Lecture 6: Bézier Curves

de Casteljau — Bending a Line Until It Becomes Anything

CS 453 — Computer Graphics

Fakhir Shaheen

Forman Christian University

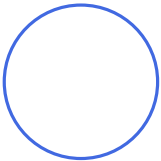
Where We Are

- **Monday**: vectors, lerp — walking from A toward B
- **Wednesday**: the dial t — lines, circles, ellipses, and motion, all from one $\mathbf{p}(t)$
- **Today**: curves you **design** — drag a handle, the curve follows. Built from *nothing but lerp.*

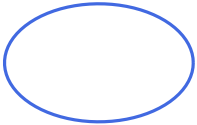
A1 is due tonight, 11:59 pm. All assignments are due Friday 11:59 pm. Monday starts a new chapter: matrices.

The Gap Wednesday Left

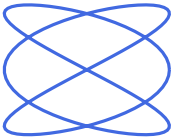
Wednesday's Curves Were All *Given* to You



`circle_point`



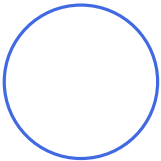
`ellipse_point`



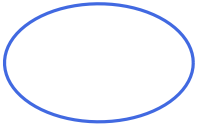
`(cos 3t, sin 2t)`

Each shape needed **someone to hand you its formula**.

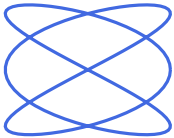
Wednesday's Curves Were All *Given* to You



`circle_point`



`ellipse_point`



`(cos 3t, sin 2t)`

this one

Each shape needed **someone to hand you its formula**.

But what is the formula for the swoosh you just sketched on a napkin? For the outline of the letter **S**? For *your* kite's flight path?

Two Ways to Get a Curve

Derive it

"I want it to swoop like *that*"

↓

solve for a formula

↓

$y = ax^3 + bx^2 + cx + d$?

Now nudge it 3 pixels left. Re-solve. Every time.

vs.

Two Ways to Get a Curve

Derive it

"I want it to swoop like *that*"

↓

solve for a formula

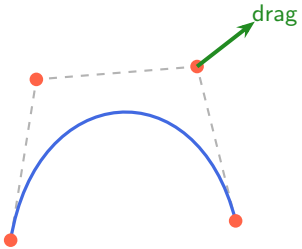
↓

$y = ax^3 + bx^2 + cx + d$?

Now nudge it 3 pixels left. Re-solve. Every time.

vs.

Steer it



A few **handles** you can grab. Move one, the curve moves with it.

Two Ways to Get a Curve

Derive it

"I want it to swoop like *that*"

↓

solve for a formula

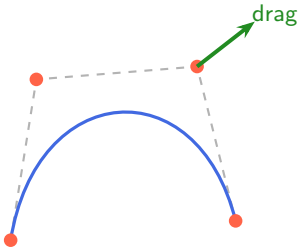
↓

$y = ax^3 + bx^2 + cx + d$?

Now nudge it 3 pixels left. Re-solve. Every time.

vs.

Steer it



A few **handles** you can grab. Move one, the curve moves with it.

Graphics picked the second one — **everywhere.**

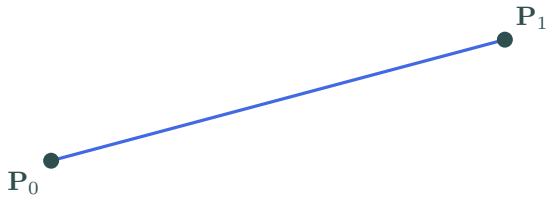
You Have Used These All Day

- every **font** on this screen: letter outlines are Bézier curves
- Illustrator, Figma, Inkscape, SVG, PDF: the pen tool *is* this
- animation software: "ease in / ease out" curves in After Effects, CSS `cubic-bezier(...)`
- camera paths, robot arms, CNC toolpaths, car body panels
  (Bézier worked at *Renault*; de Casteljau at *Citroën* — car companies, 1959–62)

Building a Curve from Lerp Alone

Start Where We Already Are: Two Points

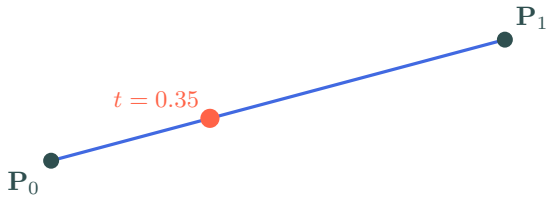
Wednesday: two points, one lerp, a straight line.



$$\mathbf{p}(t) \;=\; \mathsf{lerp}(\mathbf{P}_0, \mathbf{P}_1, t) \;=\; \mathbf{P}_0 + t\,(\mathbf{P}_1 - \mathbf{P}_0)$$

Start Where We Already Are: Two Points

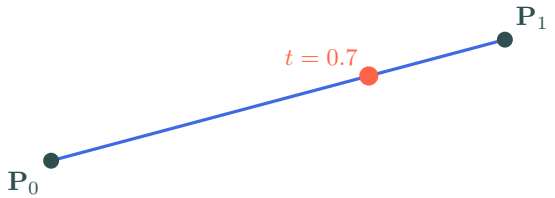
Wednesday: two points, one lerp, a straight line.



$$\mathbf{p}(t) \;=\; \mathsf{lerp}(\mathbf{P}_0, \mathbf{P}_1, t) \;=\; \mathbf{P}_0 + t\,(\mathbf{P}_1 - \mathbf{P}_0)$$

Start Where We Already Are: Two Points

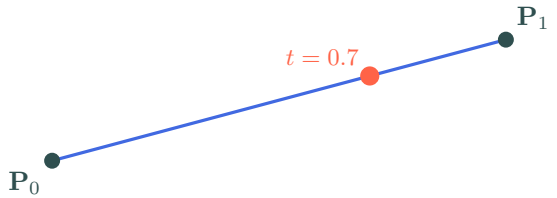
Wednesday: two points, one lerp, a straight line.



$$\mathbf{p}(t) \;=\; \mathsf{lerp}(\mathbf{P}_0, \mathbf{P}_1, t) \;=\; \mathbf{P}_0 + t\,(\mathbf{P}_1 - \mathbf{P}_0)$$

Start Where We Already Are: Two Points

Wednesday: two points, one lerp, a straight line.



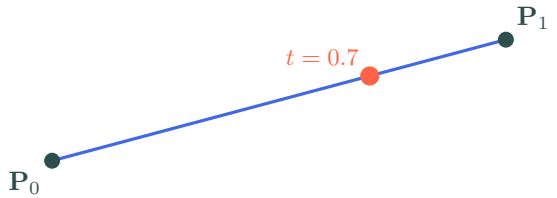
$$\mathbf{p}(t) \;=\; \text{lerp}(\mathbf{P}_0, \mathbf{P}_1, t) \;=\; \mathbf{P}_0 + t\,(\mathbf{P}_1 - \mathbf{P}_0)$$

This is a Bézier curve. **The degree-1 one.**

Two control points. It passes through both. It is perfectly straight — and perfectly boring.

Start Where We Already Are: Two Points

Wednesday: two points, one lerp, a straight line.



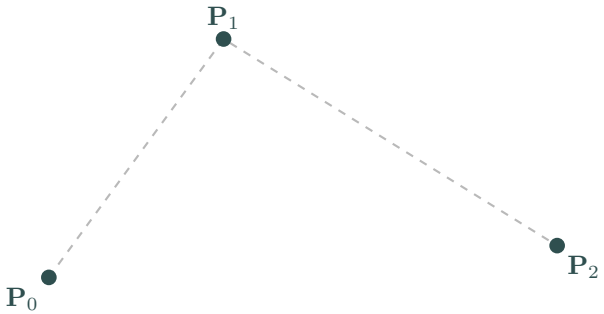
$$\mathbf{p}(t) \;=\; \text{lerp}(\mathbf{P}_0, \mathbf{P}_1, t) \;=\; \mathbf{P}_0 + t\,(\mathbf{P}_1 - \mathbf{P}_0)$$

This is a Bézier curve. **The degree-1 one.**

Two control points. It passes through both. It is perfectly straight — and perfectly boring.

