GDC 09

learn
network
inspire

www.GDConf.com

Game Developers Conference®

March 23-27, 2009 | Moscone Center, San Francisco

Interpolation and Splines

Squirrel Eiserloh

Director
TrueThought LLC

Squirrel@Eiserloh.net
Squirrel@TrueThought.com

Overview

- Averaging and Blending
- Interpolation
- Parametric Equations
- Parametric Curves and Splines

  including:

  - Bezier splines (linear, quadratic, cubic)
  - Cubic Hermite splines
  - Catmull-Rom splines
  - Cardinal splines
  - Kochanek–Bartels splines
  - B-splines

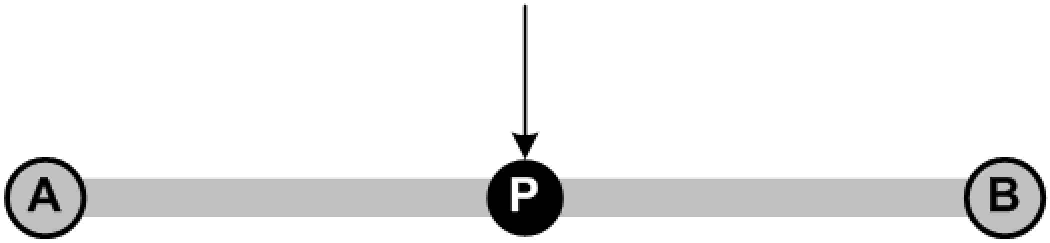
Averaging and Blending

Averaging and Blending

- First, we start off with the basics.
- I mean, really basic.
- Let's go back to grade school.

- How do you average two numbers together?

$$(A + B) / 2$$



Averaging and Blending

» Let’s change that around a bit.

(**A** + **B**) / 2

becomes

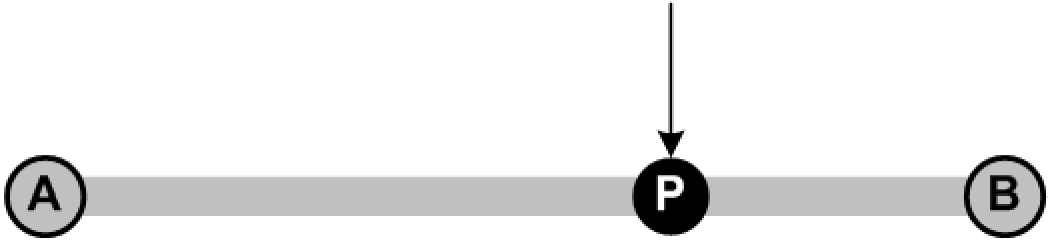
(.5 * **A**) + (.5 * **B**)

i.e. “half of **A**, half of **B**”, or “a blend of **A** and **B**”

Averaging and Blending

- We can, of course, also blend A and B unevenly (with different **weights**):

(.35 * A) + (.65 * B)

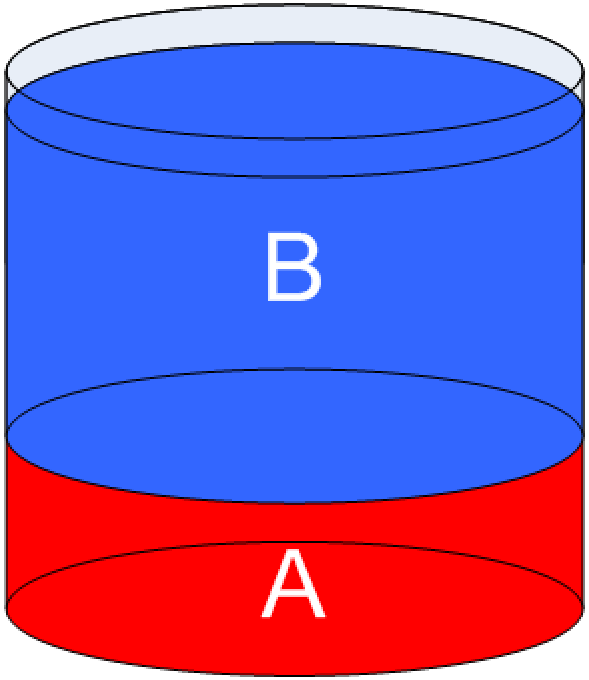


- In this case, we are blending “35% of A with 65% of B”.
- Can use any blend weights we want, as long as they add up to 1.0 (100%).

Averaging and Blending

- Like making up a bottle of liquid by mixing two different fluids together.

(we always fill the glass 100%)



Averaging and Blending

» So if we try to generalize here, we could say:

$$(s * A) + (t * B)$$

» ...where **s** is “how much of **A**” we want, and **t** is “how much of **B**” we want

» ...and **s** + **t** = 1.0 (really, **s** is just 1-**t**)

so: $((1-t) * A) + (t * B)$

Which means we can control the balance of the entire blend by changing just one number: **t**

Averaging and Blending

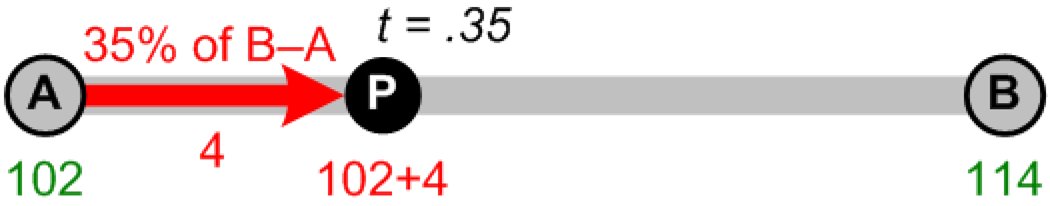
- There are two ways of thinking about this (and a formula for each):

- #1: “Blend some of **A** with some of **B**”

$$(\mathbf{s} * \mathbf{A}) + (\mathbf{t} * \mathbf{B}) \quad \leftarrow \textit{where } \boldsymbol{s} = 1\text{-}\boldsymbol{t}$$

- #2: “Start with **A**, and then add some amount of the distance from **A** to **B**”

$$\mathbf{A} + \mathbf{t}*(\mathbf{B} - \mathbf{A})$$



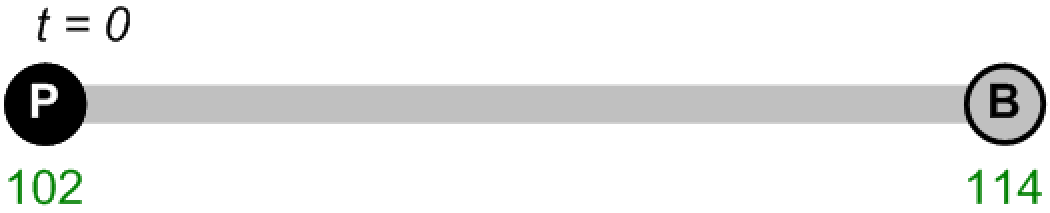
Averaging and Blending

» In both cases, the result of our blend is just plain “**A**” if **t**=0;

i.e. if we don’t want **any** of **B**.

(1.00 * **A**) + (0.00 * B) = **A**

or: **A** + 0.00*(B – A) = **A**



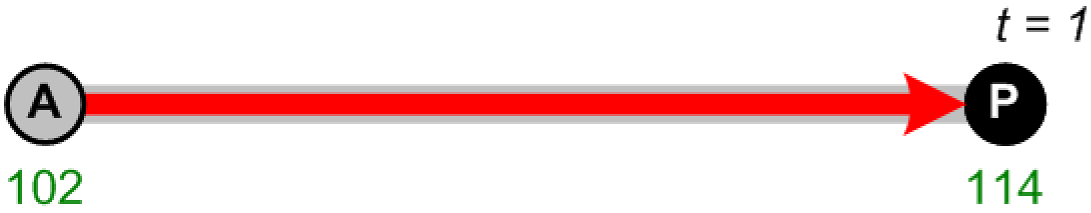
Averaging and Blending

» Likewise, the result of our blend is just plain "**B**" if **t**=1; i.e. if we don't want any of **A**.

(0.00 * A) + (1.00 * **B**) = **B**

or: **A** + 1.00*(B – **A**) =

A + **B** – A = **B**



Averaging and Blending

» However we choose to think about it, there’s a single “knob”, called **t**, that we are tweaking to get the blend of **A** and **B** that we want.

Blending Compound Data

Blending Compound Data

- We can blend more than just numbers.
- Blending 2D and 3D vectors, for example, is a cinch:
- Just blend each component (x,y,z) separately, at the same time.

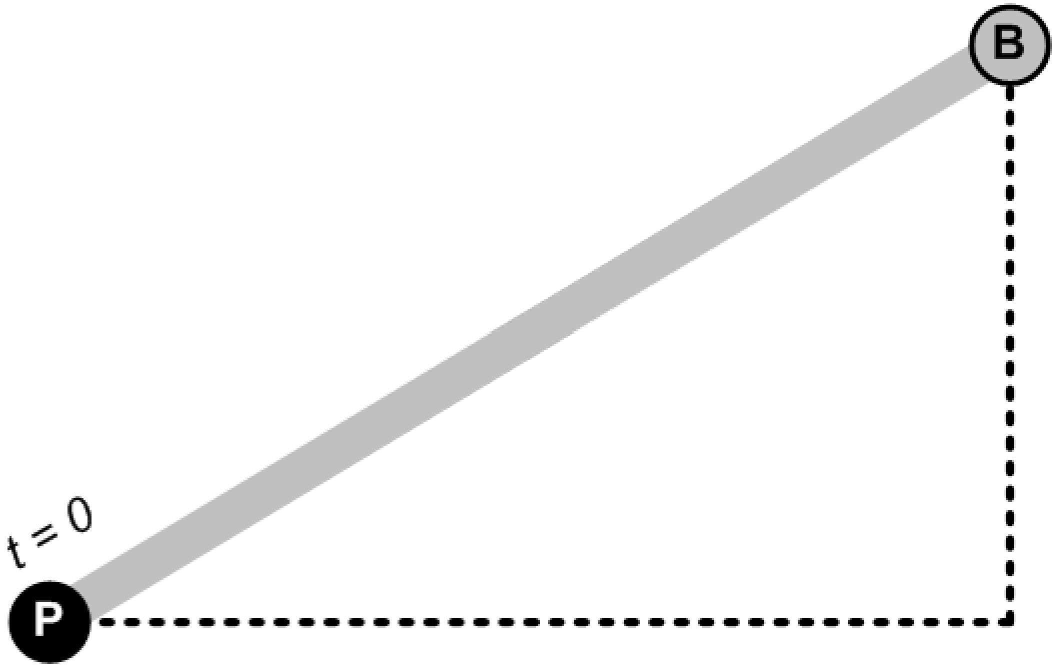
$$\mathbf{P} = (s * \mathbf{A}) + (t * \mathbf{B}) \quad \leftarrow \textit{where } s = 1-t$$

is equivalent to:

$$P_x = (s * A_x) + (t * B_x)$$
$$P_y = (s * A_y) + (t * B_y)$$
$$P_z = (s * A_z) + (t * B_z)$$

