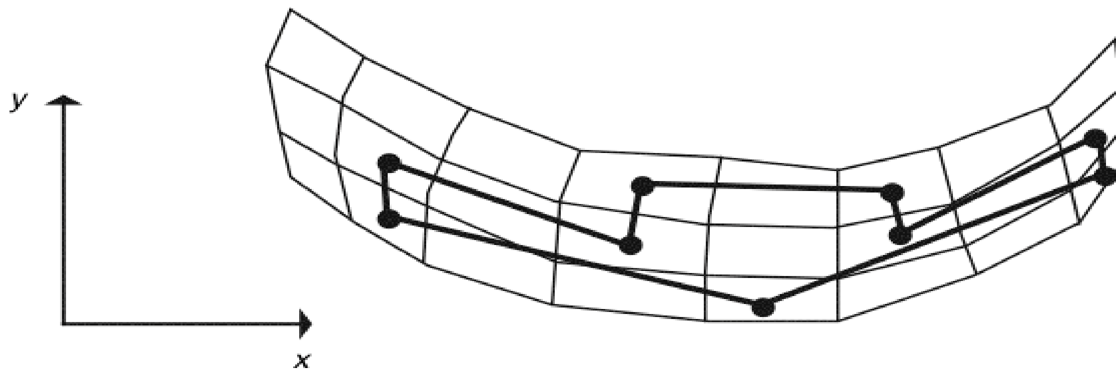
$$
\begin{aligned}
P_{u0} &= (1-u)P_{00} + uP_{10} \\
P_{u1} &= (1-u)P_{01} + uP_{11} \\
P_{uv} &= (1-v)P_{u0} + vP_{u1} \\
&= (1-u)(1-v)P_{00} + (1-v)uP_{01} + u(1-v)P_{10} + uvP_{11}
\end{aligned}
$$



Chapter 2

# 2D grid-based deforming

Once this is done for all vertices of the object, the object is distorted according to the distortion of the local grid. For the objects that contain hundreds of thousands of vertices, the grid distortion is much more efficient than individually repositioning each vertex. In addition, it is more intuitive for the user to specify a deformation.



# Chapter 2

## Polyline Deformation

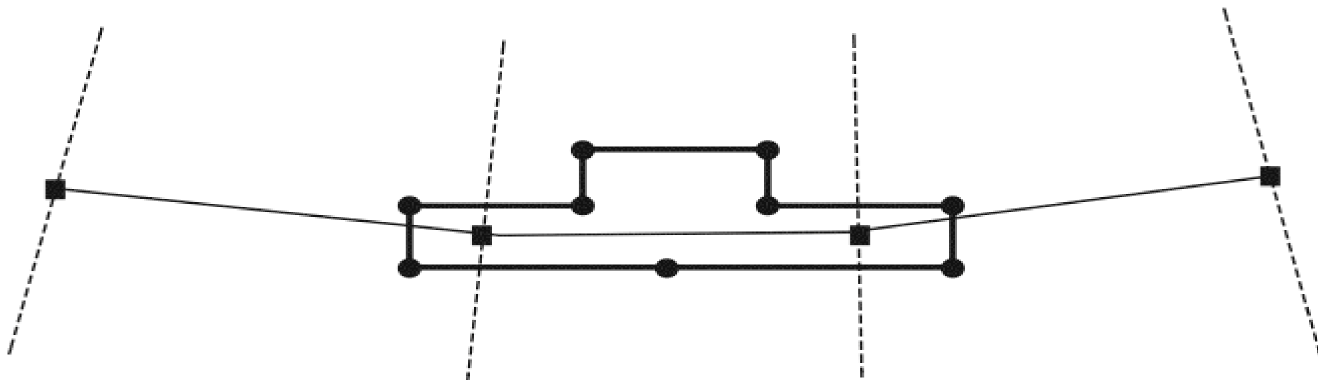
Polyline deformation is similar to the grid approach in that the object vertices are mapped to the polyline, the polyline is then modified by the user, and the object vertices are then mapped to the same relative location on the polyline.

The mapping to the polyline is performed by first locating the most relevant line segment for each object vertex. To do this, intersecting lines are formed at the junction of adjacent segments, and perpendicular lines are formed at the extreme ends of the polyline. These lines will be referred to as **boundary lines**; each polyline segment has two boundary lines. For each object, the closest polyline segment that contains the object vertex between its boundary lines is selected.

Chapter 2

# 2D Polyline Deformation

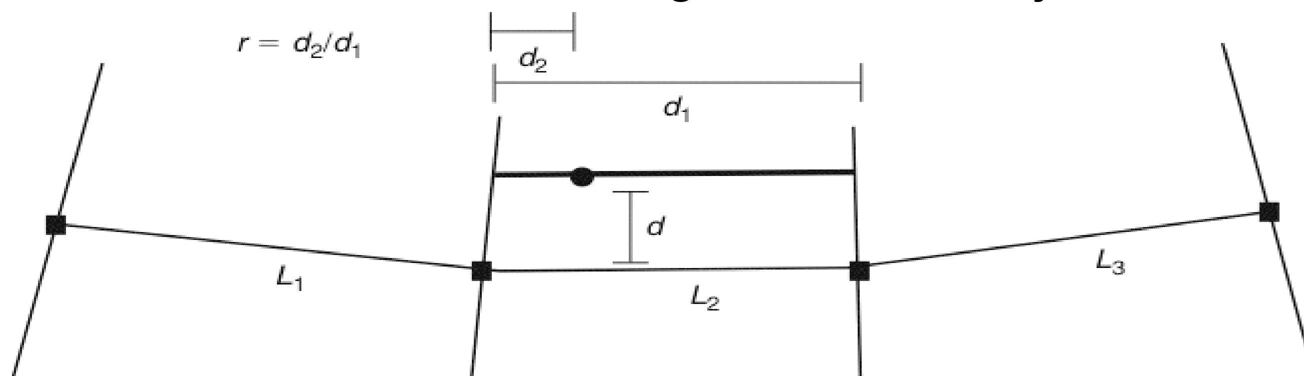
Polyline drawn through object; bisectors and perpendiculars are drawn as dashed lines.



# Chapter 2

## 2D skeleton-based bending

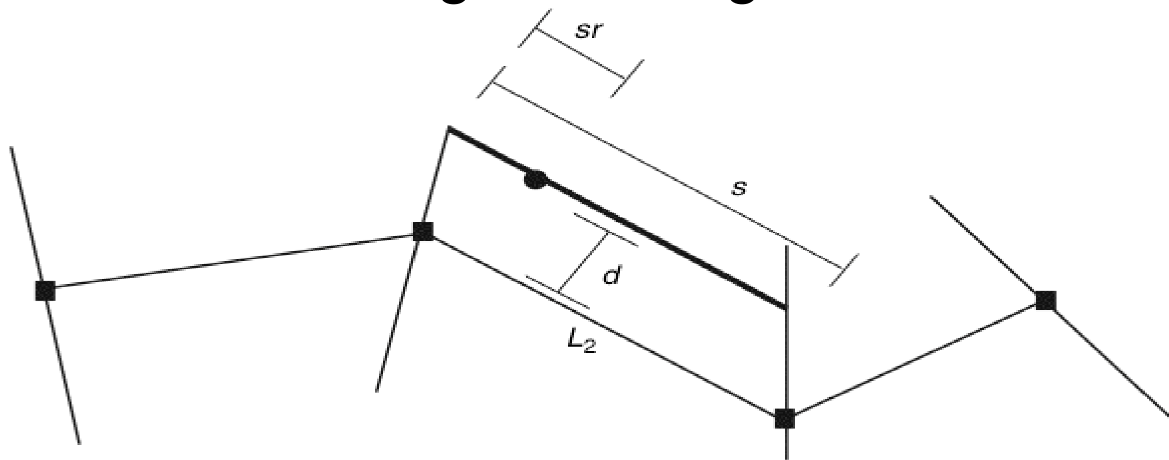
Next, each object is mapped to its corresponding polyline segment. A line segment is constructed through the object vertex parallel to the polyline segment and between the boundary lines. For a given object vertex, the following information is recorded: the closes line segment ($L_2$); the line segment's distance to the polyline segment ($d$); and the objects vertex's relative position on this line segment, that is, the ratio r of the length of the line segment ($d_1$) and the distance from one end of the line segment to the object vertex ($d_2$).



# Chapter 2

## 2D skeleton-based bending

The polyline is hen repositioned by the user and each object vertex is repositioned relative to the polyline using the information previously recorded for that vertex. A line parallel to the newly positioned segment is constructed $d$ units away and the vertex's new position is the same fraction along this line that it was in the original configuration



# Chapter 2

## Global Transformations

The idea is to globally deform the space in which an object is defined by applying a 3x3 transformation matrix, $M$, which is a function of the point being transformed:

$$p' = Mp$$

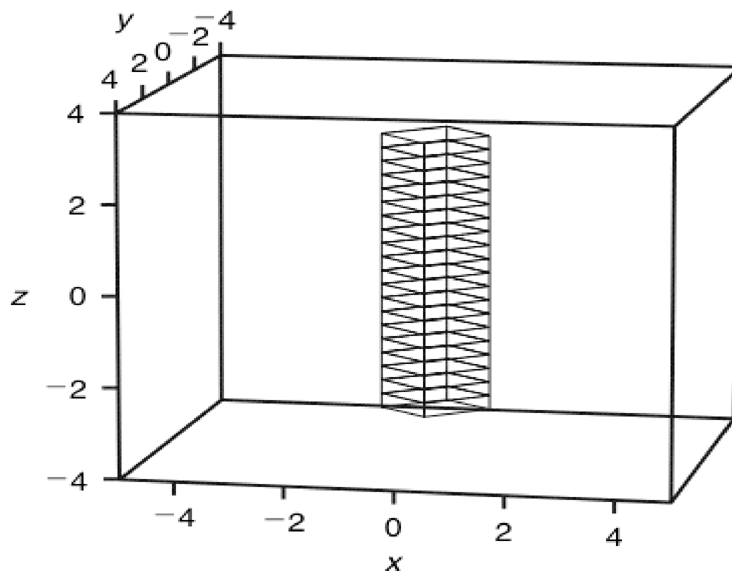
In GT, the transformation is a function of where you are in space, i.e. it depends on the point $p$:

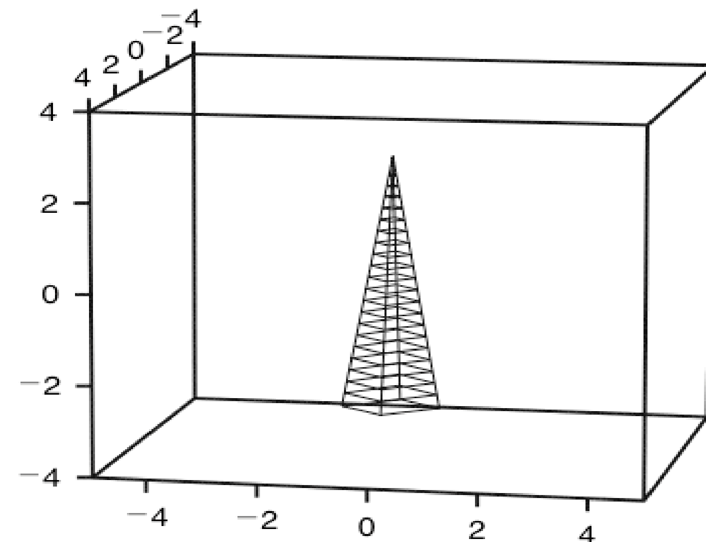
$$p' = M(p)p$$

# Chapter 2

## Global Transformations

A simple linear 2D tapering operation:




$$s(z) = \frac{(maxz - z)}{(maxz - minz)}$$

$$x' = s(z)x$$
$$y' = s(z)y$$
$$z' = z$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s(z) & 0 & 0 \\ 0 & s(z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

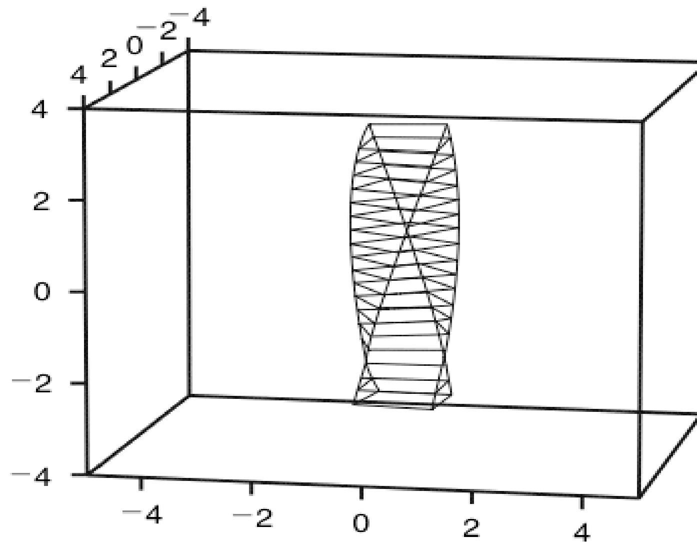
$$P' = M(p)p$$

Original object Tapered object

Chapter 2

# Global Transformations

A twist about an axis:




$$k = \text{twist factor}$$
$$x' = x\cos(kz) - y\sin(kz)$$
$$y' = x\sin(kz) + y\cos(kz)$$
$$z' = z$$