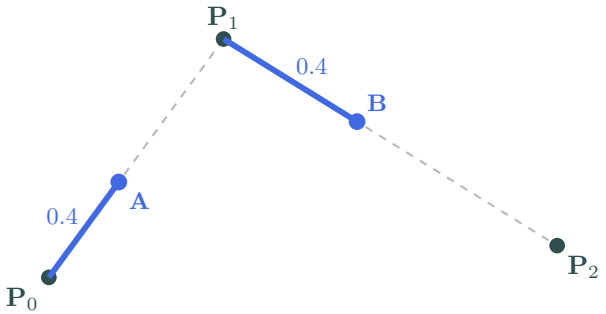
So: **add a third point.** What could it possibly mean?

Three Points: Lerp Between Two Moving Points



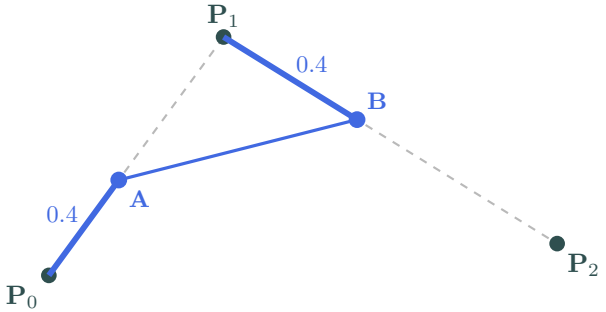
Three control points, two segments.
Pick **one** value of the dial — say $t = 0.4$ — and use it everywhere.

Three Points: Lerp Between Two Moving Points



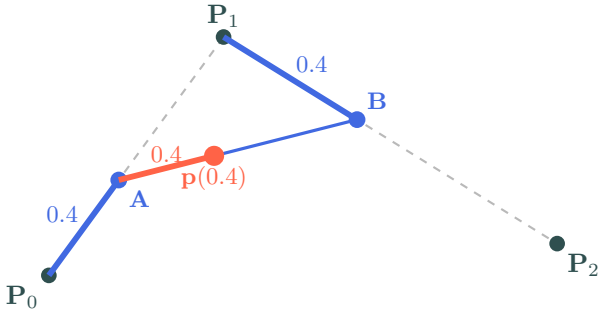
1. lerp along **both** segments *at the same time*, same t:
$$\mathbf{A} = \text{lerp}(\mathbf{P}_0, \mathbf{P}_1, 0.4), \quad \mathbf{B} = \text{lerp}(\mathbf{P}_1, \mathbf{P}_2, 0.4)$$

Three Points: Lerp Between Two Moving Points



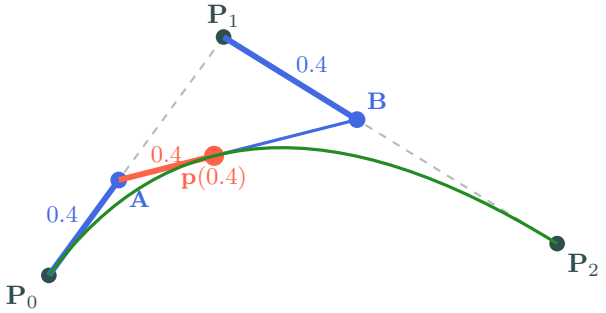
2. $\mathbf{A}$ and $\mathbf{B}$ define a **new segment** — change t and both ends move

Three Points: Lerp Between Two Moving Points



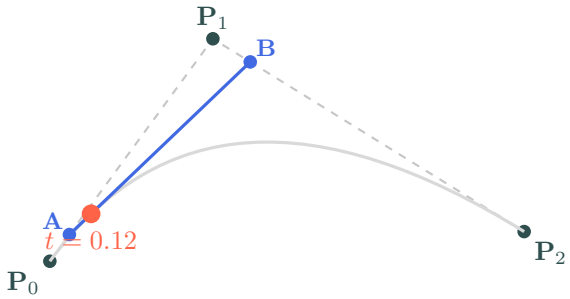
3. lerp along *that* with the **same** t — one point left:
$\mathbf{p}(0.4) = \text{lerp}(\mathbf{A}, \mathbf{B}, 0.4)$. **That is the curve point.**

Three Points: Lerp Between Two Moving Points

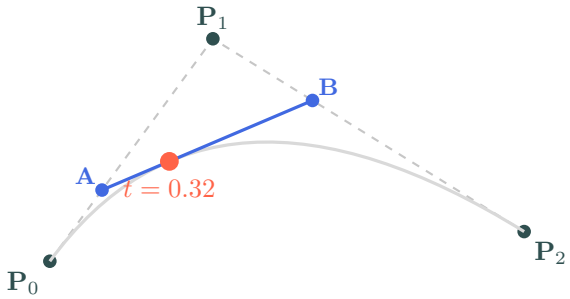


4. sweep t from 0 to 1 and the surviving point traces a curve.
Three lerps, one t. No new mathematics whatsoever.

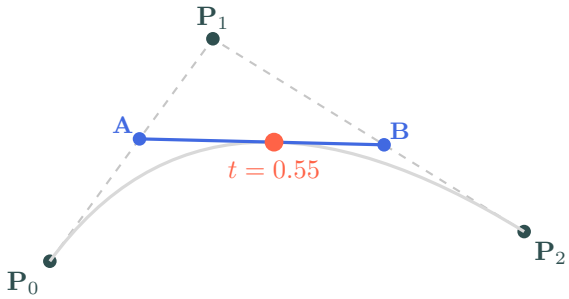
Watch the Curve Get Traced



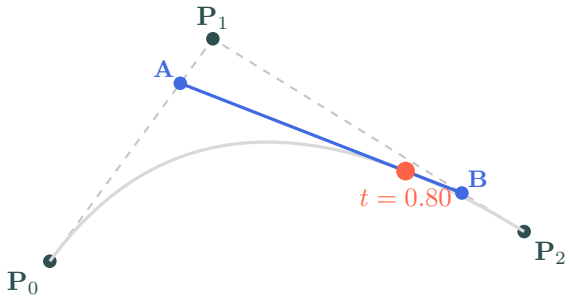
Watch the Curve Get Traced



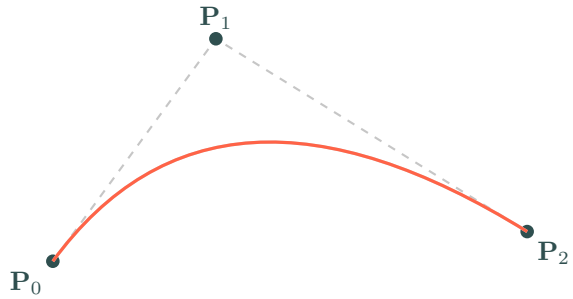
Watch the Curve Get Traced



Watch the Curve Get Traced

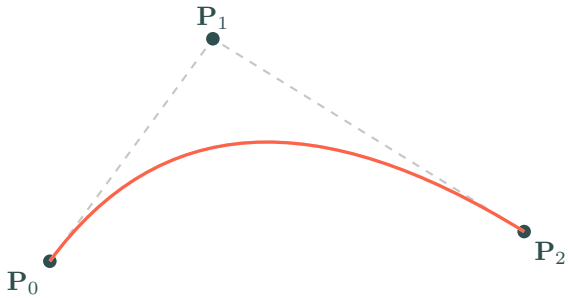


Watch the Curve Get Traced



The blue segment is a **moving ruler**; the red dot rides it.
The curve is the *trail* of a point that is always just lerping.