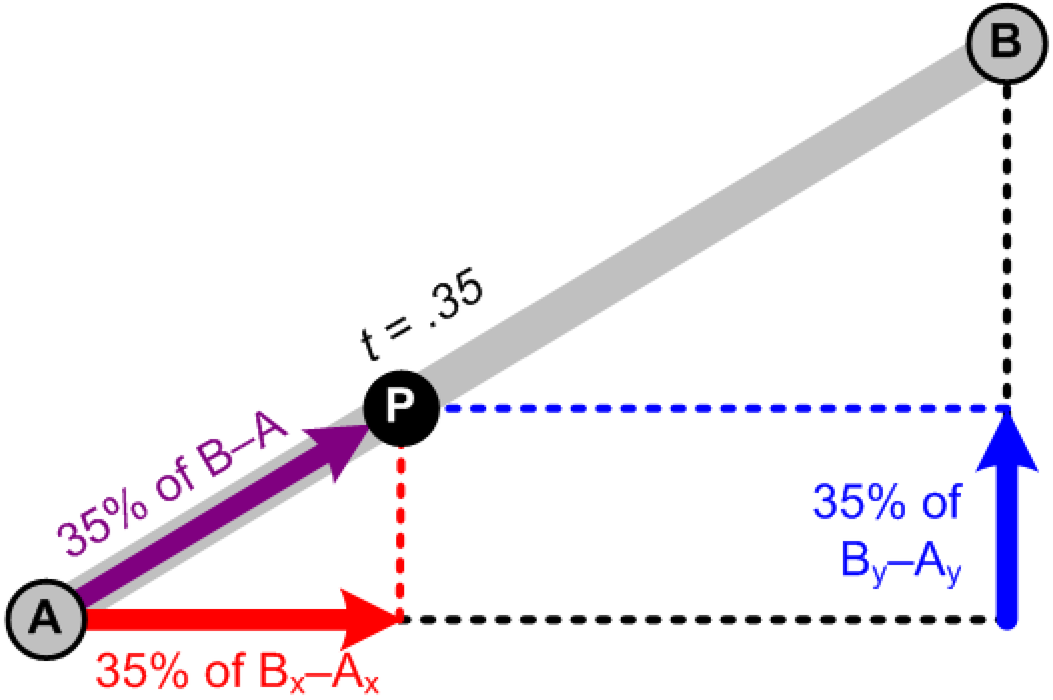
Blending Compound Data

(such as Vectors)



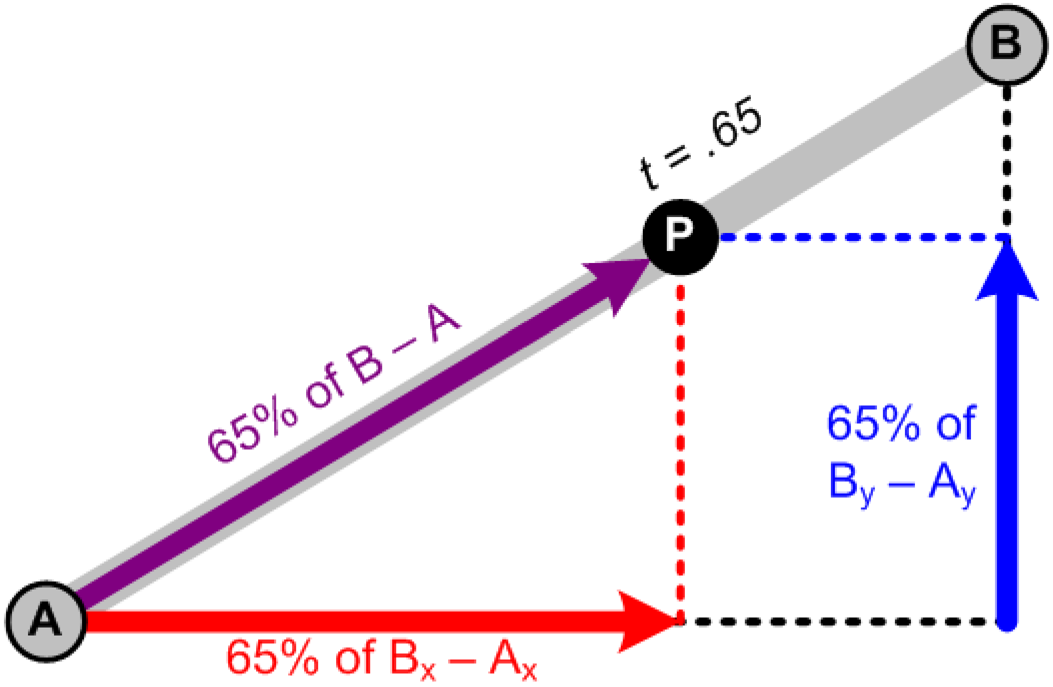
Blending Compound Data

(such as Vectors)



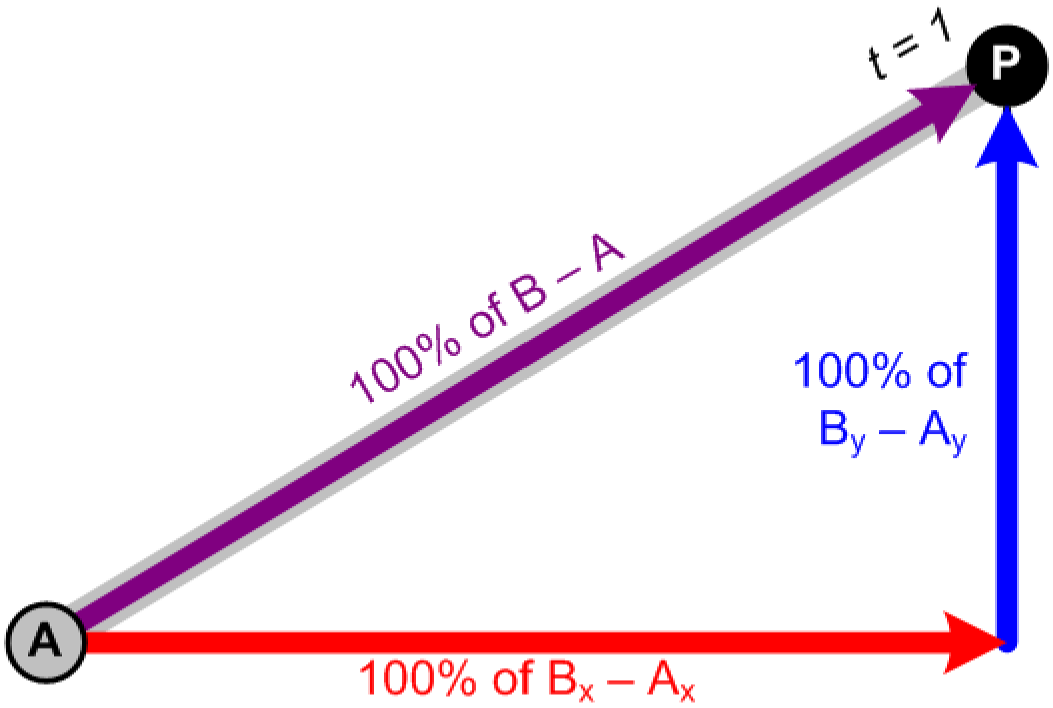
Blending Compound Data

(such as Vectors)



Blending Compound Data

(such as Vectors)



Blending Compound Data

- Need to be careful, though!
- Not all compound data types will blend correctly with this sort of (blend-the-components) approach.
- Examples: Color RGBs, Euler angles (yaw/pitch/roll), Matrices, Quaternions…

…in fact, there are a bunch that won’t.

Blending Compound Data

- Here’s an RGB color example:

- If A is **RGB(255, 0, 0)** – **bright red**

...and B is **RGB(0, 255, 0)** – **bright green**

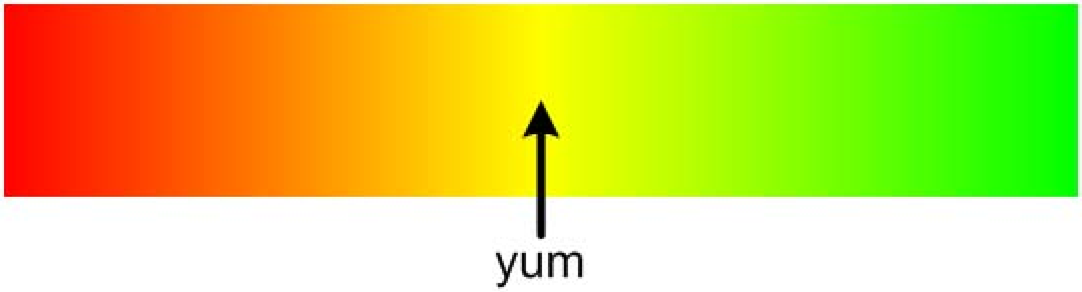
- Blending the two (with t = 0.5) gives:

  RGB(127, 127, 0)

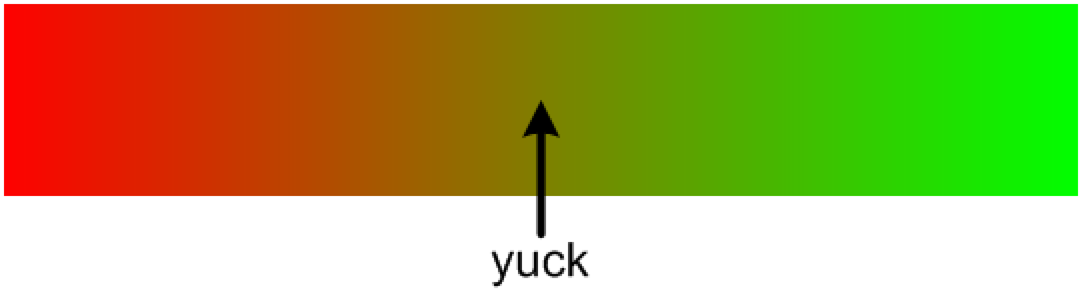
  ...which is a **dull, swampy color**. Yuck.