Chapter 2

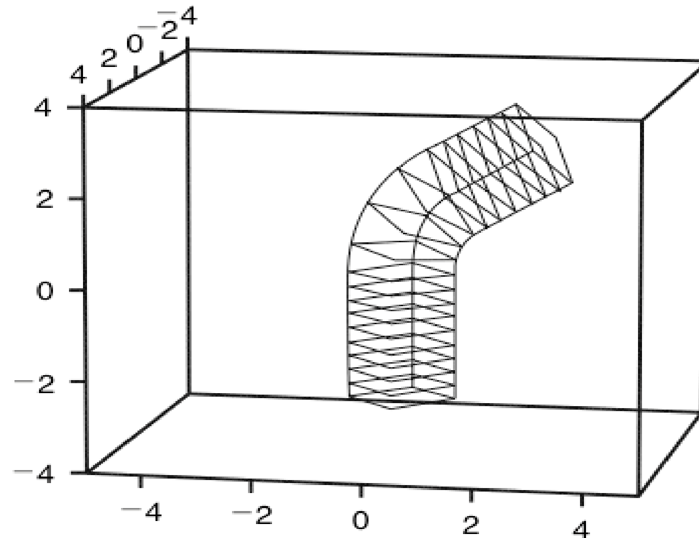
# Global Transformations

A global bend operation:

z above $z_{min}$: rotate Q

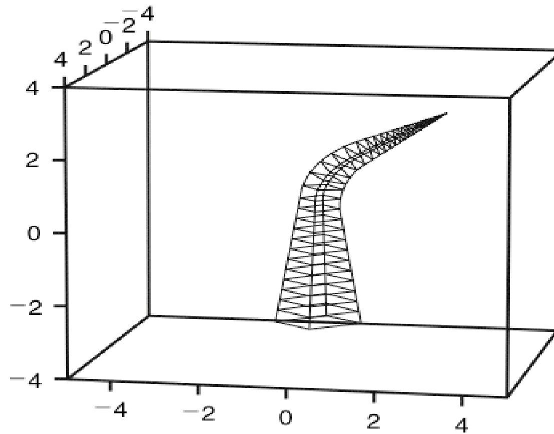
z between $z_{min}$ ,$z_{ma}$x :
Rotate from 0 to Q

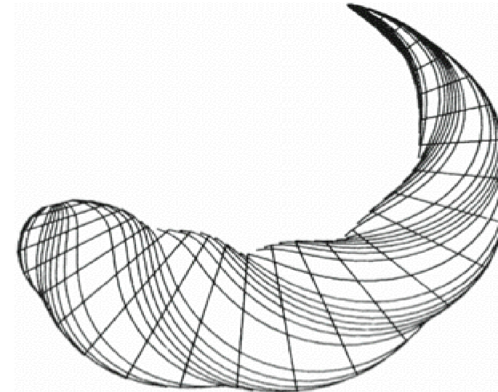
z below $z_{min}$: no rotation

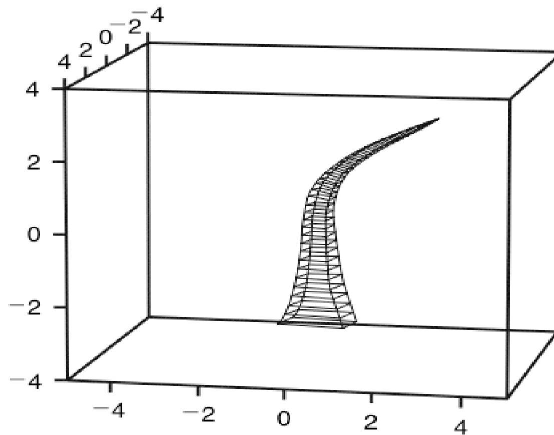


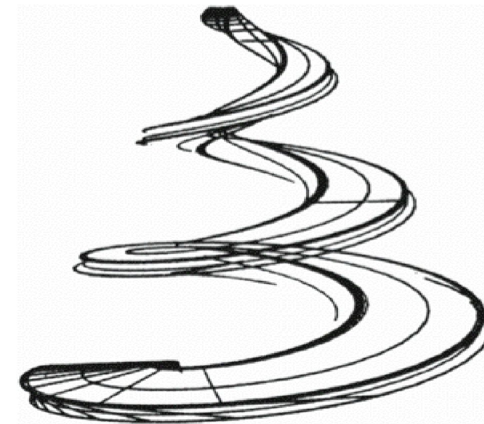
# Chapter 2

## Compound global transformations





Chapter 2

# Free-Form Deformations (FFDs)

| **2D grid-based deforming** | **FFDs** |
| --- | --- |
| 2D grid | 3D grid |
| bi-linear interpolation | tri-cubic interpolation |

# Chapter 2

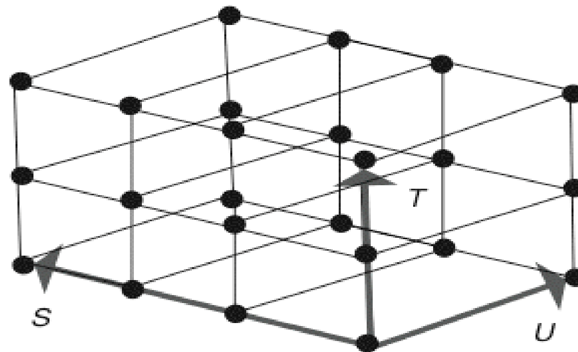
## Free-Form Deformation

A Free-Form Deformation (FFD) is essentially a three-dimensional extension of Burtnyk's grid deformation that incorporates higher-order interpolation. In both cases, a localized coordinate grid, in a standard configuration, is superimposed over an object. For each vertex of the object, coordinates relative to this local grid are determined that register the vertex to the grid. The grid is then manipulated by the user. Using its relative coordinates, each vertex is then mapped back into the modified grid, which relocates that vertex in global space. Instead of linear interpolation, cubic interpolation is typically used with FFDs, e.g. Bezier interpolation.

Chapter 2

# Free-Form Deformations

As with the 2D grid deformation, the object is embedded in a rectilinear grid (which can be composed of an unequal number of points in the three directions:



## Chapter 2

# Free-Form Deformations

In the first step of the FFD, vertices of an object are located in a 3D the rectilinear grid. Initially, the local coordinate system is defined by a not-necessarily orthogonal set of three vectors $(S, T, U)$. A vertex $P$ is registered in the local coordinate system by determining its tri-linear interpolation as shown on the next slide.

# Chapter 2

## Free-Form Deformations

$$s = (T \times U) \cdot (P - P_0) / ((T \times U) \cdot S)$$

$$t = (U \times S) \cdot (P - P_0) / ((U \times S) \cdot T)$$

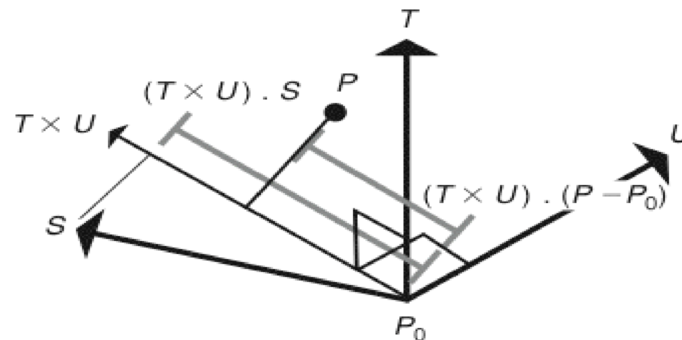
$$u = (S \times T) \cdot (P - P_0) / ((S \times T) \cdot U)$$

In the equations above, the cross product of two vectors forms a third vector that is orthogonal to the first two The denominator normalized the value being computed. In the first equation, for example, the projection of $S$ onto $T \times U$ determines the distance within which points will map into the range $0<s<1$.

# Chapter 2

## Free-Form Deformations

Register points in grid: cell x,y,z ↔ (s,t,u)



# Chapter 2

## Free-Form Deformations

Given the local coordinates (s, t, u) of a point and the unmodified local coordinate grid, a point's position can be reconstructed in global space by simply moving in the direction of the local coordinate axes according to its local coordinates:

$$P = P_0 + sS + tT + uU$$

# Chapter 2

## Free-Form Deformations

To facilitate the modification of the local coordinate system, the grid of control points is then used. If there are $l$ points in the $S$ direction, $m$ points in the $T$ direction, and $n$ points in the $U$ direction, the control points are located according to the following equation:

$$P_{ijk} = P_o + \frac{i}{l}S + \frac{j}{m}T + \frac{k}{n}U \quad \text{for} \quad \begin{matrix} 0 \le i \le l \\ 0 \le j \le m \\ 0 \le k \le n \end{matrix}$$

## Chapter 2

# Free-Form Deformations

The deformations are specified by moving the control points from their initial positions. The function that effects the deformation is a tri-variate Bezier interpolating function. The deformed position of a point $P_{stu}$ is determined by using its $(s, t, u)$ local coordinates in the following Bezier interpolating function, where $P(s, t, u)$ represents the global coordinates of the deformed point, and $P_{ijk}$ the global coordinates of the control points:

$$P(s,t,u) = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^i \left( \sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^j \left( \sum_{k=0}^{n} \binom{k}{k} (1-u)^{n-k} u^k P_{ijk} \right) \right)$$

Chapter 2

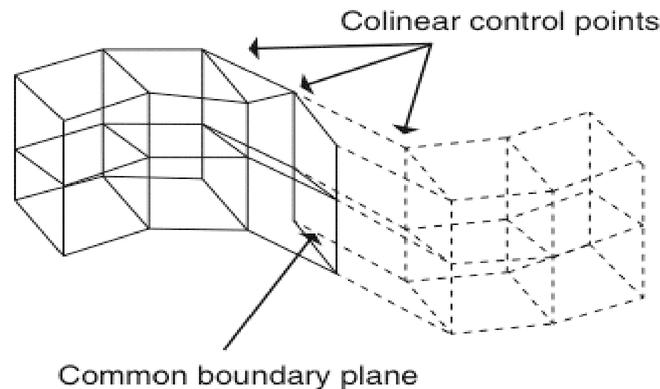
# Free-Form Deformations

Show sample video.

Chapter 2

# Free-Form Deformations

As in Bezier curve interpolation, multiple Bezier solids can e joined. The continuity is controlled by co-planarity of control points:

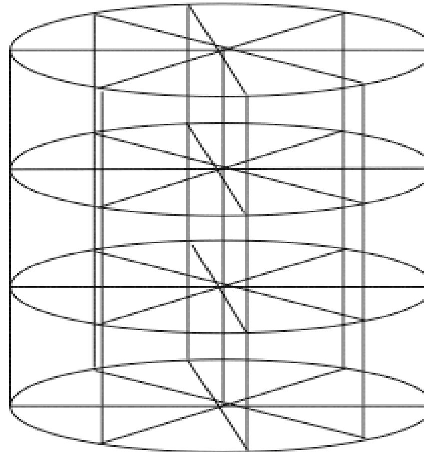



Higher-order continuity can be maintained similar to curves and surfaces as well.