Watch the Curve Get Traced



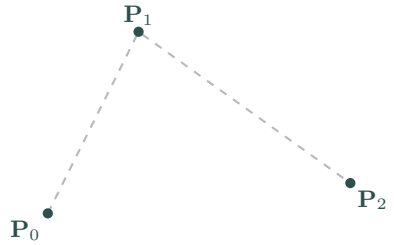
The blue segment is a **moving ruler**; the red dot rides it.
The curve is the *trail* of a point that is always just lerping.

Notice: the curve is **tangent** to $\mathbf{P}_0\mathbf{P}_1$ at the start and to $\mathbf{P}_1\mathbf{P}_2$ at the end — at $t = 0$ the ruler *is* that first segment.

The Quadratic Bezier Curve

One lerp, regrouped so its **weights** show:

$$\text{lerp}(\mathbf{P}, \mathbf{Q}, t) \;=\; \mathbf{P} + t\,(\mathbf{Q} - \mathbf{P}) \;=\; (1-t)\,\mathbf{P} + t\,\mathbf{Q}$$



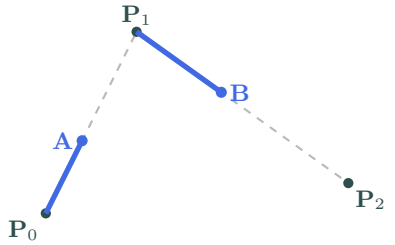
The Quadratic Bezier Curve

One lerp, regrouped so its **weights** show:

$$\text{lerp}(\mathbf{P}, \mathbf{Q}, t) \;=\; \mathbf{P} + t\,(\mathbf{Q} - \mathbf{P}) \;=\; (1-t)\,\mathbf{P} + t\,\mathbf{Q}$$

Step 1 — the two sliding points:

$$\mathbf{A} = (1-t)\mathbf{P}_0 + t\,\mathbf{P}_1$$
$$\mathbf{B} = (1-t)\mathbf{P}_1 + t\,\mathbf{P}_2$$



The Quadratic Bezier Curve

One lerp, regrouped so its **weights** show:

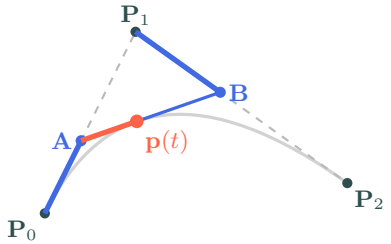
$$\text{lerp}(\mathbf{P}, \mathbf{Q}, t) \;=\; \mathbf{P} + t\,(\mathbf{Q} - \mathbf{P}) \;=\; (1-t)\,\mathbf{P} + t\,\mathbf{Q}$$

Step 1 — the two sliding points:

$$\mathbf{A} = (1-t)\mathbf{P}_0 + t\,\mathbf{P}_1$$
$$\mathbf{B} = (1-t)\mathbf{P}_1 + t\,\mathbf{P}_2$$

Step 2 — lerp those, same t: $\mathbf{p}(t) = (1-t)\mathbf{A} + t\mathbf{B}$.
Substitute:

$$\begin{aligned}\mathbf{p}(t) &= (1-t)\big[(1-t)\mathbf{P}_0 + t\,\mathbf{P}_1\big] \\ &\quad + \; t\big[(1-t)\mathbf{P}_1 + t\,\mathbf{P}_2\big] \\ &= (1-t)^2\mathbf{P}_0 + t^2\mathbf{P}_2 \\ &\quad + \; \big[(1-t)t + t(1-t)\big]\,\mathbf{P}_1\end{aligned}$$



$\mathbf{P}_1$ feeds **both** first-step lerps

The Quadratic Bezier Curve

One lerp, regrouped so its **weights** show:

$$\text{lerp}(\mathbf{P}, \mathbf{Q}, t) \;=\; \mathbf{P} + t\,(\mathbf{Q} - \mathbf{P}) \;=\; (1-t)\,\mathbf{P} + t\,\mathbf{Q}$$

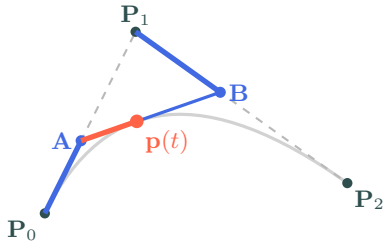
Step 1 — the two sliding points:

$$\mathbf{A} = (1-t)\mathbf{P}_0 + t\,\mathbf{P}_1$$
$$\mathbf{B} = (1-t)\mathbf{P}_1 + t\,\mathbf{P}_2$$

Step 2 — lerp those, same t: $\mathbf{p}(t) = (1-t)\mathbf{A} + t\mathbf{B}$.
Substitute:

$$\begin{aligned}\mathbf{p}(t) &= (1-t)\big[(1-t)\mathbf{P}_0 + t\,\mathbf{P}_1\big] \\ &\quad + \; t\big[(1-t)\mathbf{P}_1 + t\,\mathbf{P}_2\big] \\ &= (1-t)^2\mathbf{P}_0 + t^2\mathbf{P}_2 \\ &\quad + \; \big[(1-t)t + t(1-t)\big]\,\mathbf{P}_1\end{aligned}$$

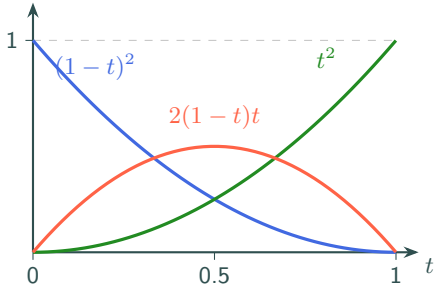
$$\mathbf{p}(t) \;=\; (1-t)^2\,\mathbf{P}_0 \;+\; 2\,(1-t)t\,\mathbf{P}_1 \;+\; t^2\,\mathbf{P}_2$$



$\mathbf{P}_1$ feeds **both** first-step lerps

Reading the Formula We Just Derived

- why the **2**? $\mathbf{P}_1$ shows up *twice* when we substitute — once inside $\mathbf{A}$, once inside $\mathbf{B}$. Each copy carries the weight $(1-t)t$; two copies add up to $2(1-t)t$. $\mathbf{P}_0$ and $\mathbf{P}_2$ show up once each, so no 2 for them

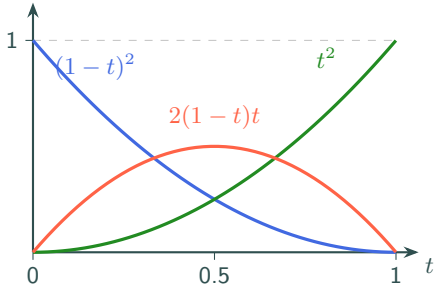


Reading the Formula We Just Derived

- why the **2**? $\mathbf{P}_1$ shows up *twice* when we substitute — once inside $\mathbf{A}$, once inside $\mathbf{B}$. Each copy carries the weight $(1-t)t$; two copies add up to $2(1-t)t$. $\mathbf{P}_0$ and $\mathbf{P}_2$ show up once each, so no 2 for them
- the three weights always sum to 1:

$$(1-t)^2 + 2(1-t)t + t^2 = \big[(1-t)+t\big]^2 = 1$$

so $\mathbf{p}(t)$ is a **weighted average** of the handles — it cannot escape them



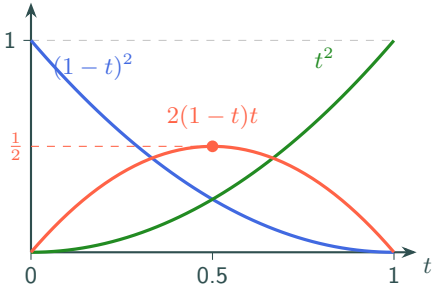
The three weights at every t. Add the heights: always exactly 1.