Blending Compound Data

» What we **wanted** was this:



...and what we got instead was this:



Blending Compound Data

- » For many compound classes, like RGB, you may need to write your own Interpolate() method that “does the right thing”, whatever that may be.

- » Jim will talk later about what happens when you try to interpolate Euler Angles (yaw/pitch/roll), Matrices, and Quaternions using this simple “naive” approach of blending the components.

Interpolation

Interpolation

- **Interpolation** is just changing blend weights *over time*. Also called "**Lerp**".
- i.e. Turning the knob (t) progressively, not just setting it to some position.
- Often we crank slowly from t=0 to t=1.

Interpolation

- Since games are generally frame-based, we usually have some Update() method that gets called, in which we have to decide what we're supposed to look like at this instant in time.
- There are two main ways of approaching this when we're interpolating:

- #1: Blend from **A** to **B** over the course of several frames (**parametric evaluation**);

- #2: Blend one step from wherever-I'm-at now to wherever-I'm-going (**numerical integration**).

Interpolation

- Games generally need to use both.
- Most physics tends to use method #2 (numerical integration). Erin will talk more about this at the end of the day.
- Many other systems, however, use method #1 (parametric evaluation).

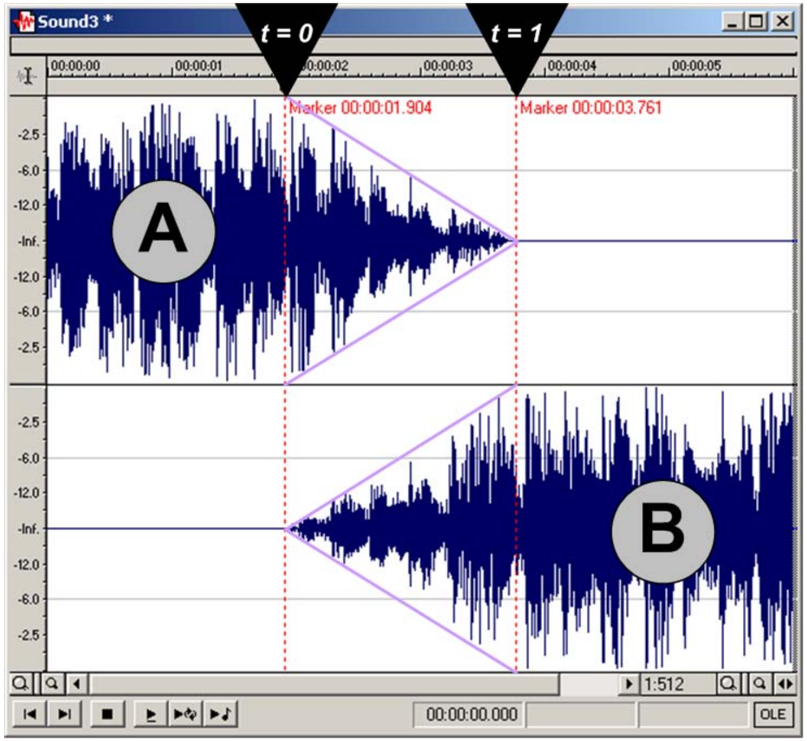
(More on that in a moment)

Interpolation

- » We use “lerping” all the time, under different names.

For example:

- » an Audio crossfade

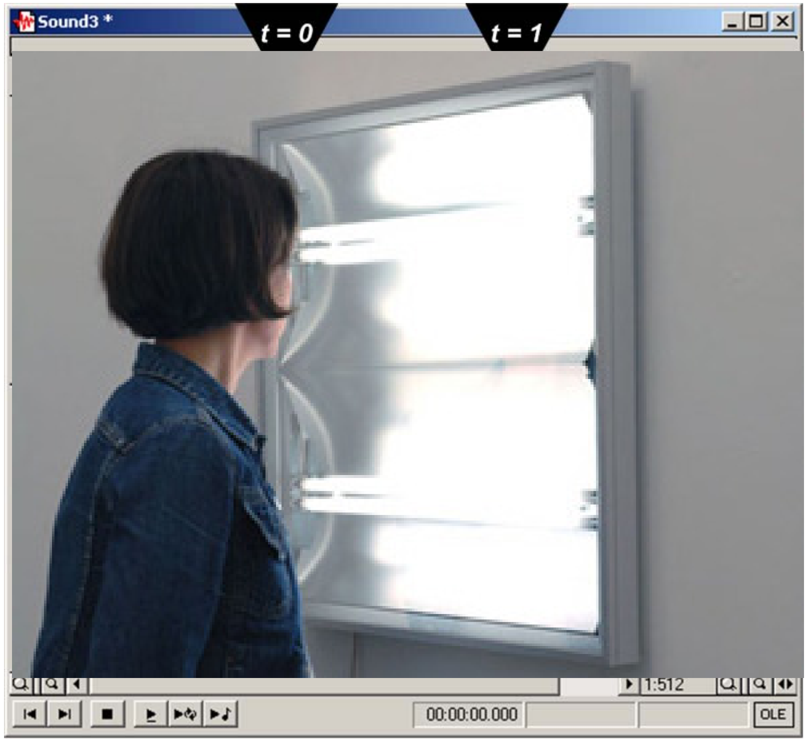


Interpolation

» We use “lerping” all the time, under different names.

For example:

» an Audio crossfade
» fading up lights



Interpolation

- » We use “lerping” all the time, under different names.

  For example:

- » an Audio crossfade
- » fading up lights
- » or this cheesy PowerPoint effect.

Interpolation

Basically, whenever we do any sort of **blend over time**, we’re lerping.

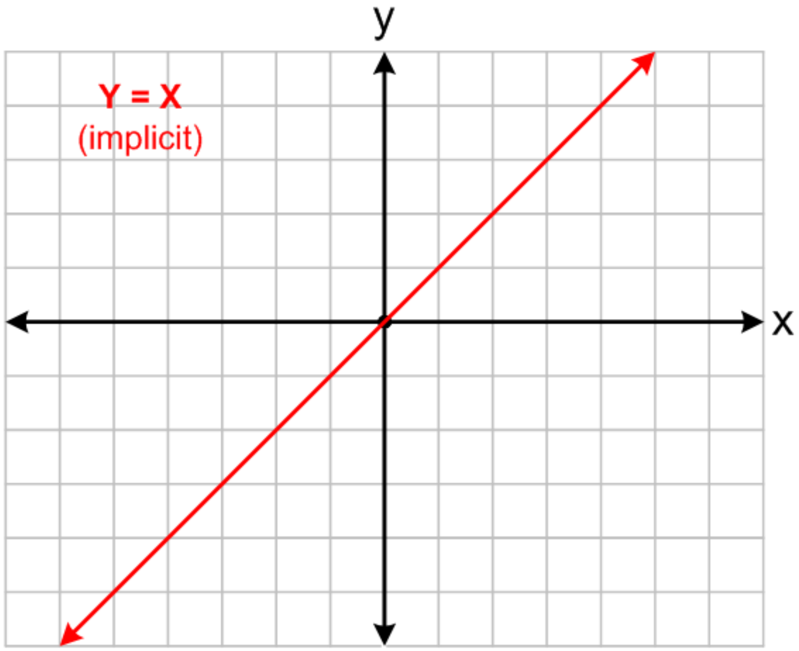
“That’s my cue to go get a margarita.”
-Squirrel’s wife

Implicit Equations

Sweetness…

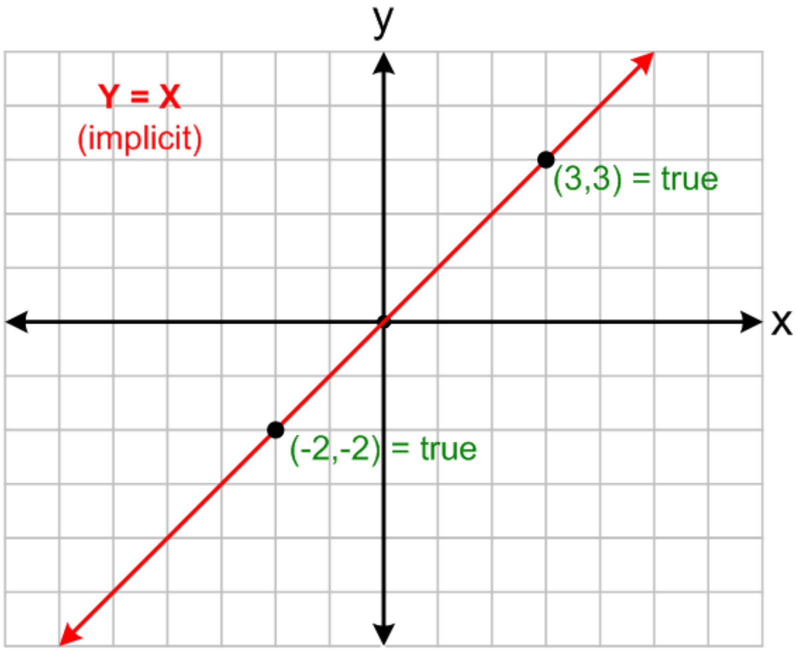
I loves me some math!