Chapter 2

# FFDs: alternate grid organizations

FFDs have been extended to include initial grids that are something other than a parallelepiped. For example, a cylindrical lattice can be formed from the standard parallelepiped by merging the opposite boundary planes in one direction and then merging all the points along the cylindrical axis:



# Chapter 2

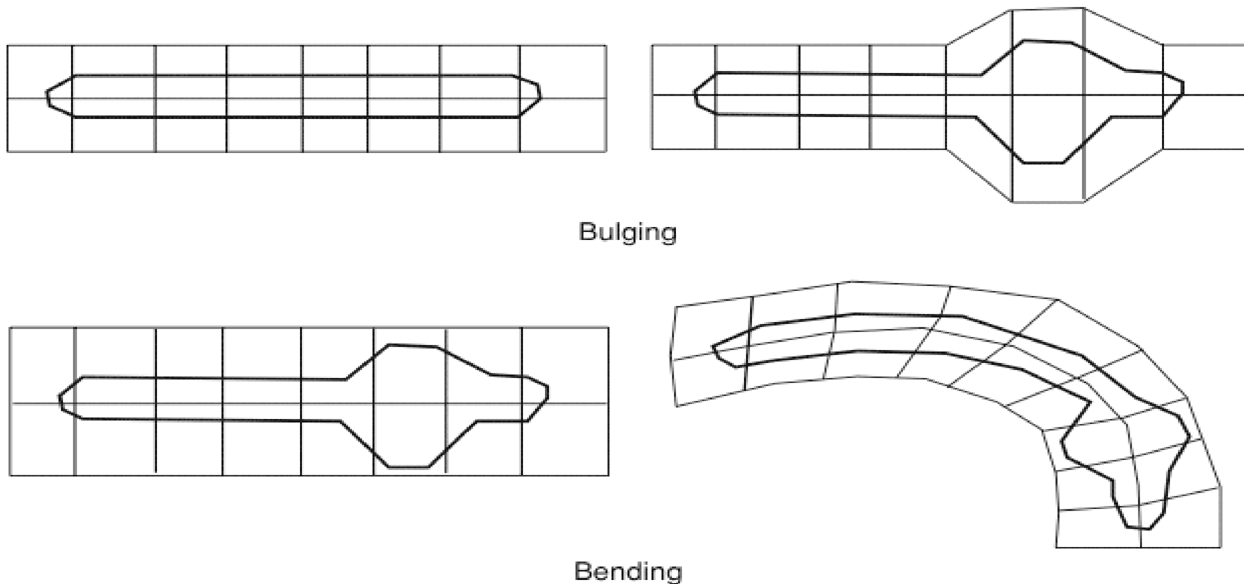
## Composite FFDs

FFDs can be composed sequentially or hierarchically. In a sequential composition, an object is modeled by progressing through a sequence of FFDs, each of which imparts a particular feature to the object. In this way, various detail elements can be added to an object in stages as opposed to trying to create one mammoth, complex FFD designed to do everything at once. For example, if a bulge is desired on a bent tube, then one FFD can be used to impart the bulge while a second one is designed to bend the object.

Chapter 2

# Composite FFDs:

sequence of bulging and bending:

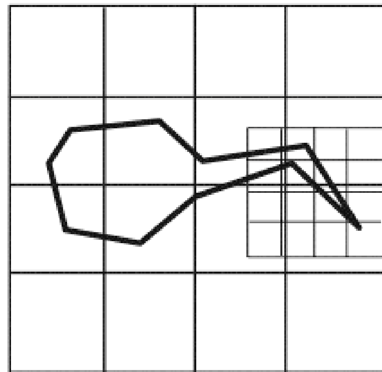


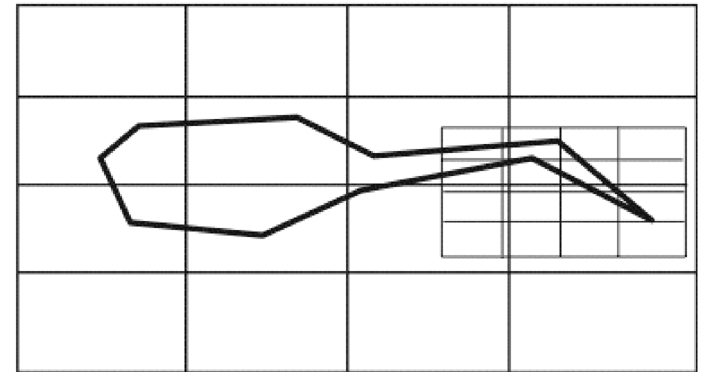
Chapter 2

# Composite FFDs

Organizing FFDs hierarchically allows the user to work at various levels of detail. Finer resolution FFDs, usually localized, are embedded inside FFDs higher in the hierarchy. As a coarser-level FFD us used to modify the object’s vertices, it also modifies the control points of any of its children FFDs that are within the space affected by the deformation. A modification made at a finer level in the hierarchy will remian well defined even as the animator works at a coarser level by modifying an FFD grid higher up in the hierarchy.

# FFDs hierarchical





# Chapter 2

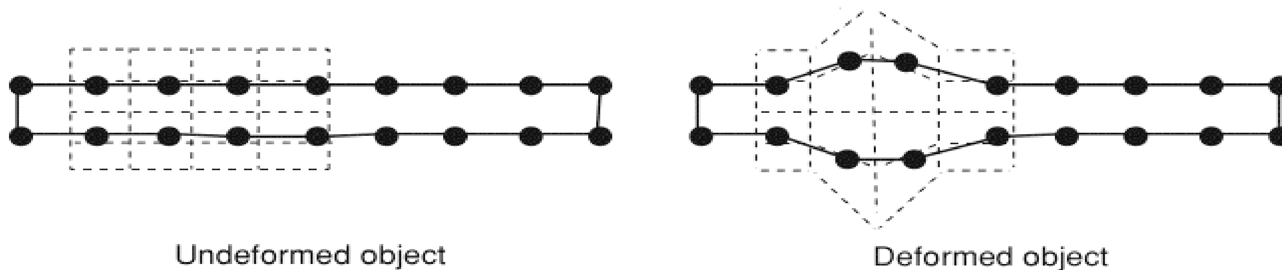
## Animated FFDs

FFDs can also be used to control an animation in one of two ways. The FFD can be constructed so that traversal of an object through the FFD space results in a continuous transformation of its shape. Alternatively, the control points of an FFD can be animated, which results in an animated deformation that automatically animates the object's shape.

Chapter 2

# FFDs – as tools to design shapes

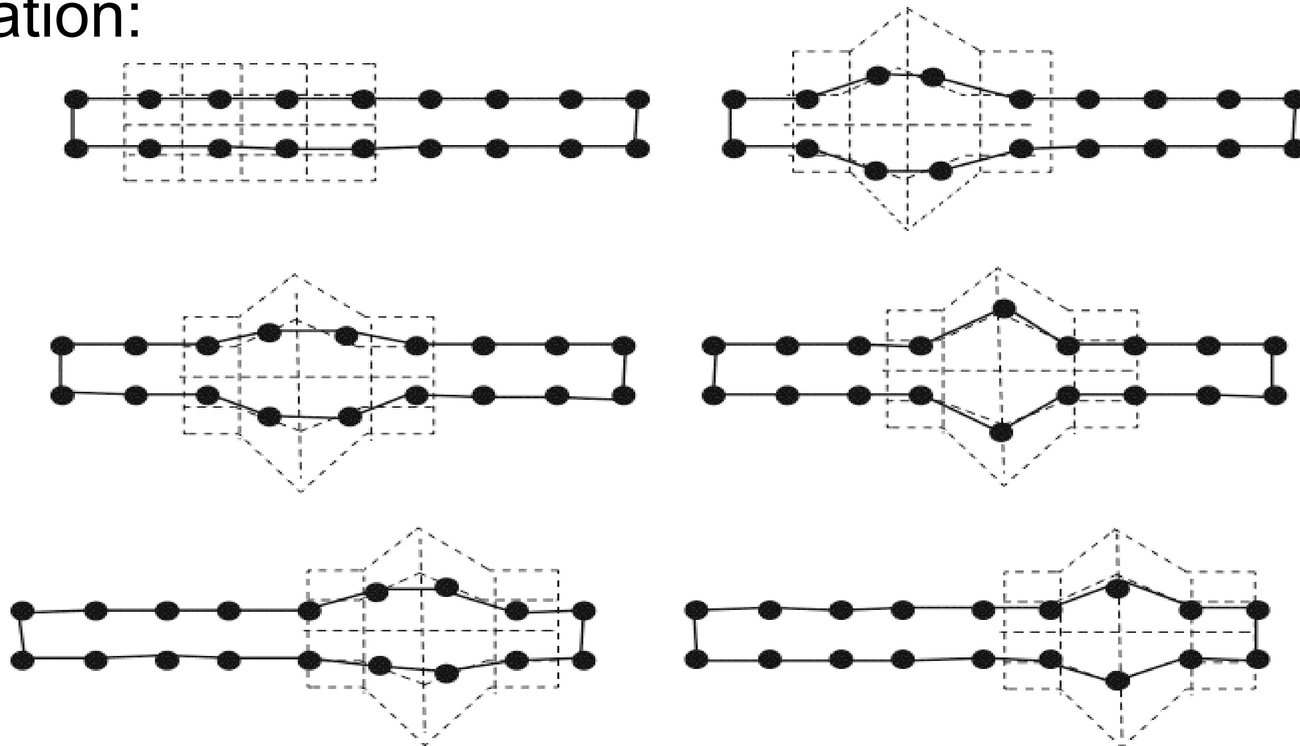
First, some form of deformation is defined based on the FFD technique:



Chapter 2

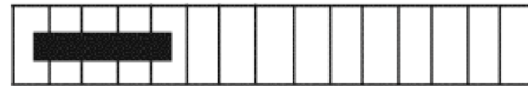
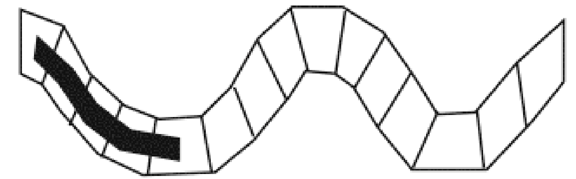
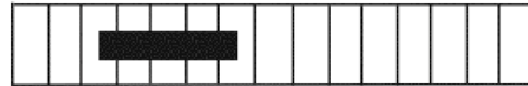
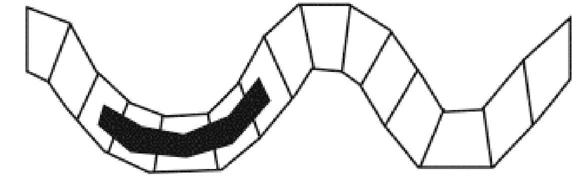
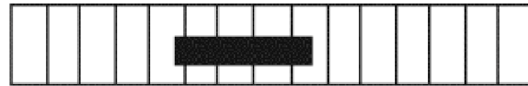
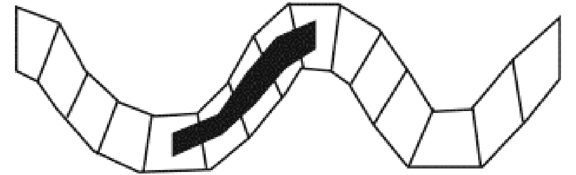
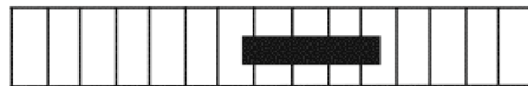
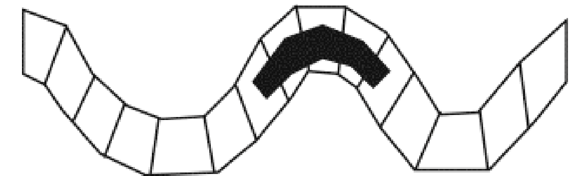
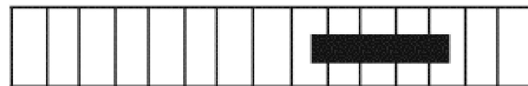
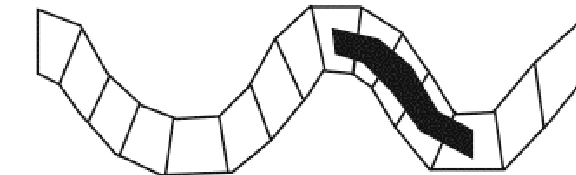
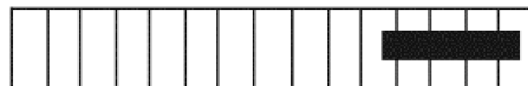
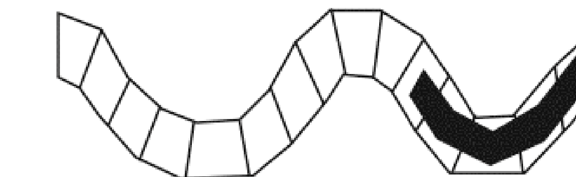
# Animated FFDs

Now, the previously defined deformation is moved relative to the object, thereby generating the animation:



# FFDs

Alternatively, the object can translate though the local deformation space of the FFD and be deformed by the progression through the FFD.

Object traversing the logical FFD coordinate space

Object traversing the distorted space

# Chapter 2

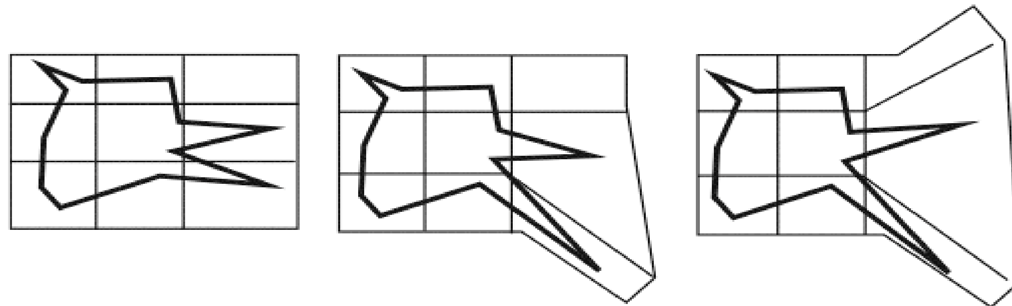
## Animated FFDs

Another way to animate an object using FFDs is to animate the control points of the FFD. For example, the FFD control points can be animated explicitly using key-frame animation, or their movement can be the result of a physically-based simulation. As the FFD grid points move, they define a changing deformation to be applied to the object's vertices.