Reading the Formula We Just Derived

- why the **2**? $\mathbf{P}_1$ shows up *twice* when we substitute — once inside $\mathbf{A}$, once inside $\mathbf{B}$. Each copy carries the weight $(1-t)t$; two copies add up to $2(1-t)t$. $\mathbf{P}_0$ and $\mathbf{P}_2$ show up once each, so no 2 for them
- the three weights always sum to 1:

$$(1-t)^2 + 2(1-t)t + t^2 = \big[(1-t)+t\big]^2 = 1$$

  so $\mathbf{p}(t)$ is a **weighted average** of the handles — it cannot escape them
- at $t=0$ the weights are $(1,0,0)$; at $t=1$, $(0,0,1)$ — the ends are pinned. And $2(1-t)t$ tops out at $\frac{1}{2}$, so $\mathbf{P}_1$ never wins



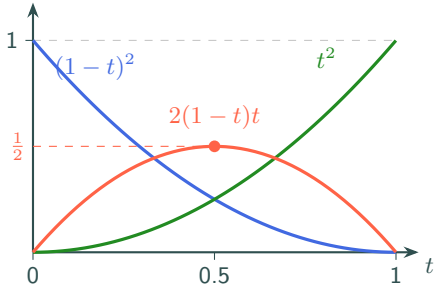
The three weights at every t. Add the heights: always exactly 1.

Reading the Formula We Just Derived

- why the **2**? $\mathbf{P}_1$ shows up *twice* when we substitute — once inside $\mathbf{A}$, once inside $\mathbf{B}$. Each copy carries the weight $(1-t)t$; two copies add up to $2(1-t)t$. $\mathbf{P}_0$ and $\mathbf{P}_2$ show up once each, so no 2 for them
- the three weights always sum to 1:

$$(1-t)^2 + 2(1-t)t + t^2 = \big[(1-t)+t\big]^2 = 1$$

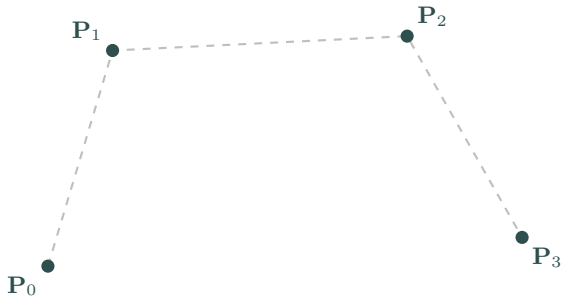
  so $\mathbf{p}(t)$ is a **weighted average** of the handles — it cannot escape them
- at $t=0$ the weights are $(1,0,0)$; at $t=1$, $(0,0,1)$ — the ends are pinned. And $2(1-t)t$ tops out at $\frac{1}{2}$, so $\mathbf{P}_1$ never wins



The three weights at every t. Add the heights: always exactly 1.

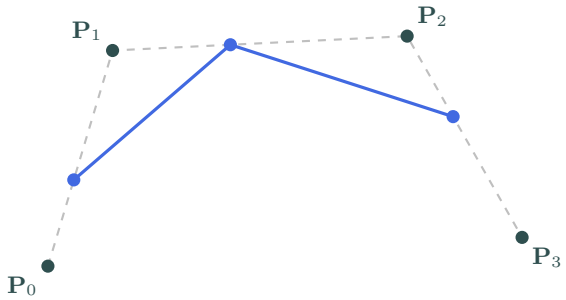
Four points, one more step: $(1-t)^3$, $3(1-t)^2t$, $3(1-t)t^2$, t^3 — one row further down Pascal's triangle, **and we will still write the loop.**

Four Points: The Same Move, One Step Longer



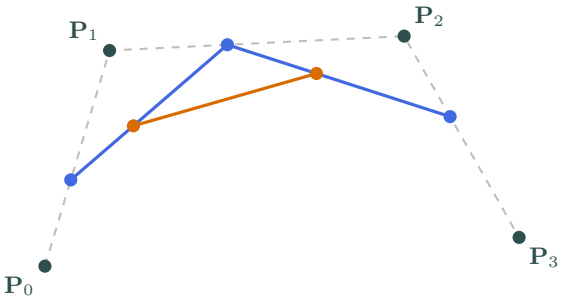
Four points, three segments. Do you already know what happens?

Four Points: The Same Move, One Step Longer



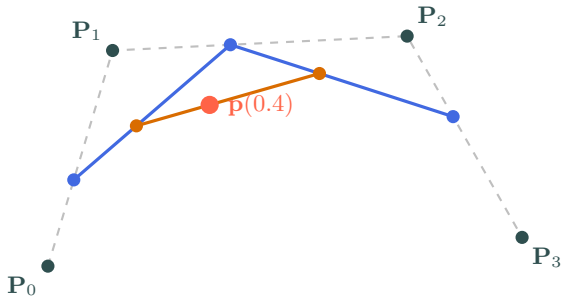
4 points $\rightarrow$ 3 points (one lerp per consecutive pair)

Four Points: The Same Move, One Step Longer



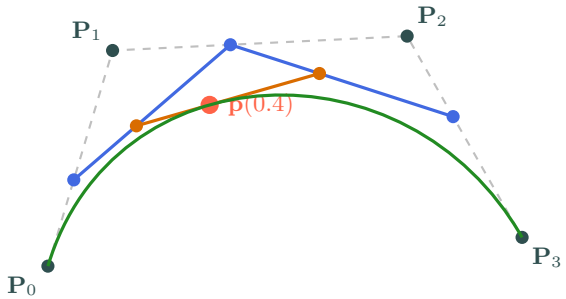
3 points $\rightarrow$ 2 points (same move again)

Four Points: The Same Move, One Step Longer



2 points $\rightarrow$ 1 point. Done — *that* is $\mathbf{p}(0.4)$.

Four Points: The Same Move, One Step Longer

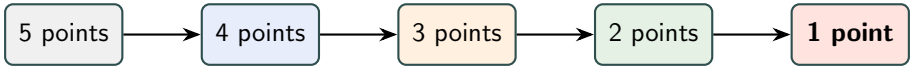


Sweep t: the **cubic Bézier** — the workhorse of every font and vector editor.

The Whole Algorithm, in One Sentence

Replace the list of points by the lerps between consecutive pairs.

Repeat until one point remains. That point is on the curve.

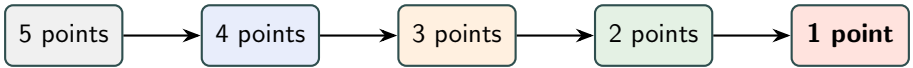


each arrow = one lerp per consecutive pair, at the same t $\mathbf{p}(t)$

The Whole Algorithm, in One Sentence

Replace the list of points by the lerps between consecutive pairs.

Repeat until one point remains. That point is on the curve.



each arrow = one lerp per consecutive pair, at the same t $\quad \mathbf{p}(t)$

n control points $\Rightarrow$ $n-1$ steps $\Rightarrow$ degree $n-1$ curve.

Two points: a line. Three: a parabola. Four: a cubic. Forty: also fine.