Implicit Equations



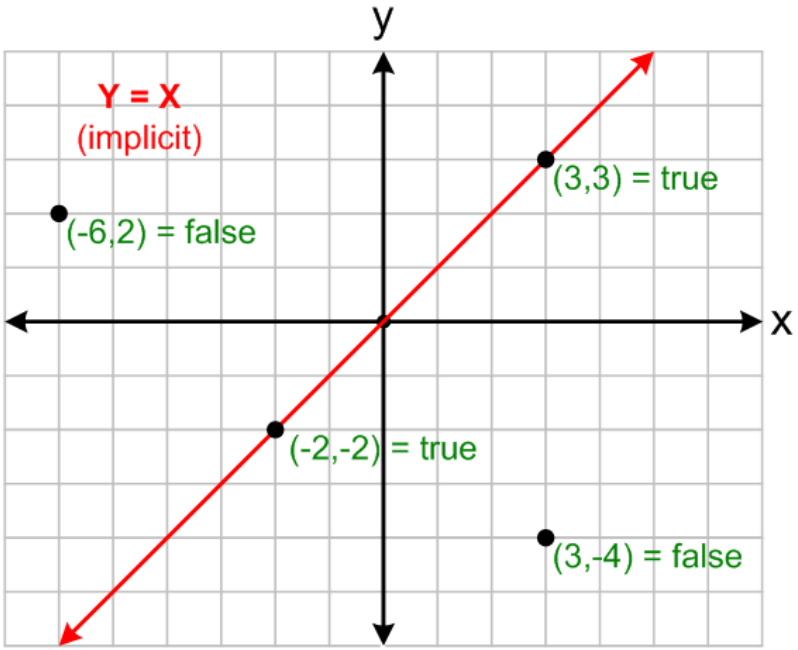
Implicit equations define what is, and isn’t, included in a set of points (a “locus”).

Implicit Equations



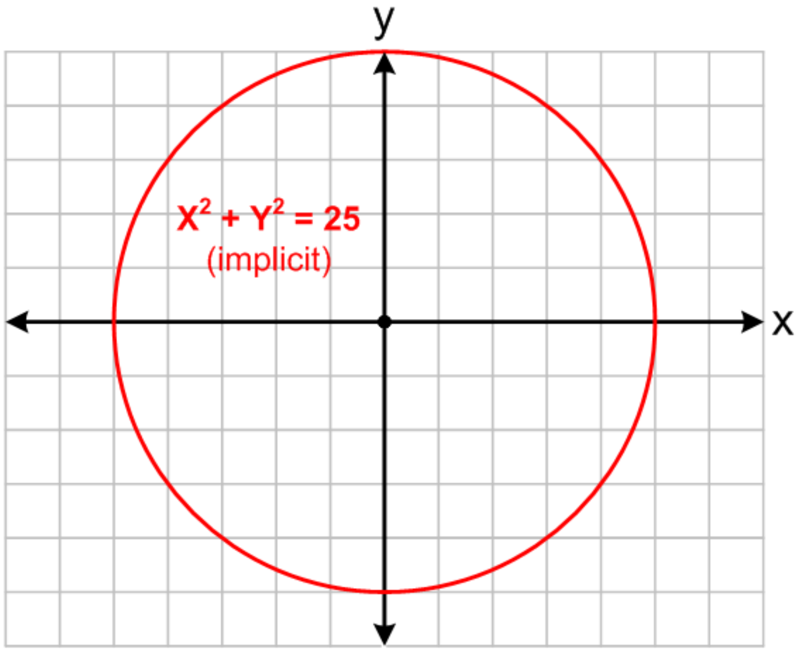
If the equation is TRUE for some x and y, then the point (x,y) is included on the line.

Implicit Equations



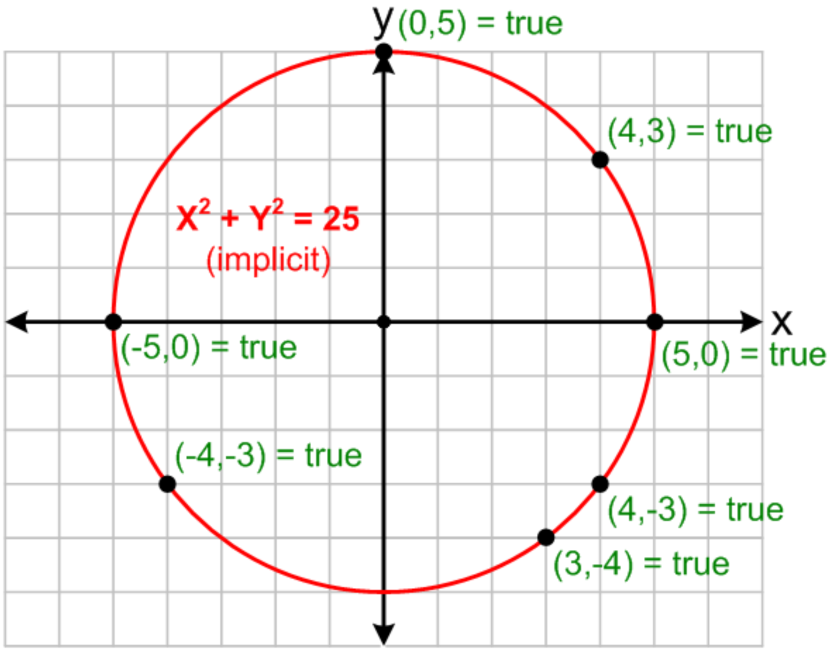
If the equation is FALSE for some x and y, then the point (x,y) is NOT included on the line.

Implicit Equations



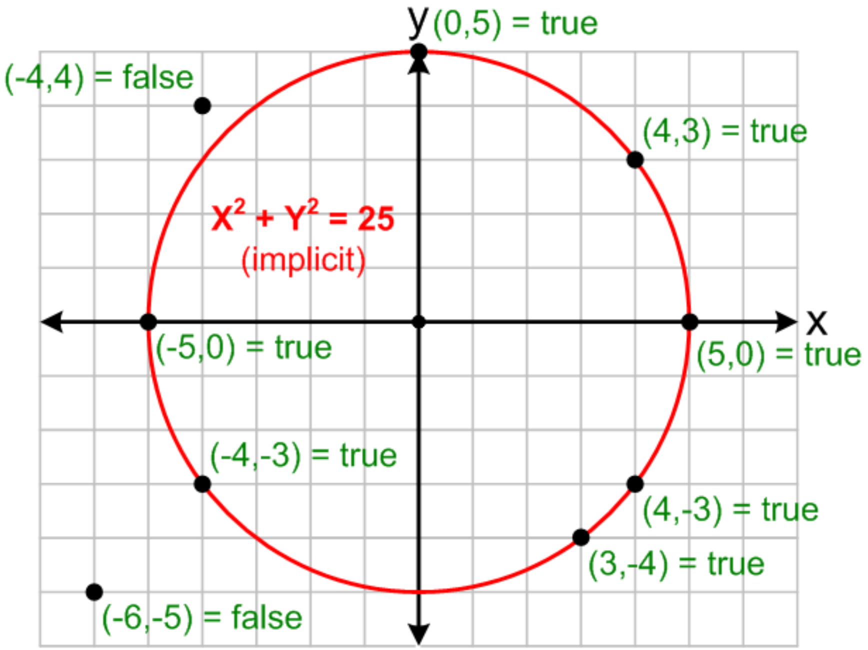
Here, the equation $X^2 + Y^2 = 25$ defines a "locus" of all the points within 5 units of the origin.

Implicit Equations



If the equation is TRUE for some x and y, then the point (x,y) is included on the circle.

Implicit Equations



If the equation is FALSE for some x and y,
then the point (x,y) is NOT included on the circle.

Parametric Equations

- A **parametric equation** is one that has been rewritten so that it has one clear "input" parameter (variable) that everything else is based in terms of.

- In other words, a parametric equation is basically **anything you can hook up to a single knob**. It's a formula that you can feed in a single number (the "knob" value, "t", usually from 0 to 1), and the formula gives back the appropriate value for that particular "t".

  Think of it as a function that takes a float and returns... whatever (a position, a color, an orientation, etc.):

```
someComplexData ParametricEquation( float t );
```

Parametric Equations

- Essentially:

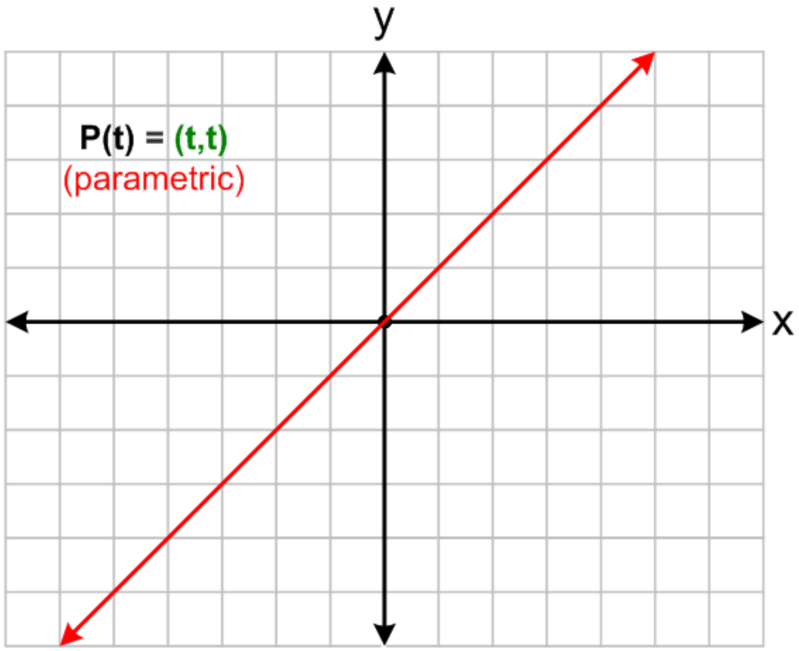
  P(t) = some formula with “t” in it

  ...as t changes, P changes
  (P depends upon t)

  P(t) can return any kind of value; whatever we want to interpolate, for instance.