# Chapter 2

## FFDs

### Facial animation by manipulating FFD



Chapter 2

# Animated FFDs

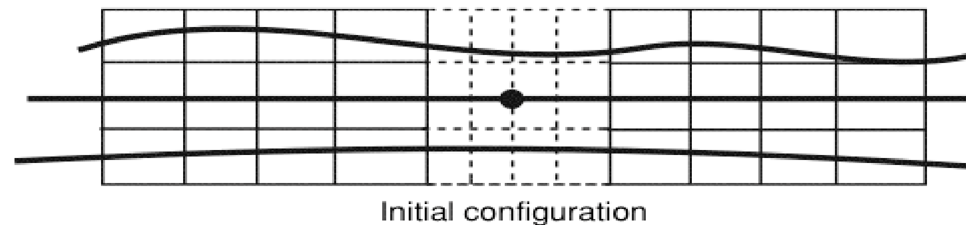
FFDs can also be used to model muscles of a human model. The muscles in this case are not meant to be anatomical representations of real muscles but to provide for a more artistic style.

As a simple example, a hinge joint with adjacent links is modeled. There are three FFDs: one for each of the two lings and one for the joint. The FFDs associated with the links will deform the skin according to a stylized muscle, and the purpose of the FFD associated with the joint is to prevent interpenetration of the skin surface in highly bent configurations:

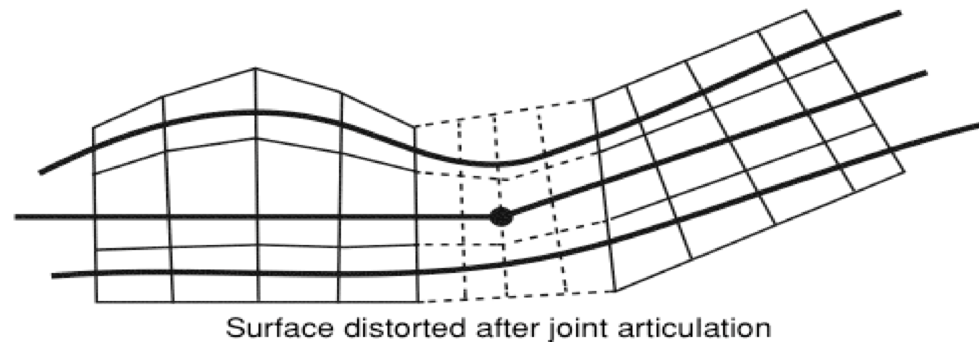
# Chapter 2

## FFDs

**Exo-muscular system**
**Skeleton -> changes FFD -> changes skin**



Initial configuration



Surface distorted after joint articulation

# Chapter 2

## 3D Shape Interpolation

We first need to define a few terms:

**topology (mathematical)**:

Describes the connectivity of the surface of an object, i.e. the number of holes and the number of separate bodies

**genus**:

Number of holes of an object

**topology (computer graphics)**:

Vertex/edge/face connectivity of an object

Chapter 2

# Interpolate between 2 objects

Correspondence problem: what part of one object to map into what part of the other object

How to handle objects of different genus?

Volumetric approaches with remeshing

Interpolation problem: how to create a sequence of intermediate objects that visually represent the transformation.

# Chapter 2

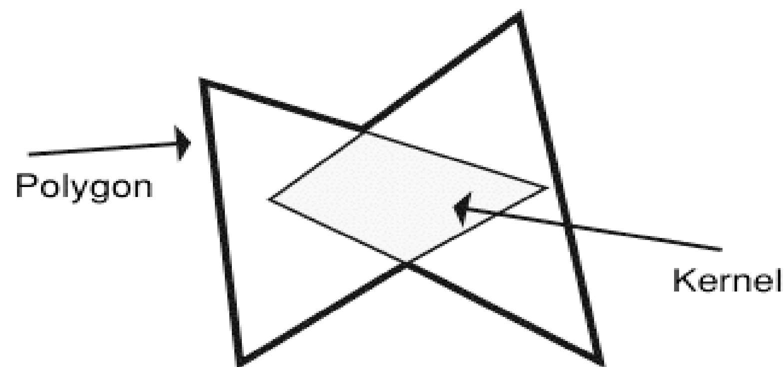
## Matching Topology

The simplest case of transforming one object into another is when the two shapes to be interpolated share the same vertex-edge topology. Here, the objects are transformed by merely interpolating the positions of the vertices on a vertex-by-vertex basis. The correspondence between the two shapes is established by the vertex-edge connectivity structure shared by the two objects. The interpolation problem is solved, as in the majority of techniques presented here, by interpolating three-dimensional vertex positions.

Chapter 2

# Radial mapping

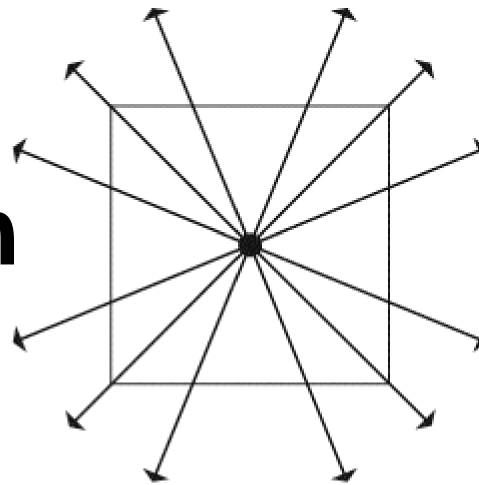
If the two objects are both star-shaped polyhedra, then polar coordinates can be used to induce a 2D mapping between the two shapes. The object surfaces are sampled by a regular distribution of rays emanating from a central point in the kernel of the object.



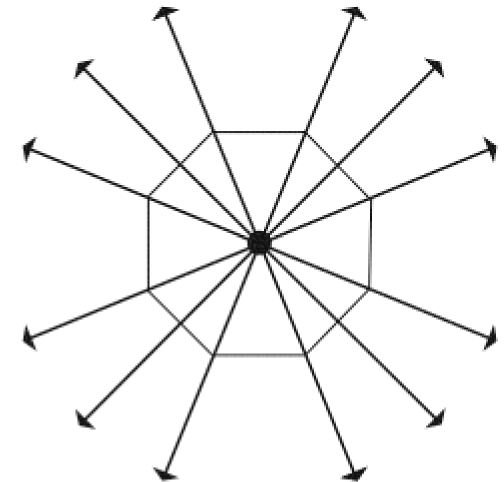
Chapter 2

# Object interpolation

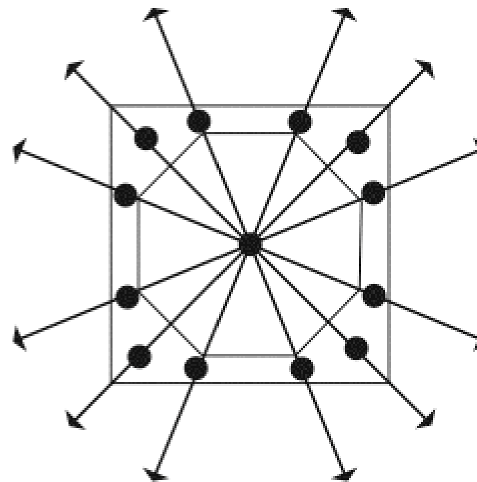
The vertices of an intermediate object are constructed by interpolating between the intersection points along a ray



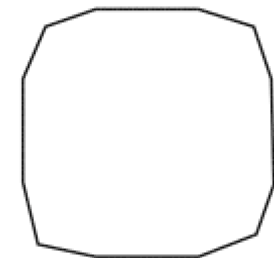
Sampling Object 1 along rays



Sampling Object 2 along rays



Points interpolated halfway between objects



Resulting object

# Chapter 2

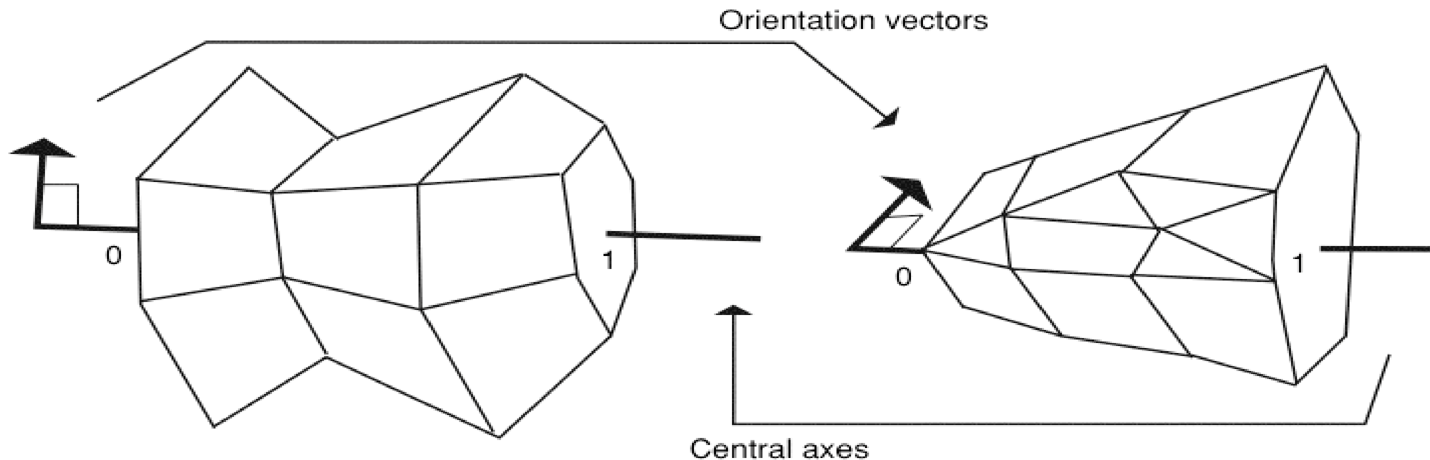
## Axial Slices

The idea can be extended to 3D objects as well. For each object, the user defines an axis that runs through the middle of the object. At regular intervals along this axis, perpendicular slices are taken of an object. These slices must be star shaped with respect to the point of intersection between the axis and the slice. This central axis is defined for both objects, and the part of each axis interior to its respective object is parameterized from zero to one. In addition, the user defines an orientation vector (or a default direction is used) that is perpendicular to the axis.

Chapter 2

# Axial Slices

**For cylinder-like objects**



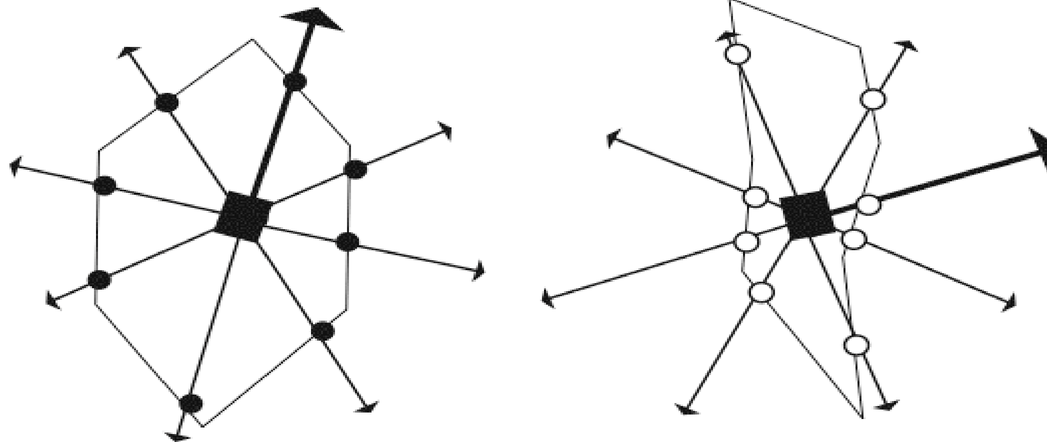
# Chapter 2

## Axial Slices

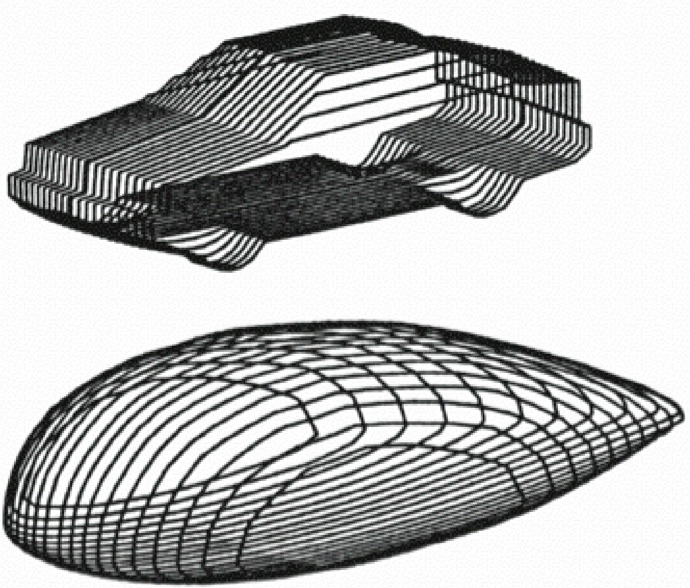
Corresponding slices (in the sense that they use the same axis parameter to define the plane of intersection) are taken from each object. The two-dimensional slices can be interpolated pair-wise (one from each object) by constructing rays that emanate from the center point and sample the boundary at regular intervals with respect to the orientation vector.