Writing It Down

From the Picture, Straight to the Loop

```
def de_casteljau(control: list[Vec2],
                 t: float) -> Vec2:
    pts = list(control)
    while len(pts) > 1:
        pts = [p + (q - p) * t
               for p, q in zip(pts, pts[1:])]
    return pts[0]
```

Read it against the picture:

From the Picture, Straight to the Loop

```
def de_casteljau(control: list[Vec2],
                 t: float) -> Vec2:
    pts = list(control)
    while len(pts) > 1:
        pts = [p + (q - p) * t
               for p, q in zip(pts, pts[1:])]
    return pts[0]
```

Read it against the picture:

- `while len(pts) > 1`
  "until one survivor"

From the Picture, Straight to the Loop

```python
def de_casteljau(control: list[Vec2],
                 t: float) -> Vec2:
    pts = list(control)
    while len(pts) > 1:
        pts = [p + (q - p) * t
               for p, q in zip(pts, pts[1:])]
    return pts[0]
```

Read it against the picture:

- `while len(pts) > 1`
  "until one survivor"
- `p + (q - p) * t`
  one lerp — Monday's line, verbatim

From the Picture, Straight to the Loop

```
def de_casteljau(control: list[Vec2],
                 t: float) -> Vec2:
    pts = list(control)
    while len(pts) > 1:
        pts = [p + (q - p) * t
               for p, q in zip(pts, pts[1:])]
    return pts[0]
```

Read it against the picture:

- `while len(pts) > 1`
  "until one survivor"
- `p + (q - p) * t`
  one lerp — Monday's line, verbatim
- `zip(pts, pts[1:])`
  consecutive pairs: n in, $n-1$ out

From the Picture, Straight to the Loop

```python
def de_casteljau(control: list[Vec2],
                 t: float) -> Vec2:
    pts = list(control)
    while len(pts) > 1:
        pts = [p + (q - p) * t
               for p, q in zip(pts, pts[1:])]
    return pts[0]
```

Read it against the picture:

- `while len(pts) > 1`
  "until one survivor"
- `p + (q - p) * t`
  one lerp — Monday's line, verbatim
- `zip(pts, pts[1:])`
  consecutive pairs: n in, $n-1$ out
- `list(control)`
  copy — never touch the caller's list

From the Picture, Straight to the Loop

```python
def de_casteljau(control: list[Vec2],
                 t: float) -> Vec2:
    pts = list(control)
    while len(pts) > 1:
        pts = [p + (q - p) * t
               for p, q in zip(pts, pts[1:])]
    return pts[0]
```

Read it against the picture:

- `while len(pts) > 1`
  "until one survivor"
- `p + (q - p) * t`
  one lerp — Monday's line, verbatim
- `zip(pts, pts[1:])`
  consecutive pairs: n in, $n-1$ out
- `list(control)`
  copy — never touch the caller's list

Four lines. That is the whole idea.

Every step is one list comprehension; the `while` runs it until nothing is left to interpolate. No degree appears anywhere in the code — which is exactly why it handles all of them.

Then the Other Two Lines Write Themselves

```
1 def quadratic_bezier(p0, p1, p2, t) -> Vec2:
2     return de_casteljau([p0, p1, p2], t)
3
4 def cubic_bezier(p0, p1, p2, p3, t) -> Vec2:
5     return de_casteljau([p0, p1, p2, p3], t)
```

Then the Other Two Lines Write Themselves

```
1 def quadratic_bezier(p0, p1, p2, t) -> Vec2:
2     return de_casteljau([p0, p1, p2], t)
3
4 def cubic_bezier(p0, p1, p2, p3, t) -> Vec2:
5     return de_casteljau([p0, p1, p2, p3], t)
```

- the algorithm lives in *one* place: `de_casteljau`

Then the Other Two Lines Write Themselves

```
def quadratic_bezier(p0, p1, p2, t) -> Vec2:
    return de_casteljau([p0, p1, p2], t)

def cubic_bezier(p0, p1, p2, p3, t) -> Vec2:
    return de_casteljau([p0, p1, p2, p3], t)
```

- the algorithm lives in *one* place: `de_casteljau`
- test with **1 to 5** control points. A real loop handles all of them. Four `if`s do not.

Then the Other Two Lines Write Themselves

```
def quadratic_bezier(p0, p1, p2, t) -> Vec2:
    return de_casteljau([p0, p1, p2], t)

def cubic_bezier(p0, p1, p2, p3, t) -> Vec2:
    return de_casteljau([p0, p1, p2, p3], t)
```

- the algorithm lives in *one* place: `de_casteljau`
- test with **1 to 5** control points. A real loop handles all of them. Four `if`s do not.
- one control point is fine: the curve is just that point. The loop runs **zero** times. An empty list is an error — raise.

The Tempting Wrong Way

The cubic has a closed formula too:

$$\mathbf{p}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\,\mathbf{P}_1 + 3(1-t)t^2\,\mathbf{P}_2 + t^3\mathbf{P}_3$$

It is correct — it is exactly what the loop computes. So why not just type it in?

The Tempting Wrong Way

The cubic has a closed formula too:

$$\mathbf{p}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\,\mathbf{P}_1 + 3(1-t)t^2\,\mathbf{P}_2 + t^3\mathbf{P}_3$$

It is correct — it is exactly what the loop computes. So why not just type it in?

you want	**with the formula**	**with the loop**
3 points	type 3 terms	pass 3 points
4 points	type 4 *different* terms	pass 4 points

The Tempting Wrong Way

The cubic has a closed formula too:

$$\mathbf{p}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\,\mathbf{P}_1 + 3(1-t)t^2\,\mathbf{P}_2 + t^3\mathbf{P}_3$$

It is correct — it is exactly what the loop computes. So why not just type it in?

you want	**with the formula**	**with the loop**
3 points	type 3 terms	pass 3 points
4 points	type 4 *different* terms	pass 4 points
10 points	look up 10 coefficients, type 10 terms; one typo, quietly wrong	pass 10 points

The Tempting Wrong Way

The cubic has a closed formula too:

$$\mathbf{p}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\,\mathbf{P}_1 + 3(1-t)t^2\,\mathbf{P}_2 + t^3\mathbf{P}_3$$

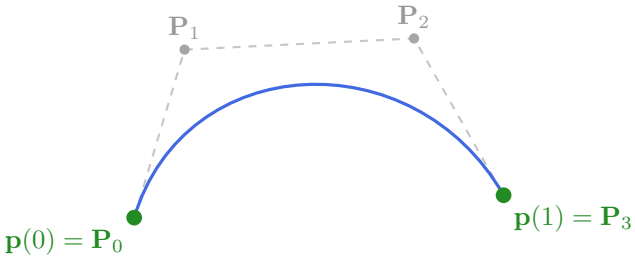
It is correct — it is exactly what the loop computes. So why not just type it in?

you want	**with the formula**	**with the loop**
3 points	type 3 terms	pass 3 points
4 points	type 4 *different* terms	pass 4 points
10 points	look up 10 coefficients, type 10 terms; one typo, quietly wrong	pass 10 points

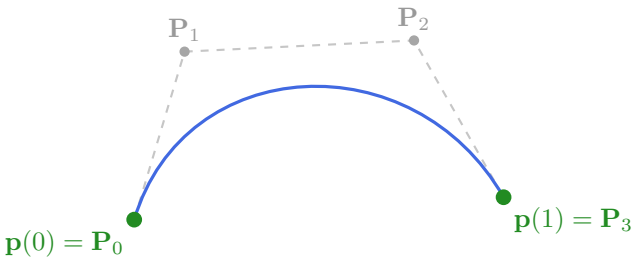
- The formula is **one degree, frozen.** The loop is **every degree**, and it finds the 3s for you.
- Need speed later? Use the formula then. **Write the loop first** — it is shorter anyway.

What the Handles Actually Do

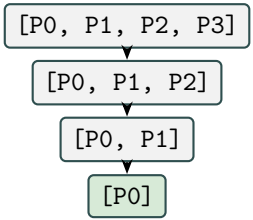
Property 1: The Ends Are Nailed Down



Property 1: The Ends Are Nailed Down

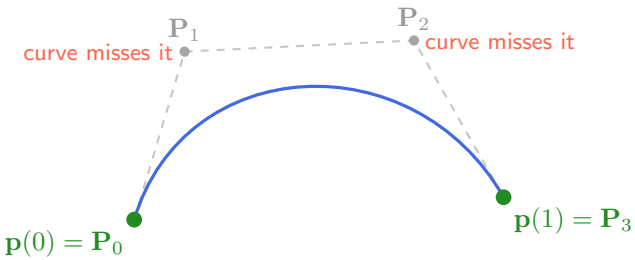


Watch the loop at $t = 0$:

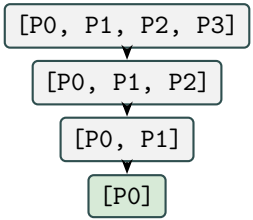


At $t = 0$ every lerp gives back its *left* point, so each step just drops the last entry. At $t = 1$, the *right* point: you end at $\mathbf{P}_3$.