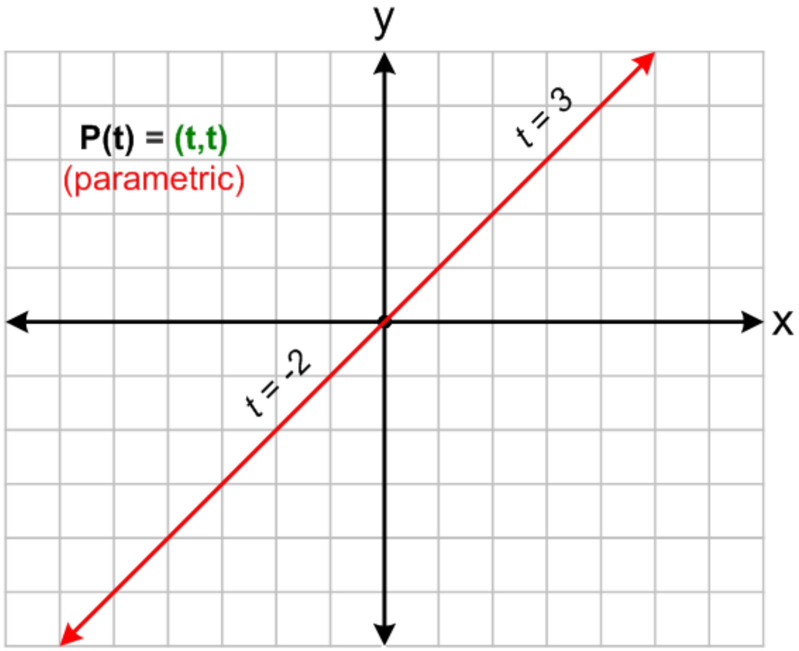
  - Position (2D, 3D, etc.)
  - Orientation
  - Scale
  - Alpha
  - etc.

Parametric Equations



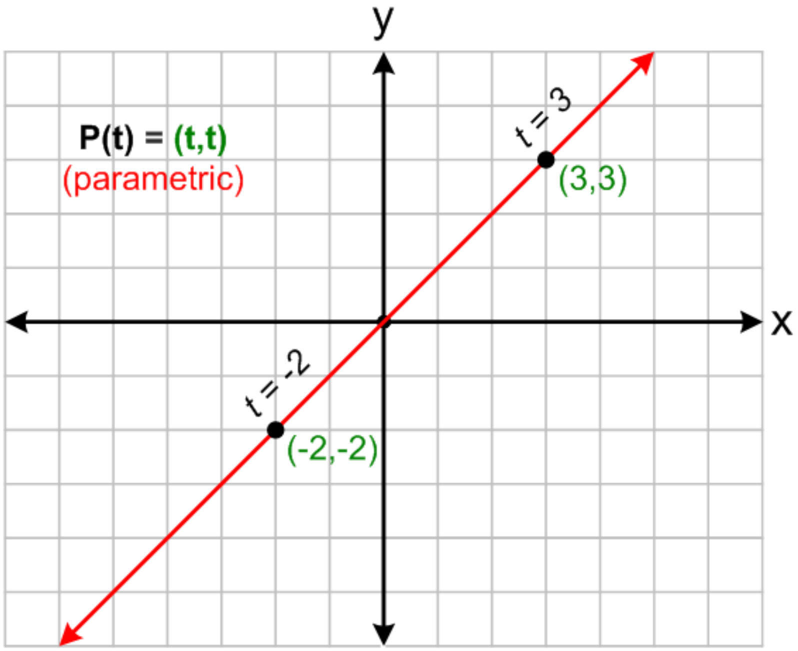
Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

Parametric Equations



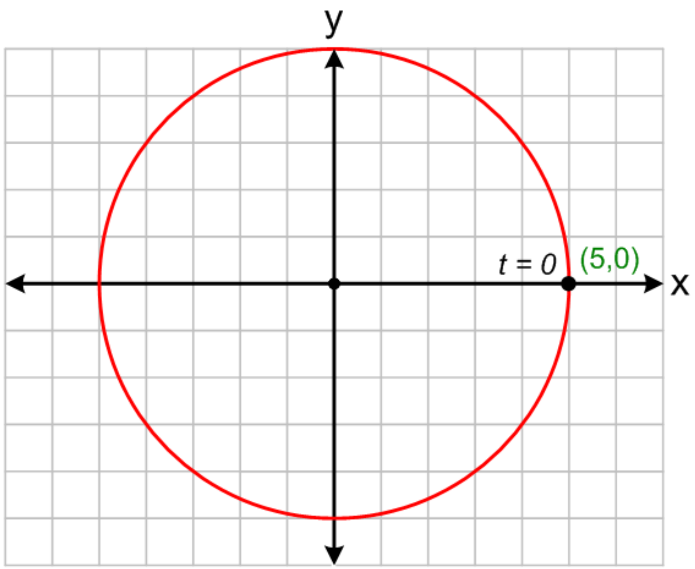
Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

Parametric Equations



Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

Parametric Equations

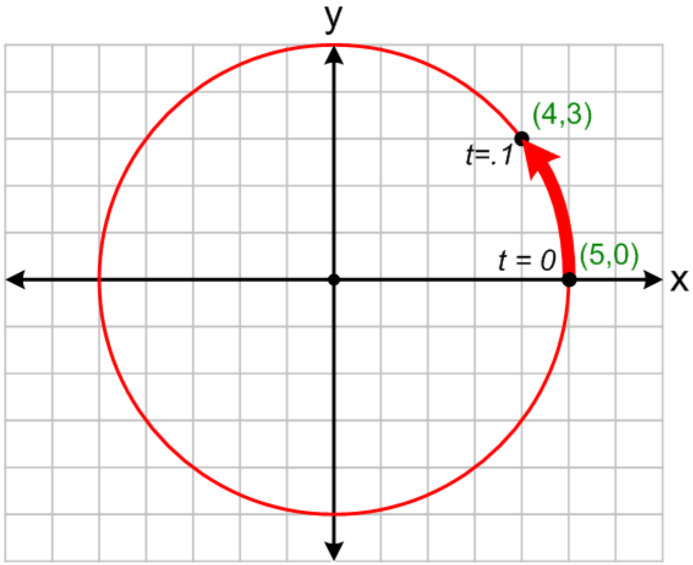


$P_x = 5 * \cos(2\pi * t)$
$P_y = 5 * \sin(2\pi * t)$

(parametric)

Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

Parametric Equations

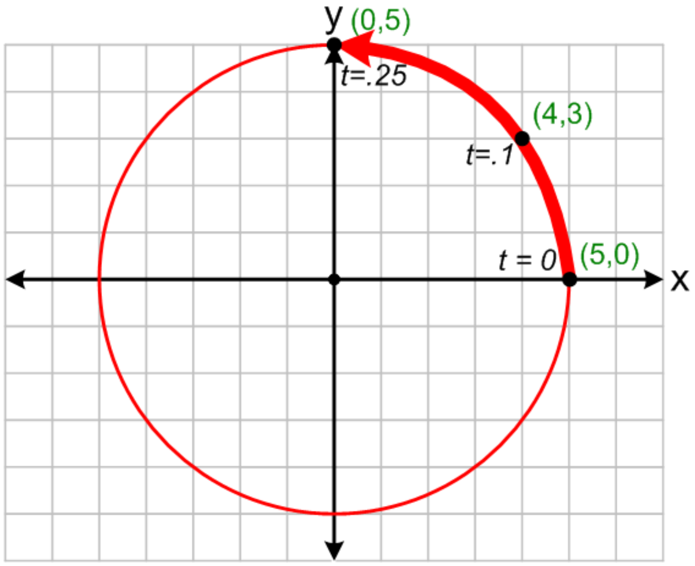


$P_x = 5 * \cos(2\pi * t)$
$P_y = 5 * \sin(2\pi * t)$

(parametric)

Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

Parametric Equations

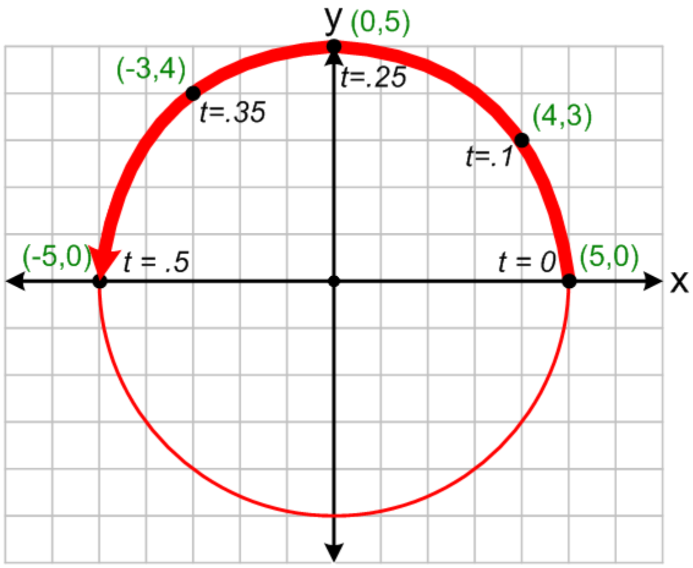


$P_x = 5 * \cos(2\pi * t)$
$P_y = 5 * \sin(2\pi * t)$

(parametric)

Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

Parametric Equations

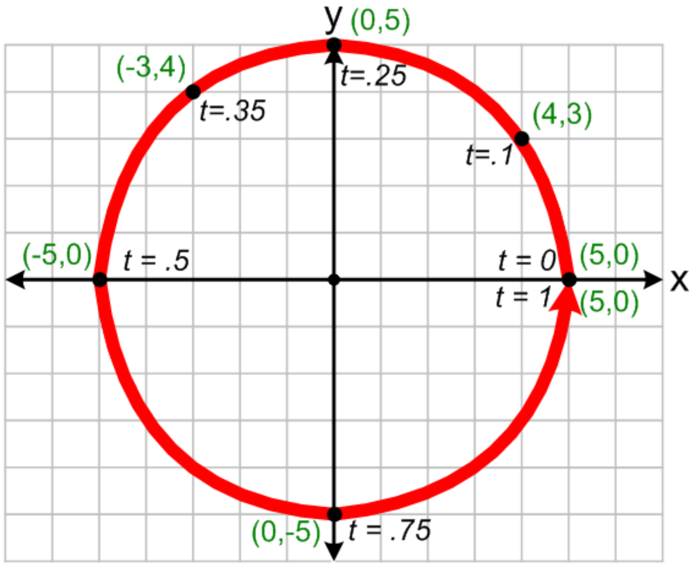


$P_x = 5 * \cos(2\pi * t)$
$P_y = 5 * \sin(2\pi * t)$

(parametric)

Example: P(t) is a 2D position...
Pick some value of t, plug it in, see where P is!