Chapter 2

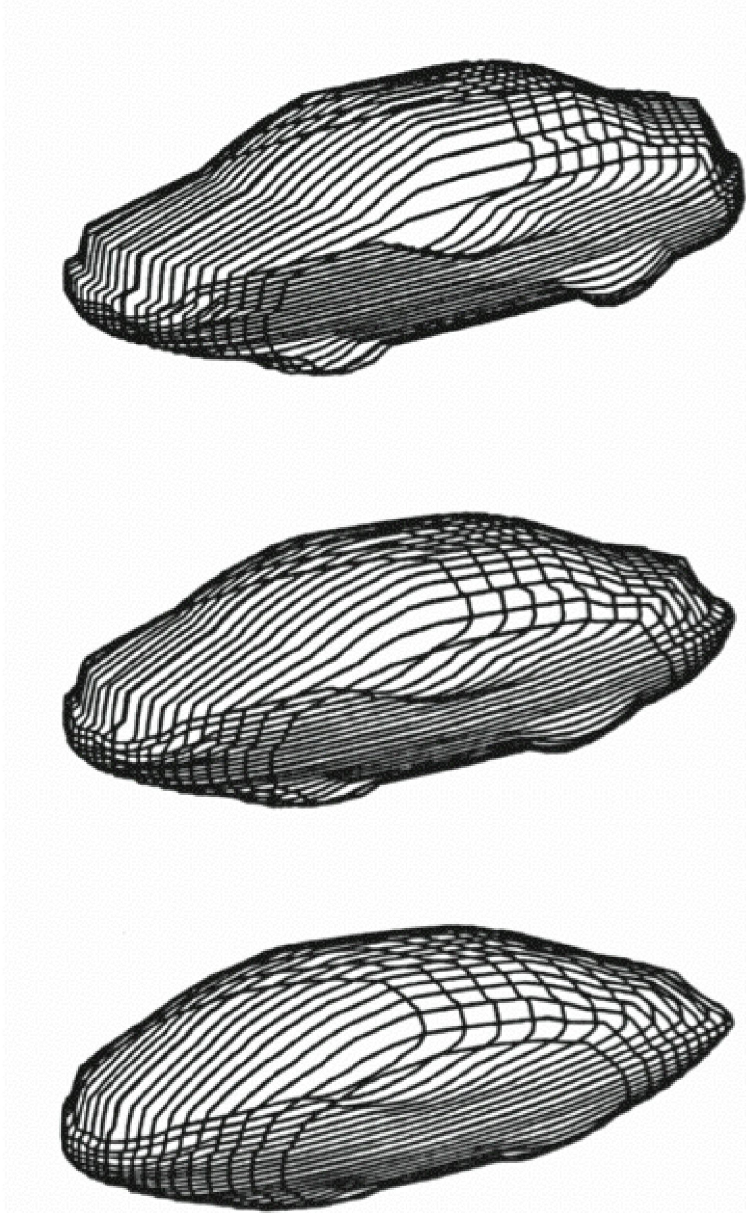
# Axial Slices



The denser the sampling, the more accurate the approximation to the original object. The corresponding points can then be interpolated in 3D space.



Original shapes sliced into contours



Interpolated shapes

Application of axial slice technique in a more complex example.

# Chapter 2

## Object Interpolation

Even among genus 0 objects, more complex polyhedra may not be star shaped or allow an internal axis to define star-shaped slices. A more complicated mapping procedure may be required to establish the 2D parameterization of the object's surfaces. One approach is to map both objects onto a common surface, such as a unit sphere. The mapping must be such that the entire object surface maps to the entire sphere with no overlap. Once both objects have been mapped onto the sphere, a union of their vertex-edge topologies can be constructed and then inversely mapped back onto each original object.

# Chapter 2

## Object Interpolation

If both objects are successfully mapped to the sphere's surface, the projected edges are intersected and merged into one topology. The new vertices and edges are a superset of both object topologies. They are then projected back onto both object surfaces. This produces two new object definitions, identical in shape to the original objects but now having the same vertex-edge topology, allowing for a vertex-by-vertex interpolation to transform one object into the other.

Chapter 2

# Object interpolation

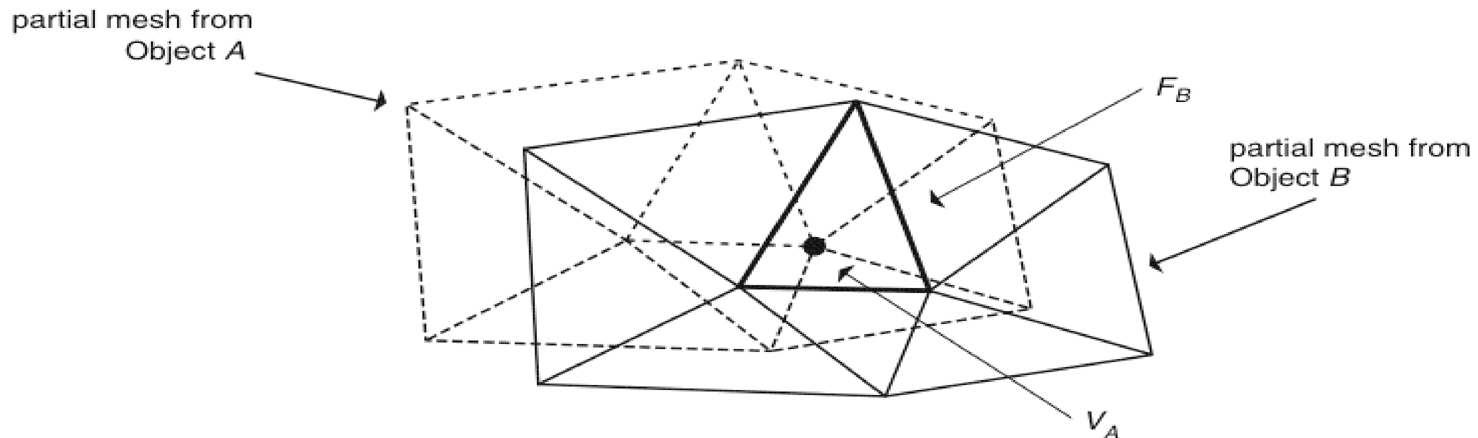
## Spherical mapping to establish matching edge-vertex topology

1. Map to sphere
2. Intersect arc-edges
3. Retriangulate
4. Remap to object shapes
5. Vertex-to-vertex interpolation

Chapter 2

# Map to sphere

The algorithm starts by considering one vertex $V_a$ of object $A$ and finding the face $F_b$ of object $B$ that contains vertex $V_a$. The edges emanating from $V_a$ are added to the work list. Face $F_b$ becomes the current face, and all edges of face $F_b$ are put on each edge's intersection-candidate list.



# Chapter 2

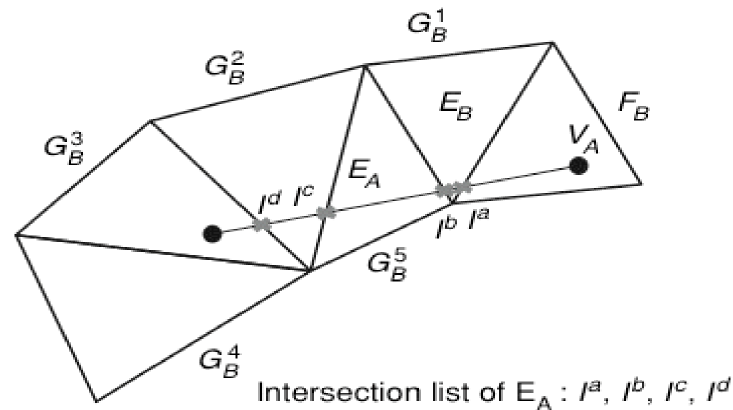
## Object Interpolation

Every edge $E_a$ is tested for any intersection with the edges on its associated intersection-candidates list. If no intersections are found, intersection processing for this edge is complete. If an intersection $I$ is found with one of the edge $E_b$ then the following steps performed:

- $I$ is added to the final model
- $I$ is added to the intersection lists of both $E_a$ and $E_b$
- The face $G_b$ on the other side of $E_b$ becomes the current face
- The other edges of face $G_b$ replace the edges in edge $E_a$'s intersection-candidate list

Chapter 2

# Object interpolation

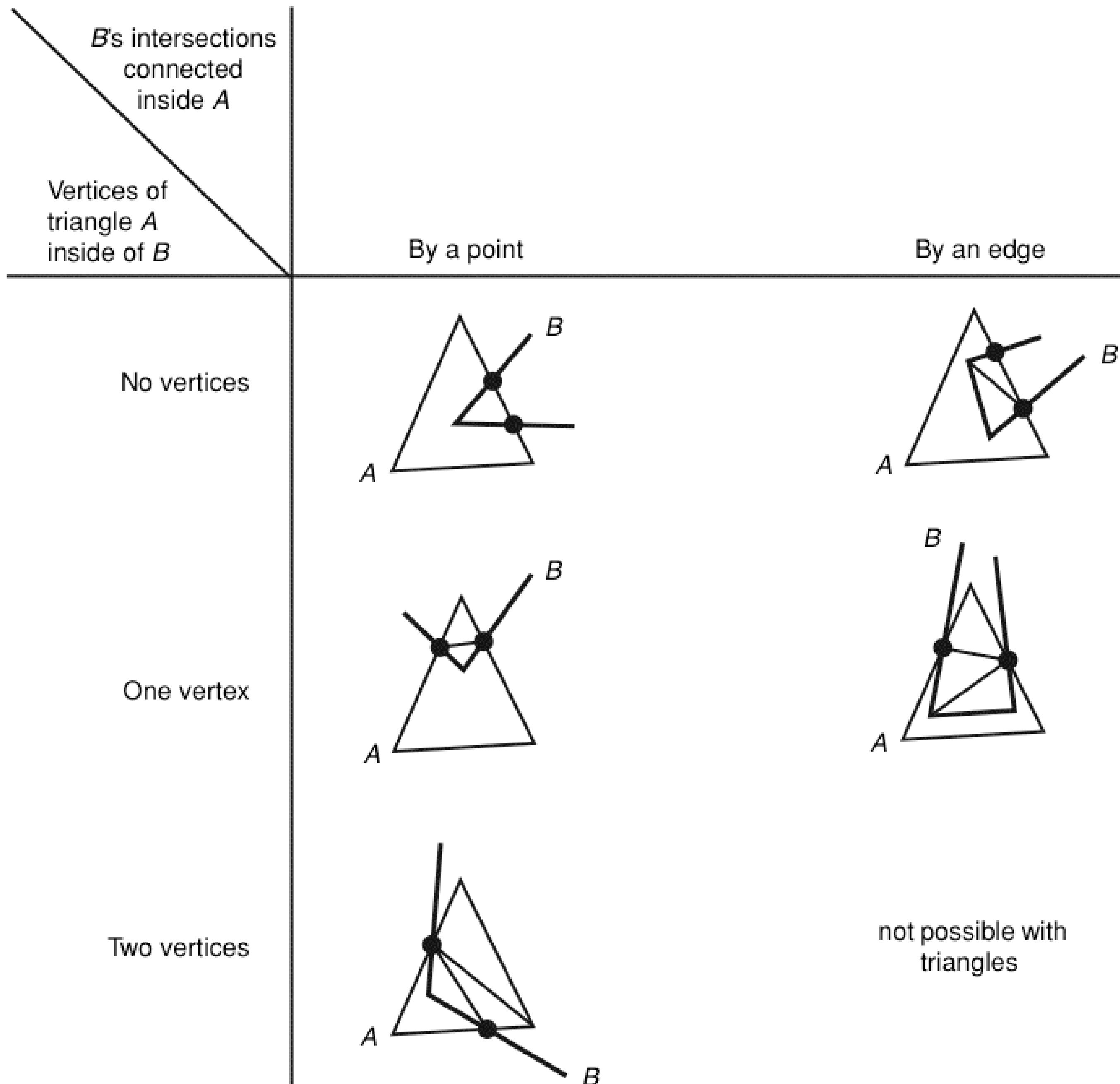



The intersections with edges of object $B$ then have to be sorted along each edge to ensure they are lined up properly along the edge.