Property 1: The Ends Are Nailed Down



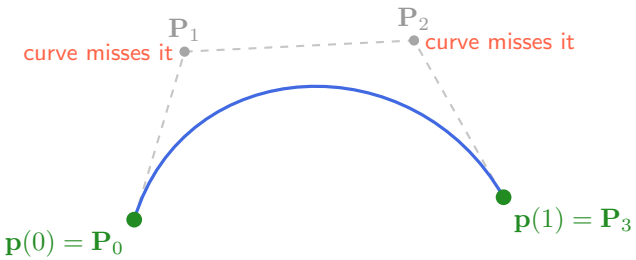
Watch the loop at $t = 0$:



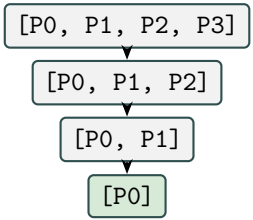
At $t = 0$ every lerp gives back its *left* point, so each step just drops the last entry. At $t = 1$, the *right* point: you end at $\mathbf{P}_3$.

$\mathbf{P}_1$ and $\mathbf{P}_2$ are **handles, not stops**. The curve leans toward them but usually does not pass through them.

Property 1: The Ends Are Nailed Down



Watch the loop at $t = 0$:

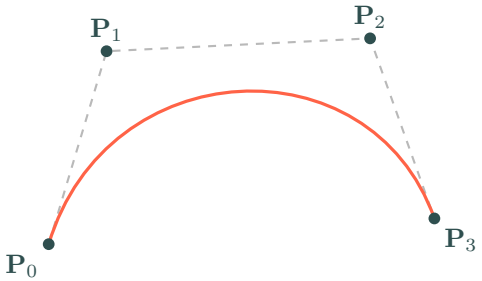


At $t = 0$ every lerp gives back its *left* point, so each step just drops the last entry. At $t = 1$, the *right* point: you end at $\mathbf{P}_3$.

$\mathbf{P}_1$ and $\mathbf{P}_2$ are **handles, not stops**. The curve leans toward them but usually does not pass through them.

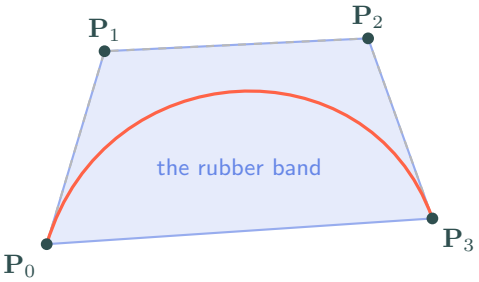
Payoff: to join two curves with no gap, make the last point of one the first point of the next.

Property 2: The Curve Stays Inside the Handles



Why: a lerp is a *mix* of its two inputs, $(1-t)\,\mathbf{P} + t\,\mathbf{Q}$, with both weights between 0 and 1. A mix of mixes is still a mix. So every curve point is a mix of the four handles — it lies *between* them.

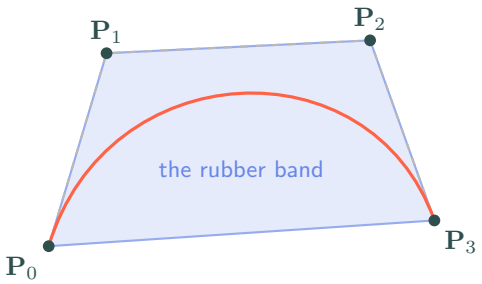
Property 2: The Curve Stays Inside the Handles



Why: a lerp is a *mix* of its two inputs, $(1-t)\,\mathbf{P} + t\,\mathbf{Q}$, with both weights between 0 and 1. A mix of mixes is still a mix. So every curve point is a mix of the four handles — it lies *between* them.

Picture: stretch a rubber band around the handles and let go. The curve never leaves that shape (the **convex hull**).

Property 2: The Curve Stays Inside the Handles

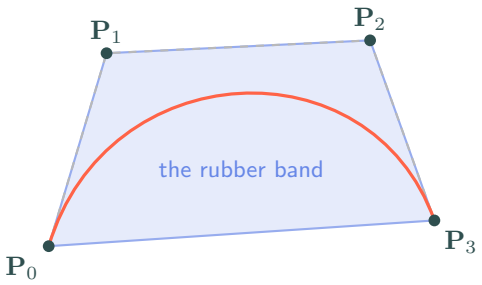


Why: a lerp is a *mix* of its two inputs, $(1-t)\,\mathbf{P} + t\,\mathbf{Q}$, with both weights between 0 and 1. A mix of mixes is still a mix. So every curve point is a mix of the four handles — it lies *between* them.

Picture: stretch a rubber band around the handles and let go. The curve never leaves that shape (the **convex hull**).

Payoff: all four handles off-screen $\Rightarrow$ the whole curve is off-screen. Skip it without drawing a single pixel.

Property 2: The Curve Stays Inside the Handles



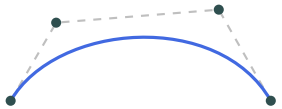
Why: a lerp is a *mix* of its two inputs, $(1-t)\,\mathbf{P} + t\,\mathbf{Q}$, with both weights between 0 and 1. A mix of mixes is still a mix. So every curve point is a mix of the four handles — it lies *between* them.

Picture: stretch a rubber band around the handles and let go. The curve never leaves that shape (the **convex hull**).

Payoff: all four handles off-screen $\Rightarrow$ the whole curve is off-screen. Skip it without drawing a single pixel.

(Same instinct later in the course: bounding volumes for ray tracing.)

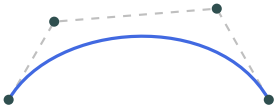
Property 3: The First Segment Is the Launch Direction



short handles

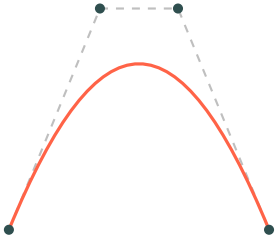
a gentle arc

Property 3: The First Segment Is the Launch Direction



short handles

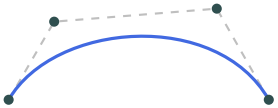
a gentle arc



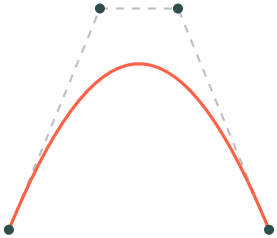
long handles

same endpoints, dramatic swoop

Property 3: The First Segment Is the Launch Direction



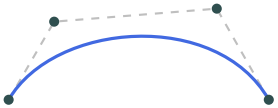
short handles
a gentle arc



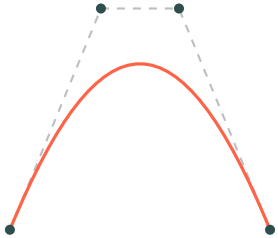
long handles
same endpoints, dramatic swoop

- the curve **leaves** $\mathbf{P}_0$ heading straight at $\mathbf{P}_1$, and **arrives** at $\mathbf{P}_3$ coming from $\mathbf{P}_2$

Property 3: The First Segment Is the Launch Direction



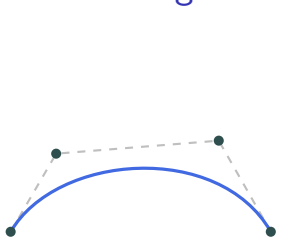
short handles
a gentle arc



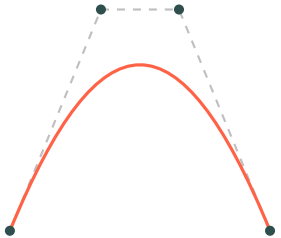
long handles
same endpoints, dramatic swoop

- the curve **leaves** $\mathbf{P}_0$ heading straight at $\mathbf{P}_1$, and **arrives** at $\mathbf{P}_3$ coming from $\mathbf{P}_2$
- *direction* of the handle sets the departure angle; *length* sets how long the curve obeys it

Property 3: The First Segment Is the Launch Direction



short handles
a gentle arc



long handles
same endpoints, dramatic swoop

- the curve **leaves** $\mathbf{P}_0$ heading straight at $\mathbf{P}_1$, and **arrives** at $\mathbf{P}_3$ coming from $\mathbf{P}_2$
- *direction* of the handle sets the departure angle; *length* sets how long the curve obeys it

This is the entire mental model a designer uses with a pen tool. Now you know why it behaves that way.