Parametric Equations



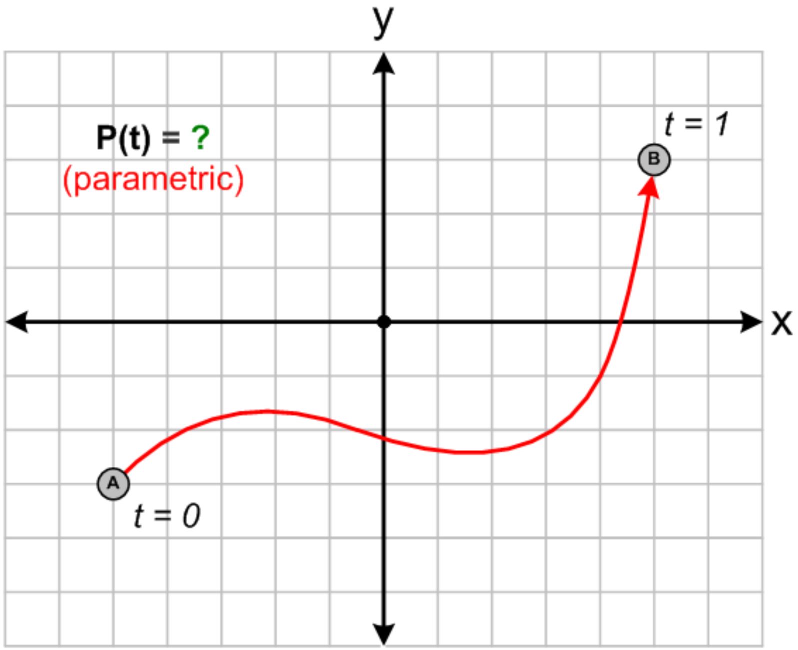
$P_x = 5 * \cos(2\pi * t)$
$P_y = 5 * \sin(2\pi * t)$

(parametric)

Example: P(t) is a 2D position…
Pick some value of t, plug it in, see where P is!

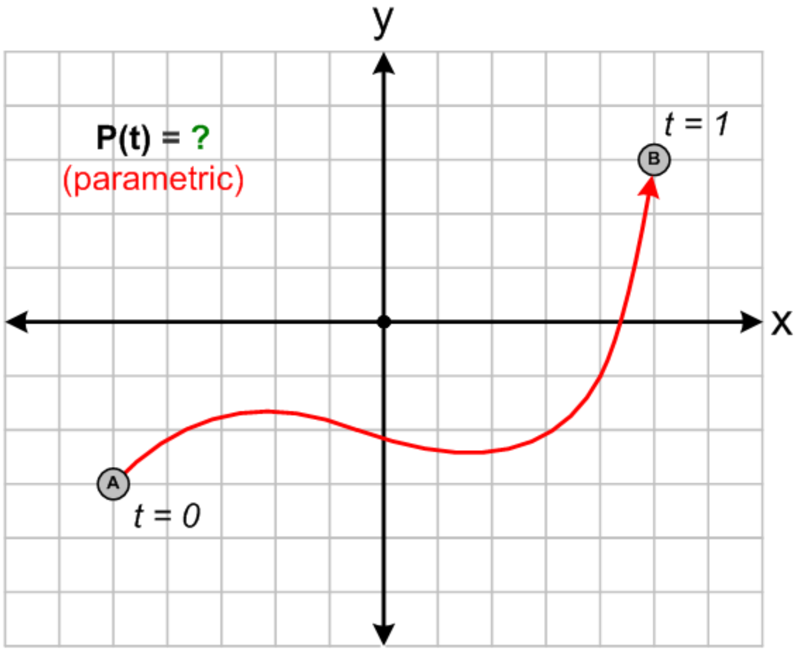
Parametric Curves

Parametric Curves



Parametric curves are curves that are defined using parametric equations.

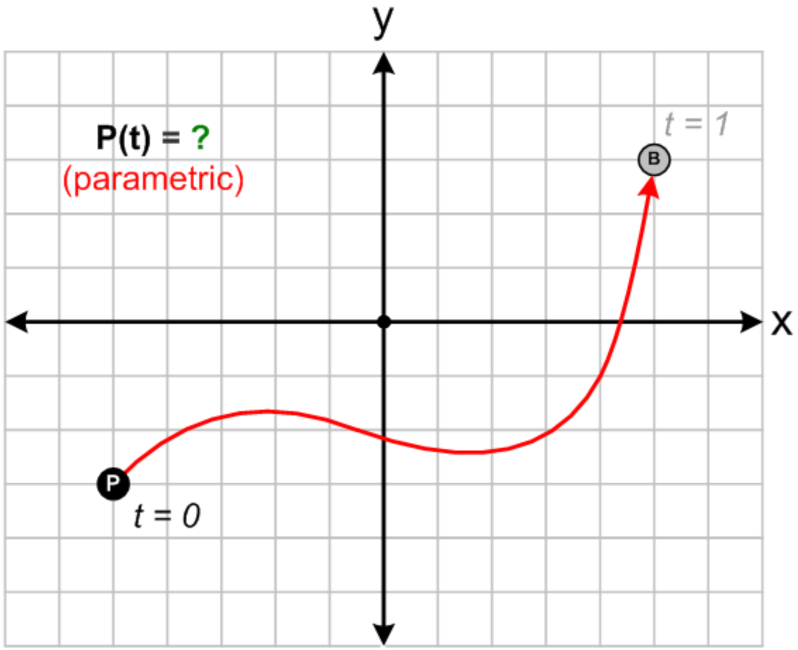
Parametric Curves



Here’s the basic idea:

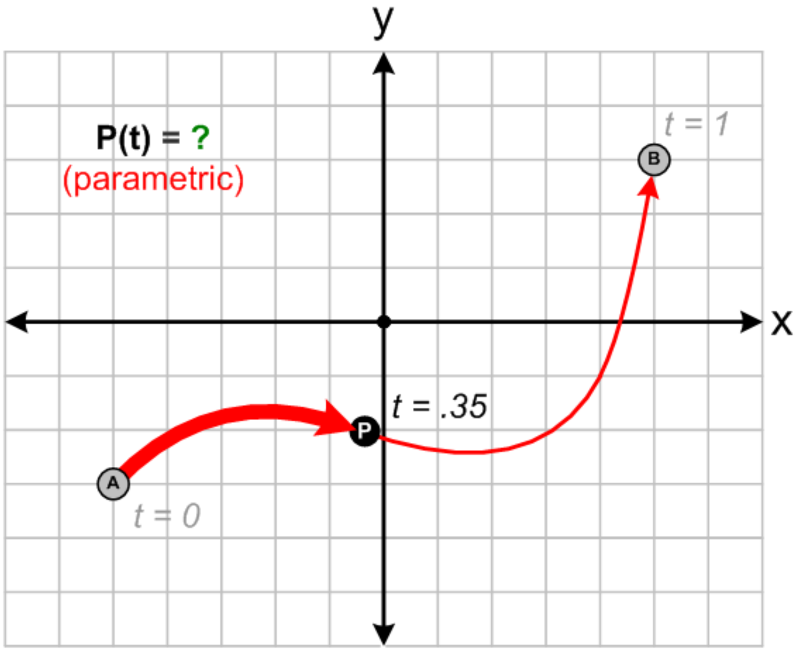
We go from *t=0* at **A** (start) to *t=1* at **B** (end)

Parametric Curves



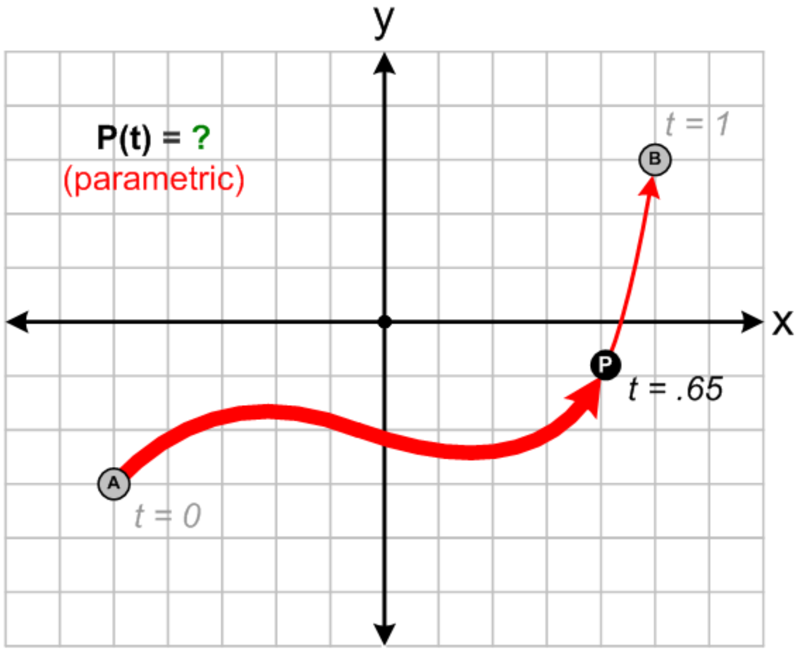
Set the knob to 0, and crank it towards 1