# Chapter 2

## Object Interpolation

Now, all the vertices, edges, and intersection points are mapped back onto the original objects. New face definitions need to be constructed for each object. Because both models started out as triangulated meshes, there are only a limited number of configurations possible when one considers the triangulation required from the combined models.



B's intersections connected inside A
Vertices of triangle A inside of B
By a point
By an edge
No vertices
One vertex
Two vertices
not possible with triangles
A
B

# Chapter 2

## Object Interpolation

After this process, both objects have the same topology so that we can easily interpolate between the two by linearly interpolating between the vertices of both objects.

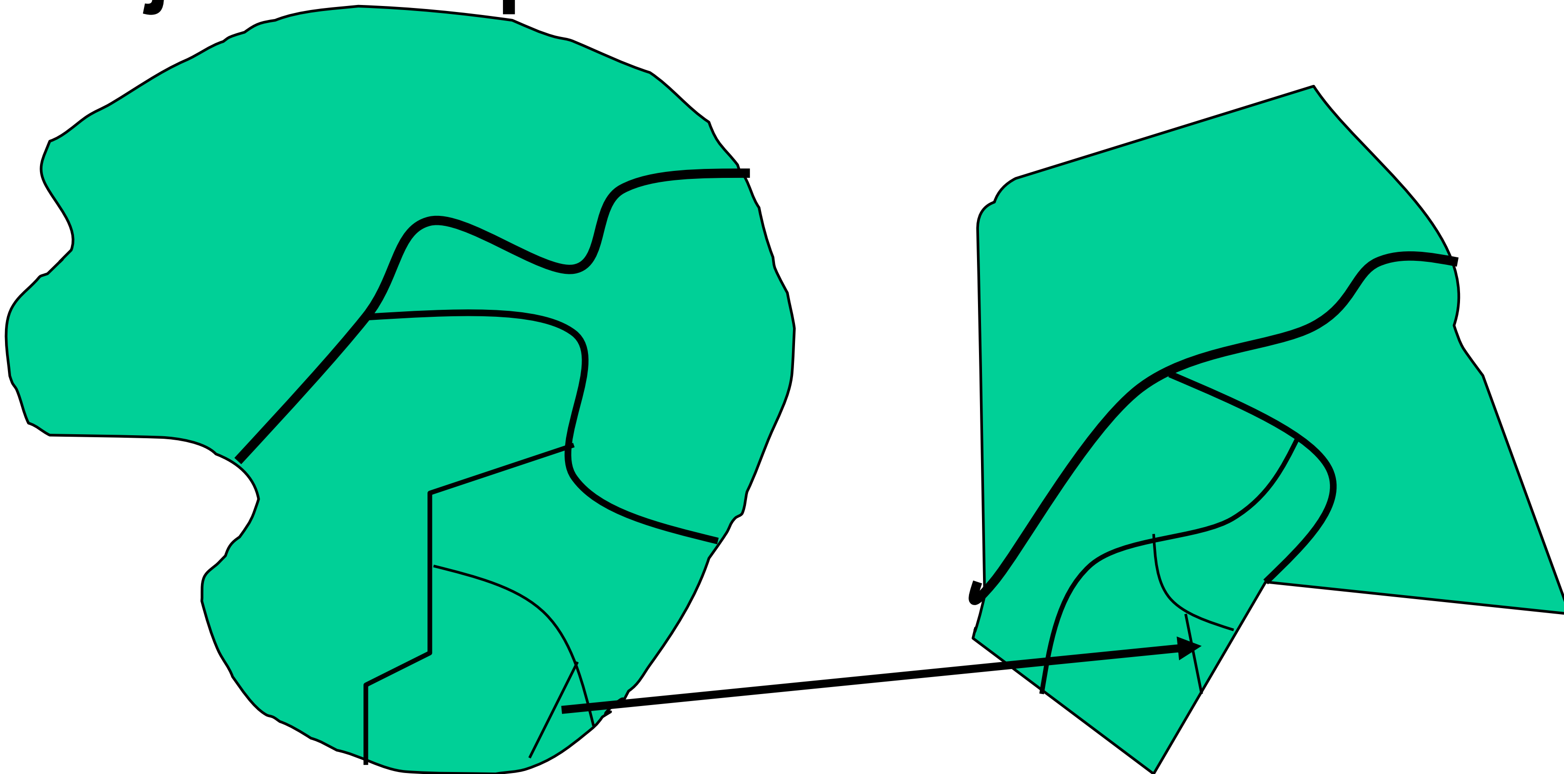
# Chapter 2

## Object Interpolation

The main problem with the previous procedure is that many new edges are created as a result of the merging operation. There is no attempt to map existing edges into one another. To avoid a plethora of new edges, a recursive approach can be taken in which each object is reduced to 2D polygonal meshes. Meshes from each object are matched by associating the boundary vertices and adding new ones when necessary. The meshes are then similarly split and the procedure is recursively applied until everything has been reduced to triangles.

# Chapter 2

## Object interpolation – recursive sheets



**Continually add vertices to make corresponding boundaries have an equal number**

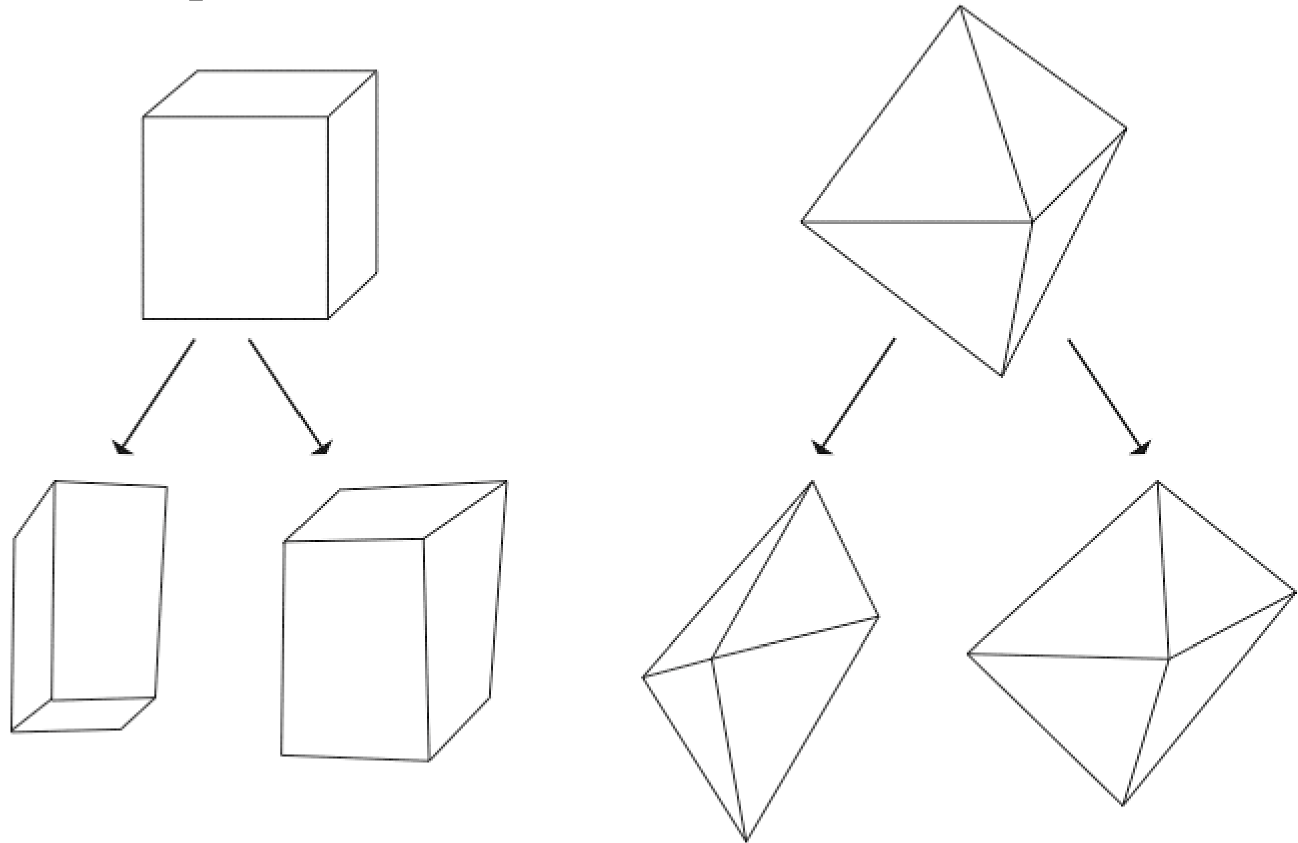
# Chapter 2

## Object Interpolation

The initial objects are divided into an initial number of polygonal meshes. Each mesh is associated with a mesh from the other object so that adjacency relationships are maintained by the mapping. The simplest way to do this is merely to break each object into two meshes: a front mesh and a back mesh. A front and back mesh can be constructed by searching for the shortest paths between the topmost, bottommost, leftmost, and rightmost vertices of the object and then appending these paths:

# Chapter 2

## Object interpolation



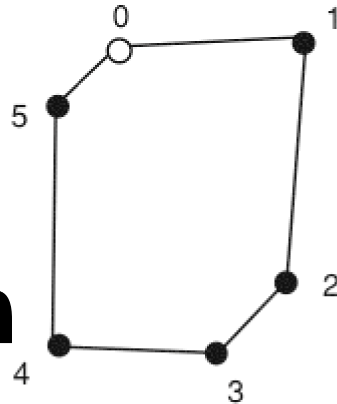
# Chapter 2

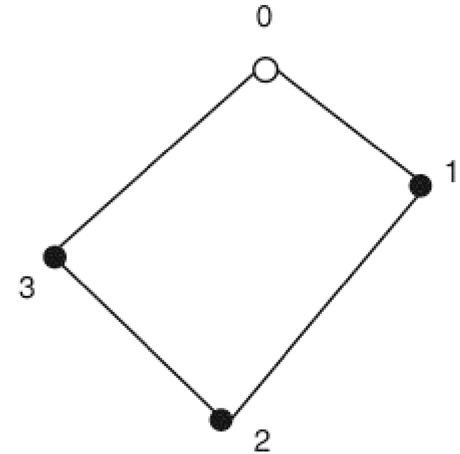
## Object Interpolation

If one of the meshes has fewer boundary vertices than the other, then new vertices must be introduced along its boundary to make up for the difference. There are various ways to do this, and the success of the algorithm is not dependent on the method. A suggested method is to compute the normalized distances of each vertex along the boundary as measured from the first vertex of the boundary. For the boundary with fewer vertices, new vertices can be added one at a time by searching for the largest gap in normalized distances for successive vertices in the boundary:

## Chapter 2

# Object interpolation



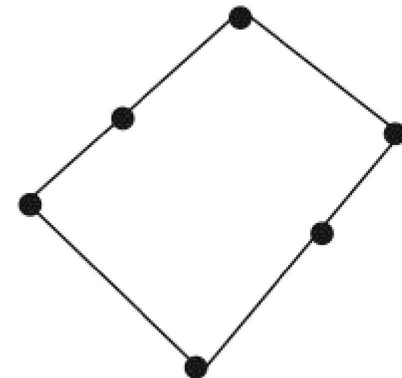

○ First vertex of boundary

Normalized distances

| | |
|---|---|
| 0 | 0.00 |
| 1 | 0.15 |
| 2 | 0.20 |
| 3 | 0.25 |
| 4 | 0.40 |
| 5 | 0.70 |

Normalized distances

| | |
|---|---|
| 0 | 0.00 |
| 1 | 0.30 |
| 2 | 0.55 |
| 3 | 0.70 |



Boundary after adding additional vertices

# Chapter 2

## Object Interpolation

When the boundaries have the same number of vertices, a vertex on one boundary is said to be associated with the vertex on the other boundary at the same relative location. Once the meshes have been associated, each mesh is recursively divided. One mesh is chosen for division, and a path of edges is found across it. One good approach for this is to choose two vertices across the boundary from each other and try to find an existing path of edges between them.

# Chapter 2

## Object Interpolation

Once a path has been found across one mesh, then a path across the mesh it is associated with must be established between corresponding vertices. This may require creating new vertices and new edges. When these paths have been created, the meshes can be divided along these paths, creating two pairs of new meshes. The boundary association, finding a path of edges, and mesh splitting are recursively applied to each new mesh until all of the meshes have been reduced to single triangles. At this point the new vertices and new edges have been added to one or both objects so that both objects have the same topology.

Chapter 2

# Morphing

Image blending
Move pixels to corresponding pixels
Blend colors

# Chapter 2

## Morphing

To transform one image into another, the user defines a curvilinear grid over each of the two images to be morphed. It is the user's responsibility to define the grids so that corresponding elements in the images are in the corresponding cells of the grids. The user defines the grid by locating the same number of grid intersection points in both images; the grid must be defined at the borders of the images in order to include the entire image. A curved mesh is then generated using the grid intersection points as control points for an interpolation scheme, such as Catmull-Rom or Bezier splines.