Drawing Them, Moving Along Them

Drawing with the Pen Tool

Wednesday's pipeline does not care what `f` is:

```
1  kite = lambda t: curves.cubic_bezier(a, b, c, d, t)
2
3  pts = sample_parametric(kite, 0.0, 1.0, 64)    # same sampler!
4  draw_polyline(img, pts, GOLD)                   # same drawer!
```

only this box is new

Sampling Cost, Honestly

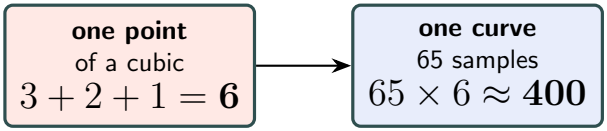
How much work is a Bézier curve? Count the lerps.

one point
of a cubic
$3 + 2 + 1 = \mathbf{6}$

lerps per `de_casteljau` call

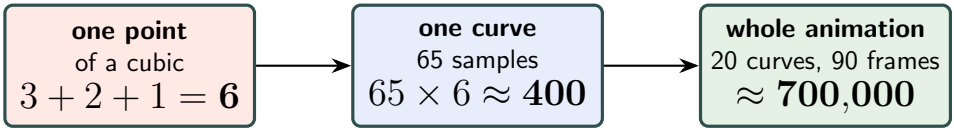
Sampling Cost, Honestly

How much work is a Bézier curve? Count the lerps.



Sampling Cost, Honestly

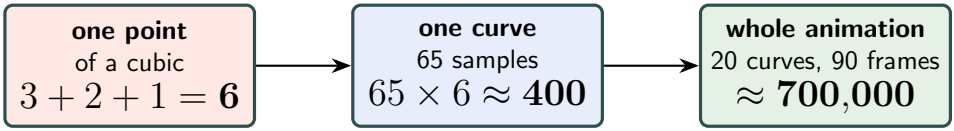
How much work is a Bézier curve? Count the lerps.



Python does millions of these a second. **You will not notice.**

Sampling Cost, Honestly

How much work is a Bézier curve? Count the lerps.

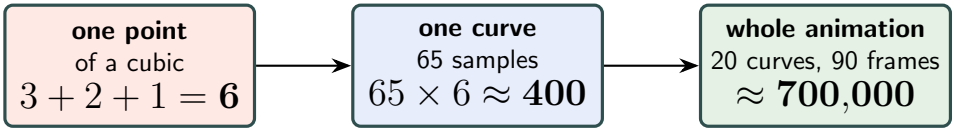


Python does millions of these a second. **You will not notice.**

One tip: if the curve itself does not move, sample it *once* and reuse the point list every frame.

Sampling Cost, Honestly

How much work is a Bézier curve? Count the lerps.

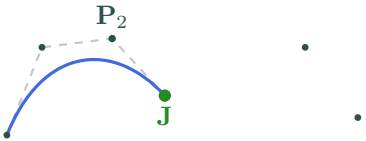


Python does millions of these a second. **You will not notice.**

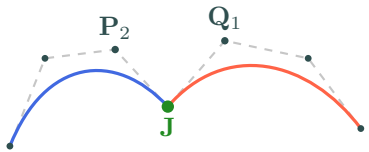
One tip: if the curve itself does not move, sample it *once* and reuse the point list every frame.

Measure before you optimise anything in this course.

Chaining: Long Shapes from Short Curves



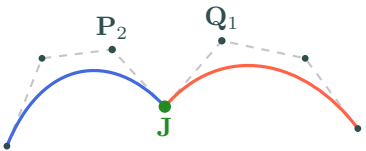
Chaining: Long Shapes from Short Curves



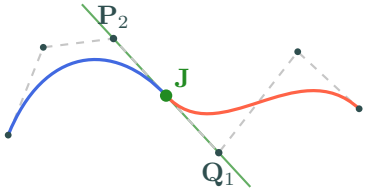
Position matches: no gap, but a **corner**.

- **Position:** curve 2 starts at $\mathbf{J}$, exactly where curve 1 ends. Endpoints are exact (Property 1), so there is never a gap.

Chaining: Long Shapes from Short Curves



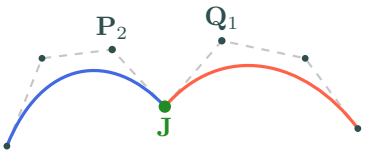
Position matches: no gap, but a **corner**.



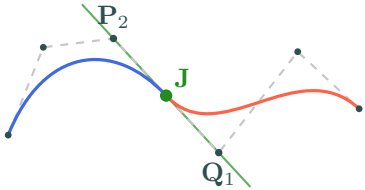
Position + direction match: **smooth**.

- **Position:** curve 2 starts at $\mathbf{J}$, exactly where curve 1 ends. Endpoints are exact (Property 1), so there is never a gap.
- **Direction:** curve 1 *arrives* along $\mathbf{P}_2 \to \mathbf{J}$; curve 2 *leaves* along $\mathbf{J} \to \mathbf{Q}_1$ (Property 3). Put $\mathbf{P}_2$, $\mathbf{J}$, $\mathbf{Q}_1$ on **one straight line** and the two directions agree.

Chaining: Long Shapes from Short Curves



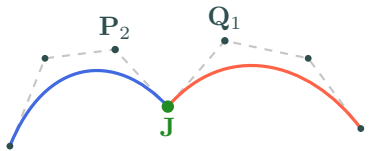
Position matches: no gap, but a **corner**.



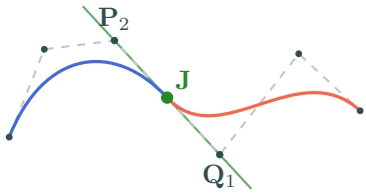
Position + direction match: **smooth**.

- **Position:** curve 2 starts at $\mathbf{J}$, exactly where curve 1 ends. Endpoints are exact (Property 1), so there is never a gap.
- **Direction:** curve 1 *arrives* along $\mathbf{P}_2 \to \mathbf{J}$; curve 2 *leaves* along $\mathbf{J} \to \mathbf{Q}_1$ (Property 3). Put $\mathbf{P}_2$, $\mathbf{J}$, $\mathbf{Q}_1$ on **one straight line** and the two directions agree.
- **Why care:** a corner in a letter looks broken; a corner in a flight path makes the kite turn *instantly*.

Chaining: Long Shapes from Short Curves



Position matches: no gap, but a **corner**.



Position + direction match: **smooth**.

- **Position:** curve 2 starts at $\mathbf{J}$, exactly where curve 1 ends. Endpoints are exact (Property 1), so there is never a gap.
- **Direction:** curve 1 *arrives* along $\mathbf{P}_2 \to \mathbf{J}$; curve 2 *leaves* along $\mathbf{J} \to \mathbf{Q}_1$ (Property 3). Put $\mathbf{P}_2$, $\mathbf{J}$, $\mathbf{Q}_1$ on **one straight line** and the two directions agree.
- **Why care:** a corner in a letter looks broken; a corner in a flight path makes the kite turn *instantly*.

Every letter you are reading is cubics chained like this: **many short pieces**, not one high-degree curve.