# Chapter 2

# Morphing

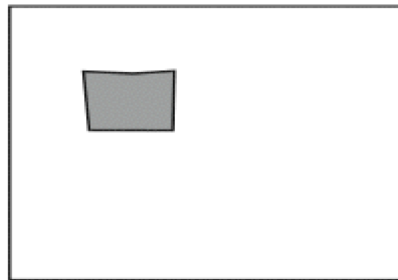


Image *A*

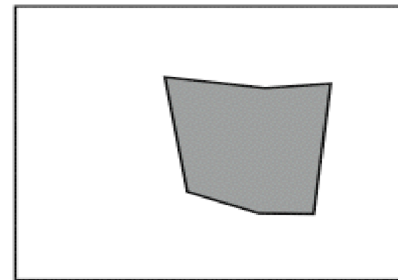


Image *B*

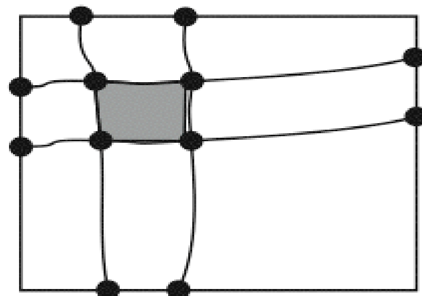


Image *A* with grid points and curves defined

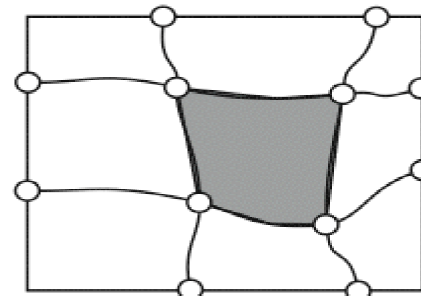


Image *B* with grid points and curves defined

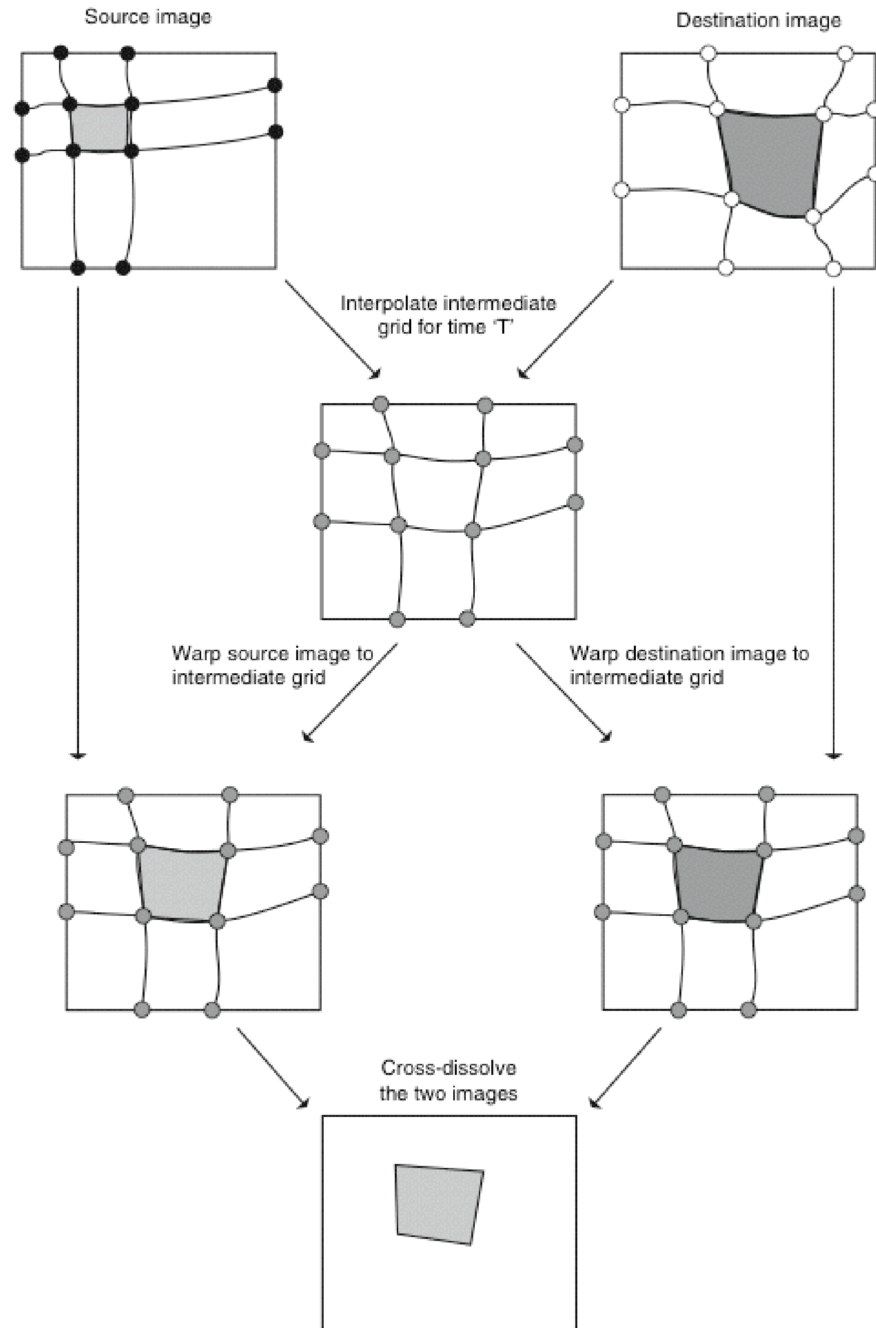
# Chapter 2

## Morphing

To generate an intermediate image along the way from the source image to the destination image, the vertices of the source and destination grids are interpolated to form an intermediate grid. This interpolation can be done linearly, or grid from adjacent key frames can be used to perform higher-order interpolation. Pixels from the source and destination images are stretched and compressed according to the intermediate grid so that warped versions of both the source image and the destination grid are generated.

## Chapter 2

# Morphing



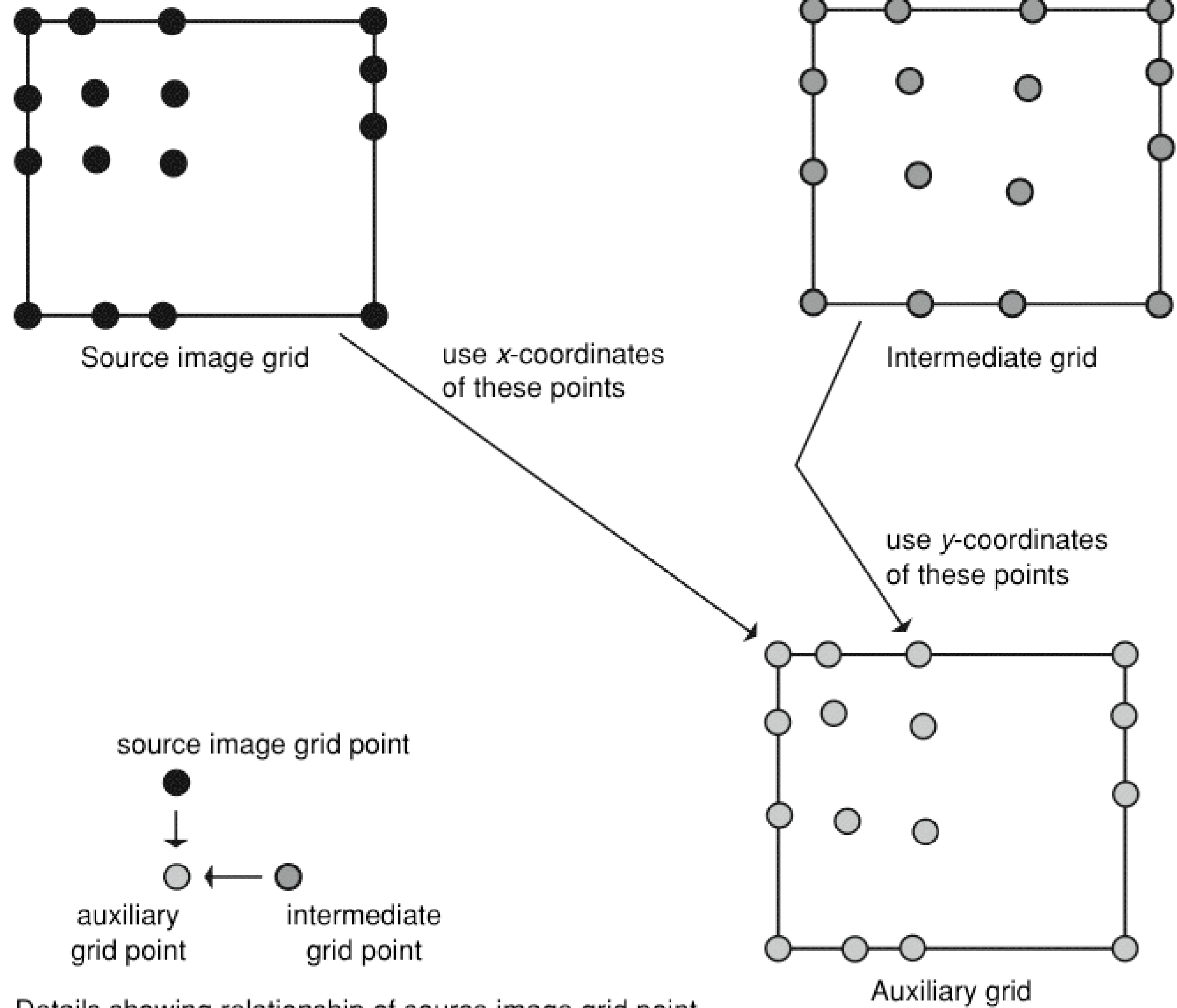
# Chapter 2

## Morphing

First, the pixels from the source image are stretched and compressed in the x-direction to fit the interpolated grid. These pixels are then stretched and compressed in the y-direction to fit the intermediate grid. To carry this out, an auxiliary grid is computed that, for each grid point, uses the x-coordinate from the corresponding grid point of the source image grid and the y-coordinate from the corresponding point of the intermediate grid.

# Chapter 2

# Morphing

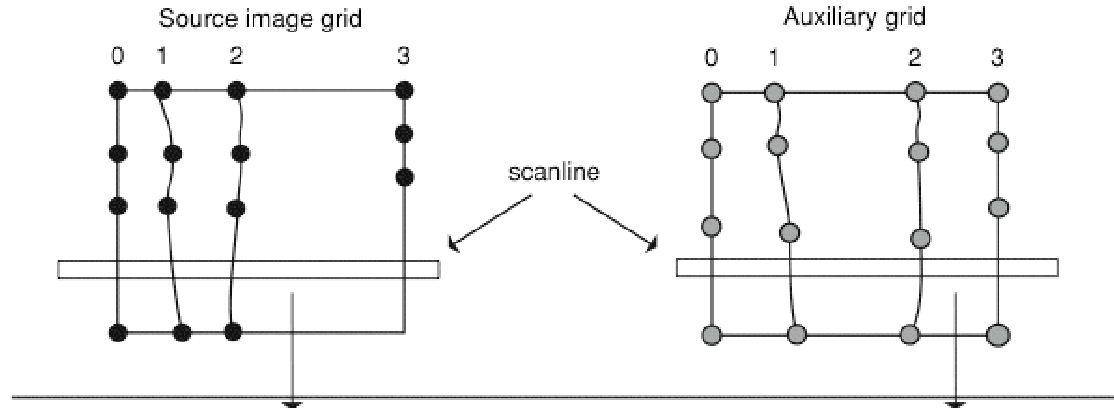



Details showing relationship of source image grid point, intermediate grid point, and auxiliary grid point

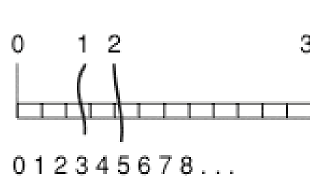
# Chapter 2

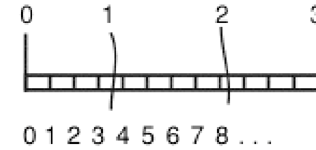
# Morphing

This auxiliary grid is then used to distort the source pixels in the x-direction.

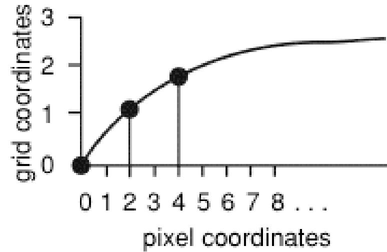



grid coordinates

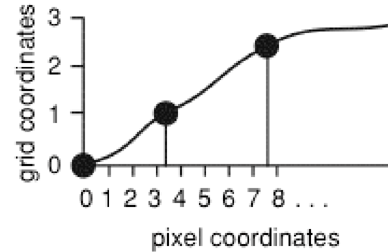


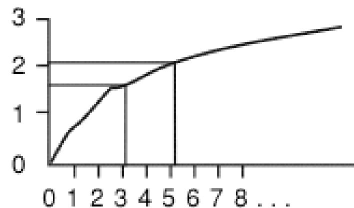

pixel coordinates



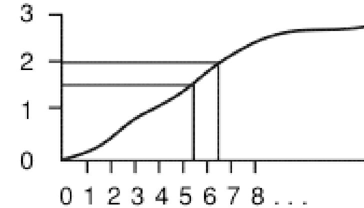

pixel coordinate to grid coordinate graph for source image




pixel coordinate to grid coordinate graph for auxiliary image




Use the graph to see where the column indices map to image pixels. (Here, half of pixel 3 and all of pixels 4 and 5 are useful)

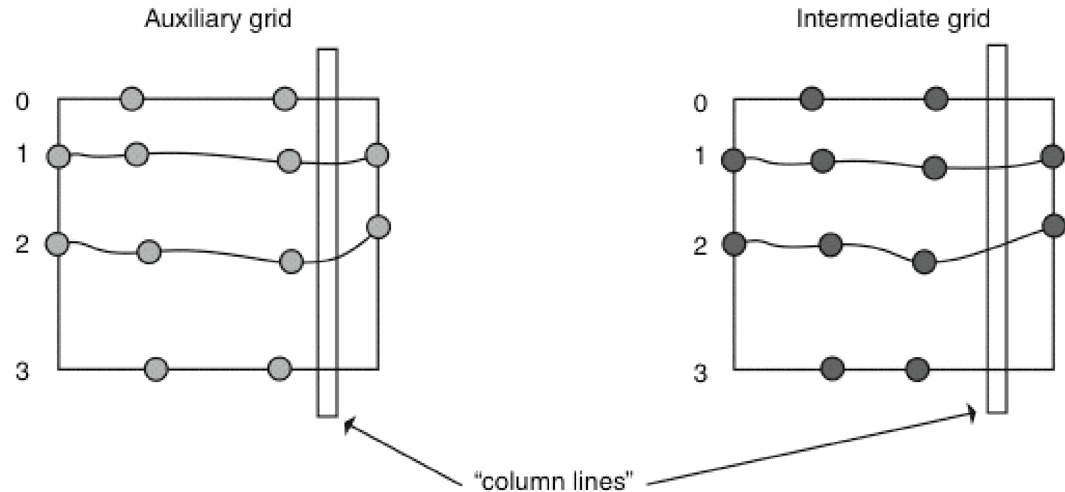


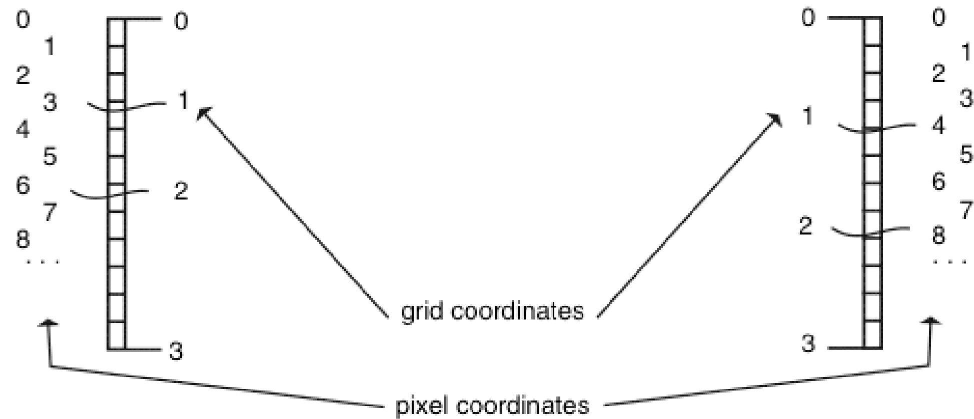

Use the graph to determine the image pixel's range in terms of the column indices (pixel 6 is shown)

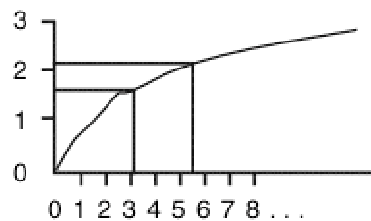
## Chapter 2

# Morphing

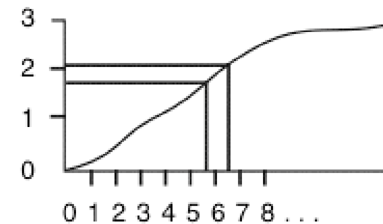
Then we can distort in y-direction as well.




Use row index coordinates to determine the pixel coordinates in auxiliary image

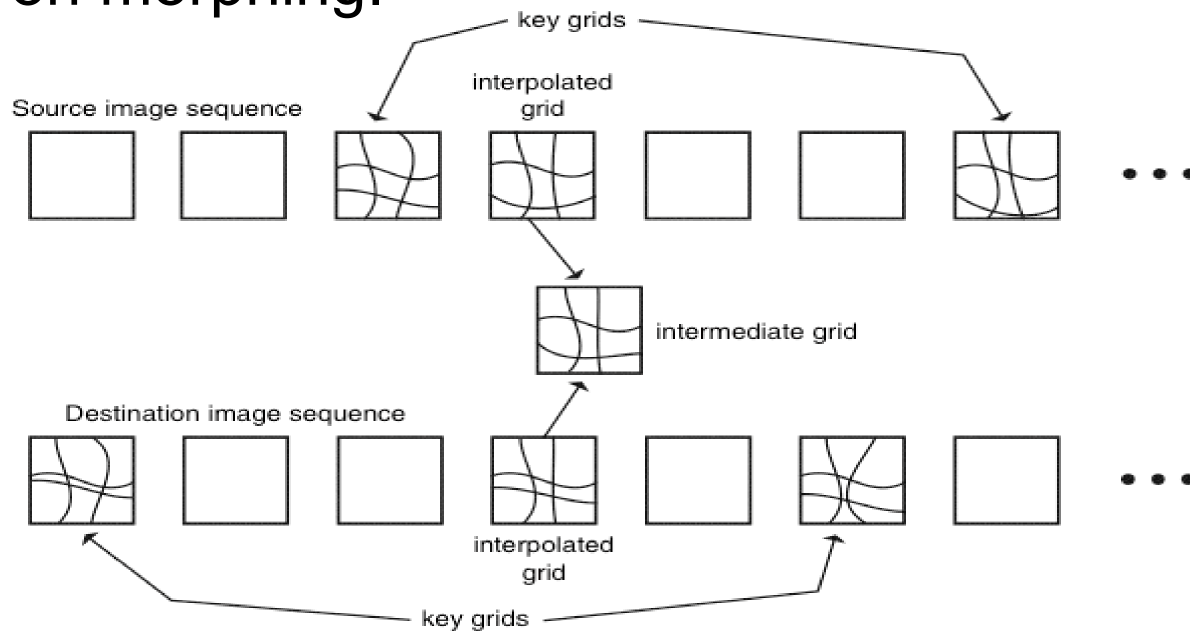



For a given pixel in the intermediate image, determine the coordinates in terms of row indices

Chapter 2

# Morphing

By specifying grid sequences for both the source and the destination images, animations can be generated based on morphing.

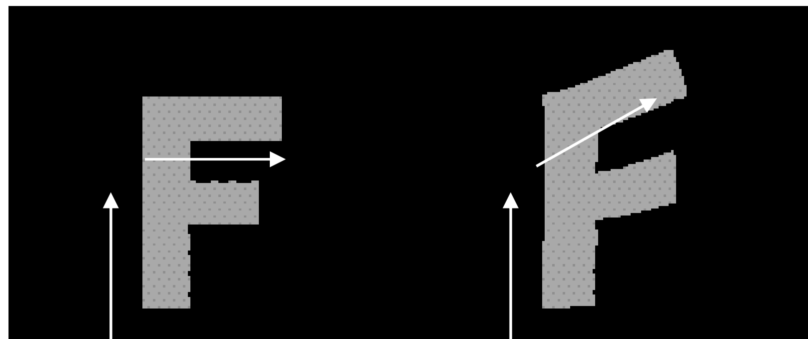


Chapter 2

# Morphing: feature based

Given: corresponding user-defined feature lines in source and destination images

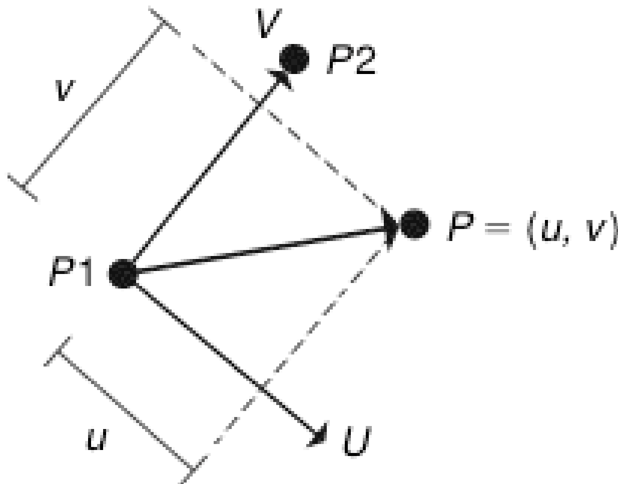
These feature lines define a coordinate transformation which can be applied to the image.



## Chapter 2

# Morphing: feature based

Compute coordinates $(u,v)$ of a pixel based on the destination feature defined by $P_1$ and $P_2$



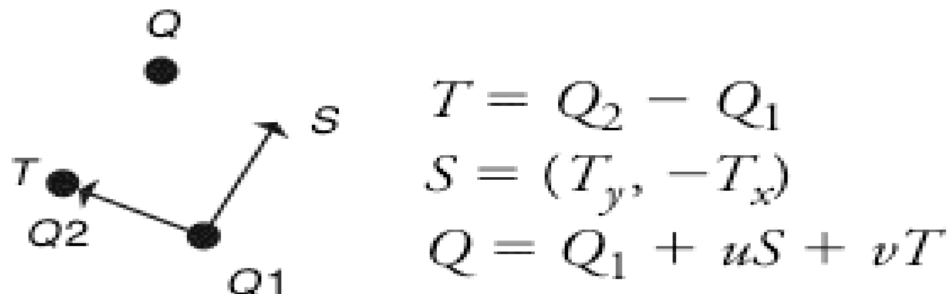

$$v = (P - P_1) \cdot \frac{(P_2 - P_1)}{|P_2 - P_1|^2}$$

$$u = \left| (P - P_1) \times \frac{(P_2 - P_1)}{|P_2 - P_1|^2} \right|$$

Chapter 2

# Morphing: feature based

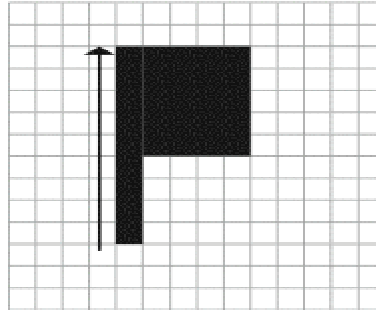
For every destination pixel, compute the location of the source pixel based on the source feature defined by $Q_1$ and $Q_2$



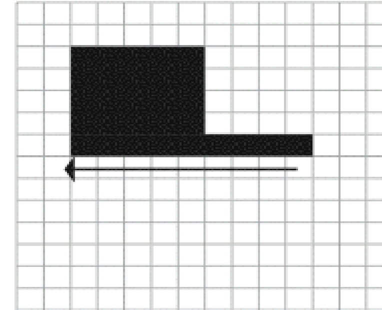
# Chapter 2

# Morphing: feature based

Source image and feature line

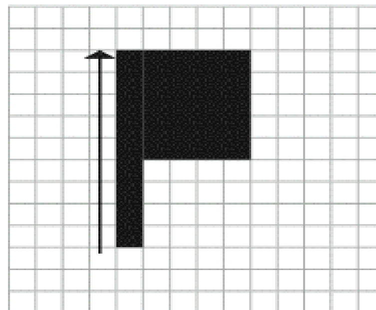


Intermediate feature line and resulting image

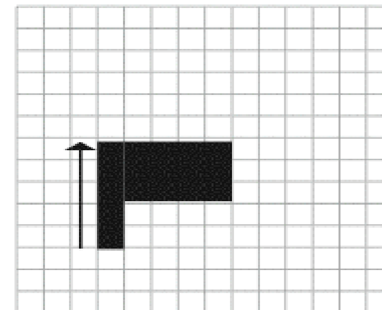


First example

Source image and feature line



Intermediate feature line and resulting image



Second example

# Chapter 2

Morphing video that illustrates 3D facial animation