The Same Curve, Now as a Flight Path

```
FRAMES = 120
pts = sample_parametric(kite, 0.0, 1.0, 64)

def render_frame(i):
    u = i / (FRAMES - 1)          # 0 -> 1
    img = Canvas(W, H, SKY)
    draw_polyline(img, pts, FAINT)      # path
    p = curves.cubic_bezier(a, b, c, d, u)
    painter.fill_circle(img,            # kite
        round(p.x), round(p.y), 7, GOLD)
    return img
```

Wednesday's loop, **one line changed**:

The Same Curve, Now as a Flight Path

```
FRAMES = 120
pts = sample_parametric(kite, 0.0, 1.0, 64)

def render_frame(i):
    u = i / (FRAMES - 1)          # 0 -> 1
    img = Canvas(W, H, SKY)
    draw_polyline(img, pts, FAINT)      # path
    p = curves.cubic_bezier(a, b, c, d, u)
    painter.fill_circle(img,            # kite
        round(p.x), round(p.y), 7, GOLD)
    return img
```

Wednesday's loop, **one line changed**:

- **shape**: sample the path once, draw it every frame

The Same Curve, Now as a Flight Path

```
FRAMES = 120
pts = sample_parametric(kite, 0.0, 1.0, 64)

def render_frame(i):
    u = i / (FRAMES - 1)          # 0 -> 1
    img = Canvas(W, H, SKY)
    draw_polyline(img, pts, FAINT)      # path
    p = curves.cubic_bezier(a, b, c, d, u)
    painter.fill_circle(img,            # kite
        round(p.x), round(p.y), 7, GOLD)
    return img
```

Wednesday's loop, **one line changed**:

- **shape**: sample the path once, draw it every frame
- **position**: one point, at u

The Same Curve, Now as a Flight Path

```
FRAMES = 120
pts = sample_parametric(kite, 0.0, 1.0, 64)

def render_frame(i):
    u = i / (FRAMES - 1)          # 0 -> 1
    img = Canvas(W, H, SKY)
    draw_polyline(img, pts, FAINT)      # path
    p = curves.cubic_bezier(a, b, c, d, u)
    painter.fill_circle(img,            # kite
        round(p.x), round(p.y), 7, GOLD)
    return img
```

Wednesday's loop, **one line changed**:

- **shape**: sample the path once, draw it every frame
- **position**: one point, at u
- **time**: u comes from the frame number

The Same Curve, Now as a Flight Path

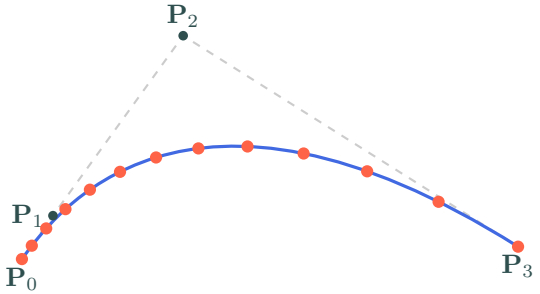
```
FRAMES = 120
pts = sample_parametric(kite, 0.0, 1.0, 64)

def render_frame(i):
    u = i / (FRAMES - 1)          # 0 -> 1
    img = Canvas(W, H, SKY)
    draw_polyline(img, pts, FAINT)      # path
    p = curves.cubic_bezier(a, b, c, d, u)
    painter.fill_circle(img,            # kite
        round(p.x), round(p.y), 7, GOLD)
    return img
```

Wednesday's loop, **one line changed**:

- **shape**: sample the path once, draw it every frame
- **position**: one point, at u
- **time**: u comes from the frame number

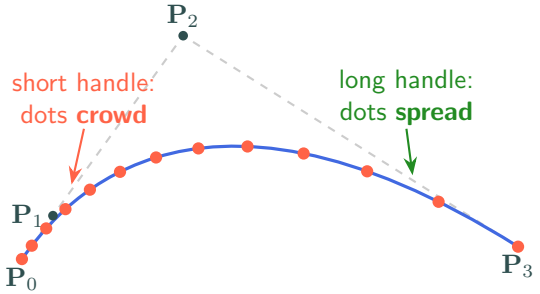
Bonus: $\mathbf{p}(u + 0.01) - \mathbf{p}(u)$ points the way the kite is flying. Normalize it, and the kite can face forward.

One More Honest Subtlety — Same One as Wednesday



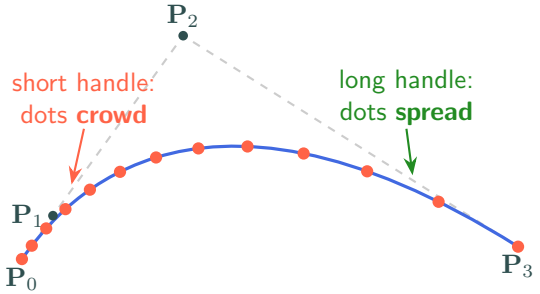
Equal **increments** of t are **not** equal **distances** along the curve.
Thirteen dots, $t = 0, \frac{1}{12}, \frac{2}{12}, \ldots, 1$. Same t-spacing, very different distances.

One More Honest Subtlety — Same One as Wednesday



Equal **increments** of t are **not** equal **distances** along the curve.
Thirteen dots, $t = 0, \frac{1}{12}, \frac{2}{12}, \ldots, 1$. Same t-spacing, very different distances.

One More Honest Subtlety — Same One as Wednesday



Equal **increments** of t are **not** equal **distances** along the curve.
Thirteen dots, $t = 0, \frac{1}{12}, \frac{2}{12}, \ldots, 1$. Same t-spacing, very different distances.

- **Drawing:** does not matter. The shape is right either way.
- **Motion:** the kite *speeds up* where the dots spread out. Often that looks good. If it does not, the fix is real work — not today.

Your Turn

Where This Goes Next

- **surfaces**: run de Casteljau in two directions and a curve becomes a *patch* — how car bodies were designed before anyone said "mesh"

Where This Goes Next

- **surfaces**: run de Casteljau in two directions and a curve becomes a *patch* — how car bodies were designed before anyone said "mesh"
- **Week 10 onward**: everything becomes triangles. Curves get *tessellated* — sampled — exactly as you did today, just on a GPU

Where This Goes Next

- **surfaces**: run de Casteljau in two directions and a curve becomes a *patch* — how car bodies were designed before anyone said "mesh"
- **Week 10 onward**: everything becomes triangles. Curves get *tessellated* — sampled — exactly as you did today, just on a GPU
- **easing**: a 1-D Bézier over *time* is the "ease-in-out" behind every polished UI animation

Where This Goes Next

- **surfaces**: run de Casteljau in two directions and a curve becomes a *patch* — how car bodies were designed before anyone said "mesh"
- **Week 10 onward**: everything becomes triangles. Curves get *tessellated* — sampled — exactly as you did today, just on a GPU
- **easing**: a 1-D Bézier over *time* is the "ease-in-out" behind every polished UI animation
- **the real lesson**: repeated linear interpolation buys curvature. That trick reappears in shading, texture filtering, and animation blending all term

Before Monday

1. Implement `de_casteljau`. Then check it by hand:

```
mid = cubic_bezier(p0, p1, p2, p3, 0.5)
assert cubic_bezier(p0, p1, p2, p3, 0.0) == p0
assert de_casteljau([p0], 0.7) == p0   # degree 0
```

2. **Animate something along a curve.** You have every tool: lines, circles, ellipses, Béziers, thick strokes, fills, frames, ffmpeg. Start early — rendering 120 frames is not instant

3. Sanity-check that the video actually plays. A folder of 120 PPMs is not an animation

Exit Ticket

Today's questions *(2 minutes)*

(a) `de_casteljau` is given **5** control points. How many lerps does it do in the *first* step, and how many steps in total?

(b) A cubic has control points $\mathbf{P}_0 \ldots \mathbf{P}_3$. Which of them does the curve pass through *exactly*?

CS 453 — Exit Ticket #6

Name: _____

Roll no: _____

(a) _____

(b) _____

hand in on your way out

Monday: we stop moving points one at a time.
Matrices: transform *everything* at once.