Parametric Curves



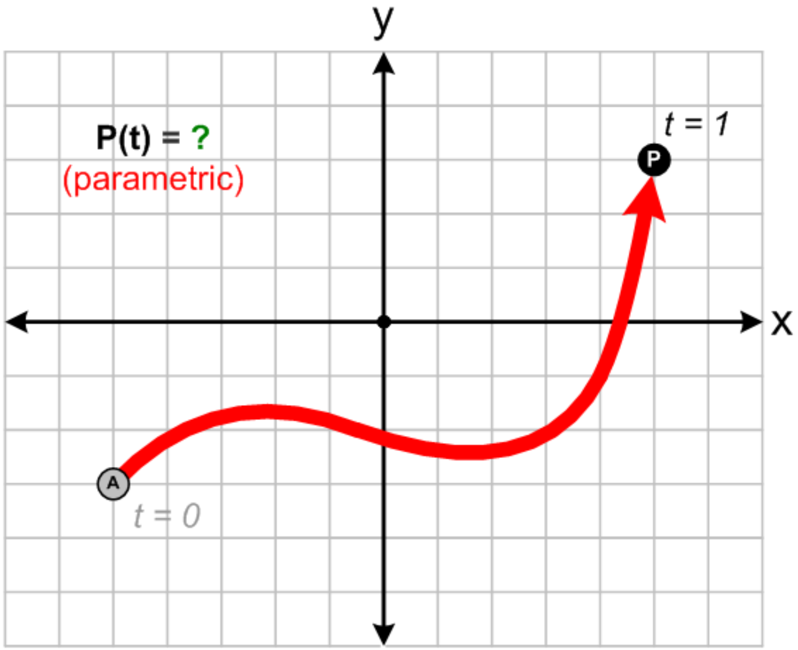
As we turn the knob, we keep plugging the latest t into the curve equation to find out where P is now

Parametric Curves



Note: All parametric curves are **directional**; i.e. they have a start & end, a forward & backward

Parametric Curves



So that’s the basic idea.

Now how do we actually do it?

Bezier Curves

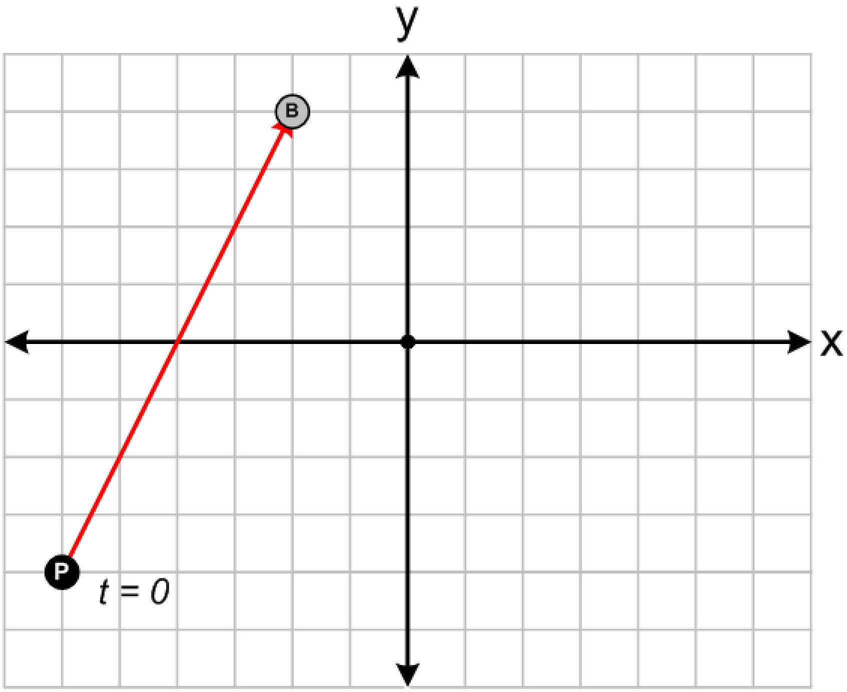
Linear Bezier Curves

Bezier curves are the easiest kind to understand.

The simplest kind of Bezier curves are
Linear Bezier curves.

They’re so simple, they’re not even curvy!

Linear Bezier Curves

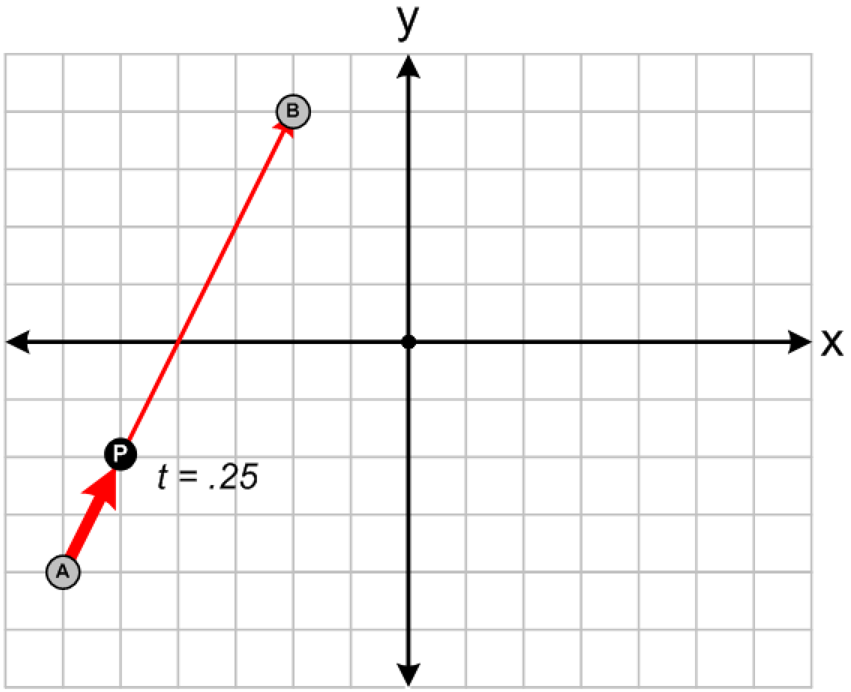


P = ((**1-t**) * **A**) + (t * **B**)   // weighted average

or, as I prefer to write it:

P = (**s** * **A**) + (t * **B**)   ← *where **s** = 1-t*

Linear Bezier Curves

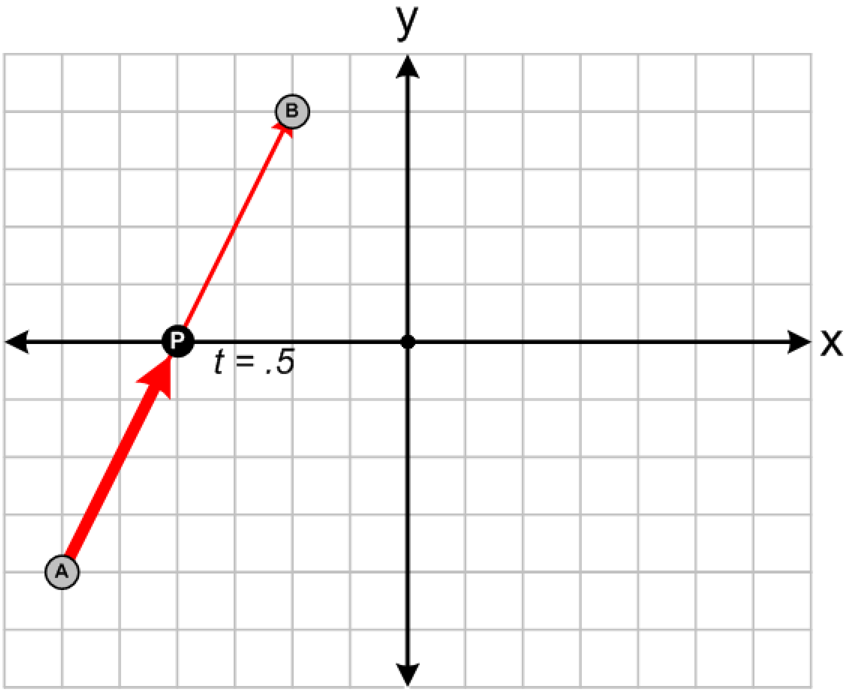


P = (**(1-t)** * **A**) + (t * **B**) // weighted average

or, as I prefer to write it:

P = (**s** * **A**) + (t * **B**) ← *where **s** = 1-t*

Linear Bezier Curves

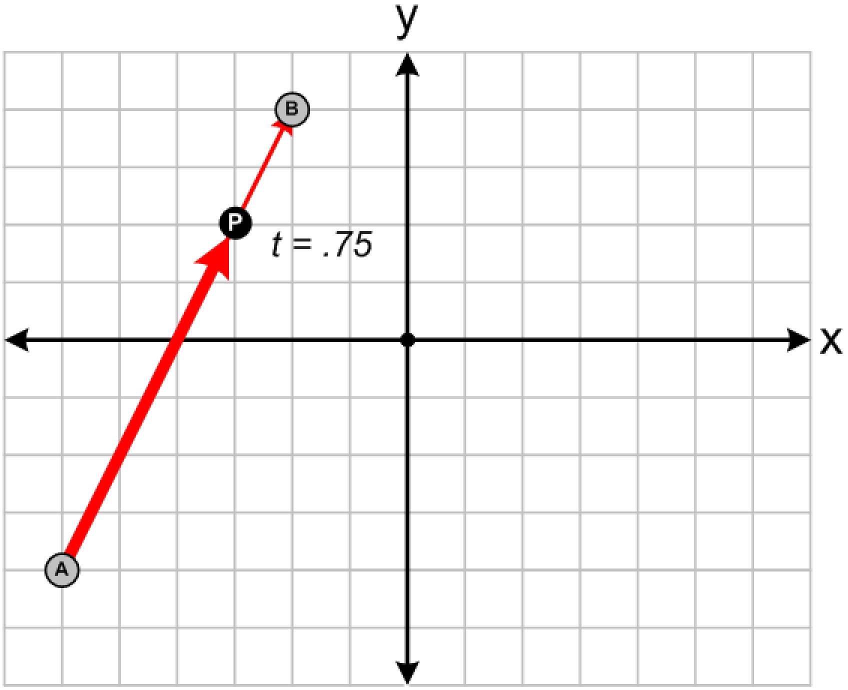


P = ((1-t) * A) + (t * B) // weighted average

or, as I prefer to write it:

P = (s * A) + (t * B) ← *where s = 1-t*

Linear Bezier Curves



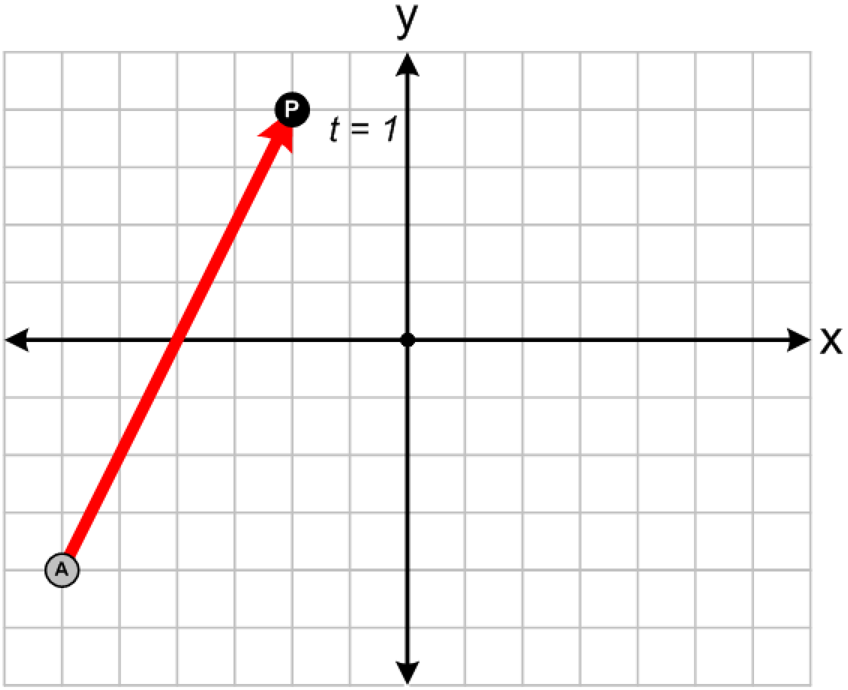
So, for **t = 0.75** (75% of the way from **A** to **B**):

P = (**(1-t)** * **A**) + (**t** * **B**)

or

P = (**.25** * **A**) + (**.75** * **B**)

Linear Bezier Curves



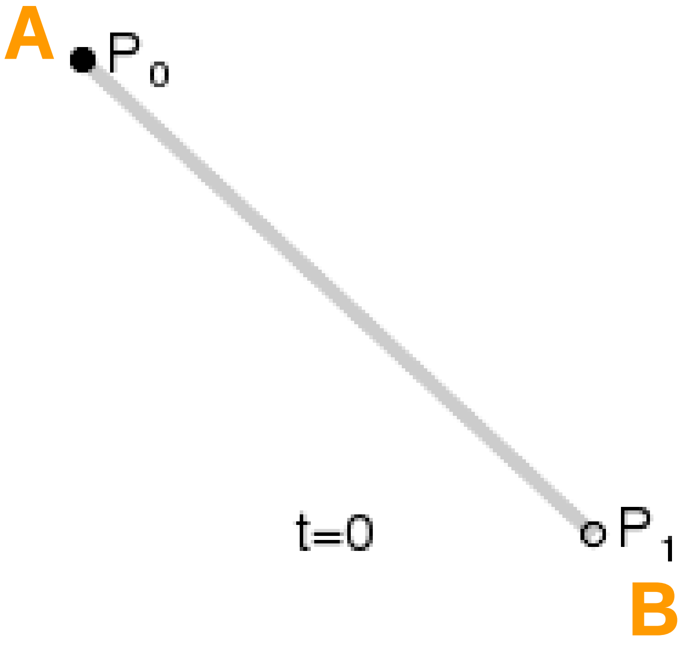
So, for **t = 0.75** (75% of the way from **A** to **B**):

P = (**(1-t)** * **A**) + (**t** * **B**)

or

P = (**.25** * **A**) + (**.75** * **B**)

Linear Bezier Curves



Here it is in motion (thanks, internet!)

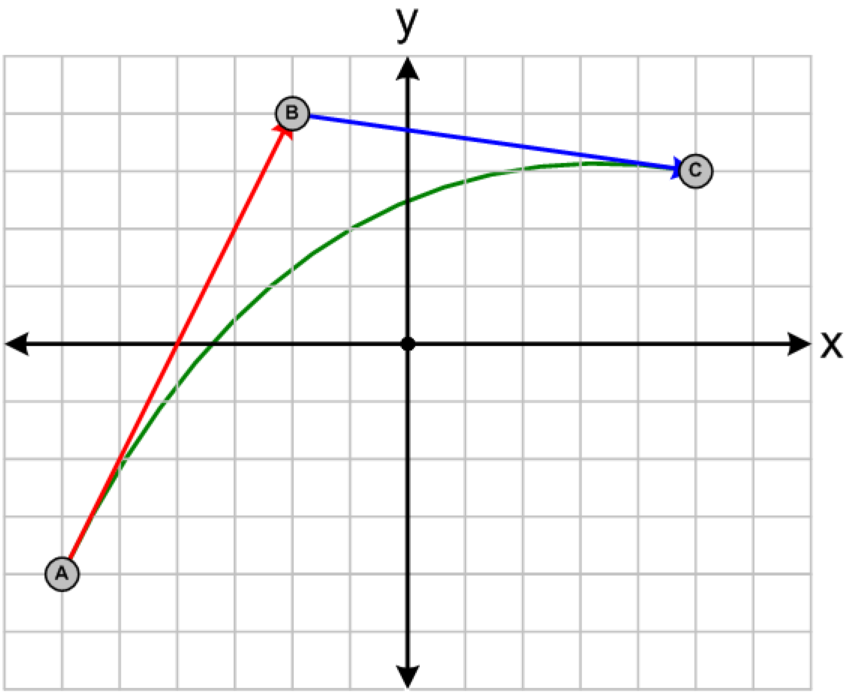
Quadratic Bezier Curves

Quadratic Bezier Curves

A Quadratic Bezier curve is just a **blend of two Linear** Bezier curves.

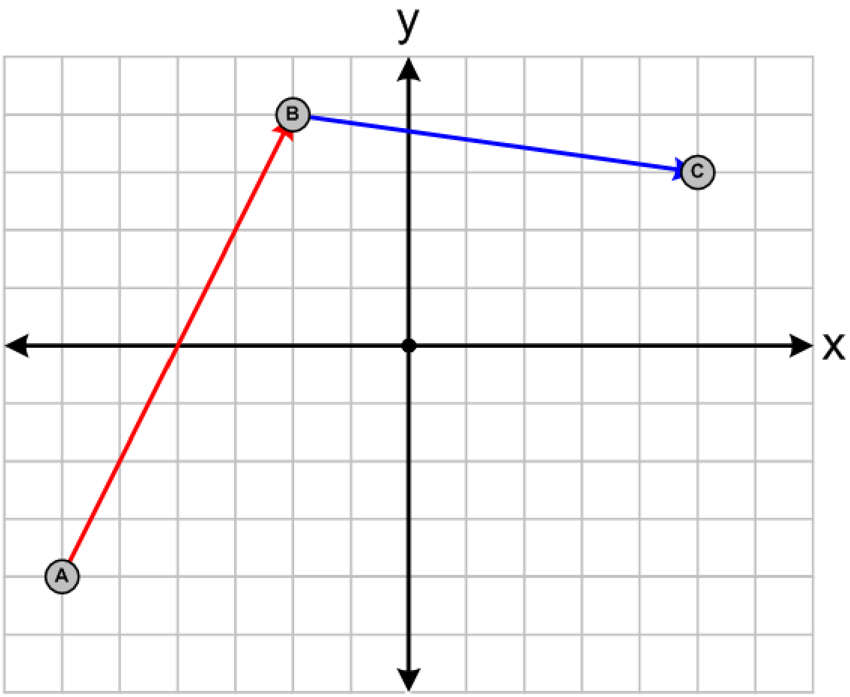
The word “quadratic” means that if we sniff around the math long enough, we’ll see $\mathbf{t^2}$. (In our Linear Beziers we saw **t** and **1-t**, but never $\mathbf{t^2}$).

Quadratic Bezier Curves



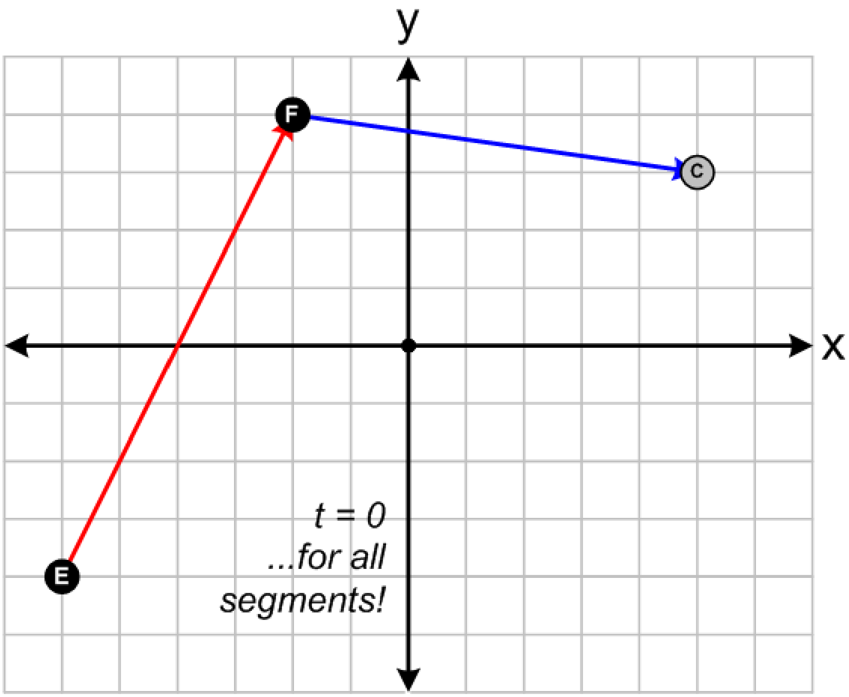
- Three **control points**: **A**, **B**, and **C**

Quadratic Bezier Curves



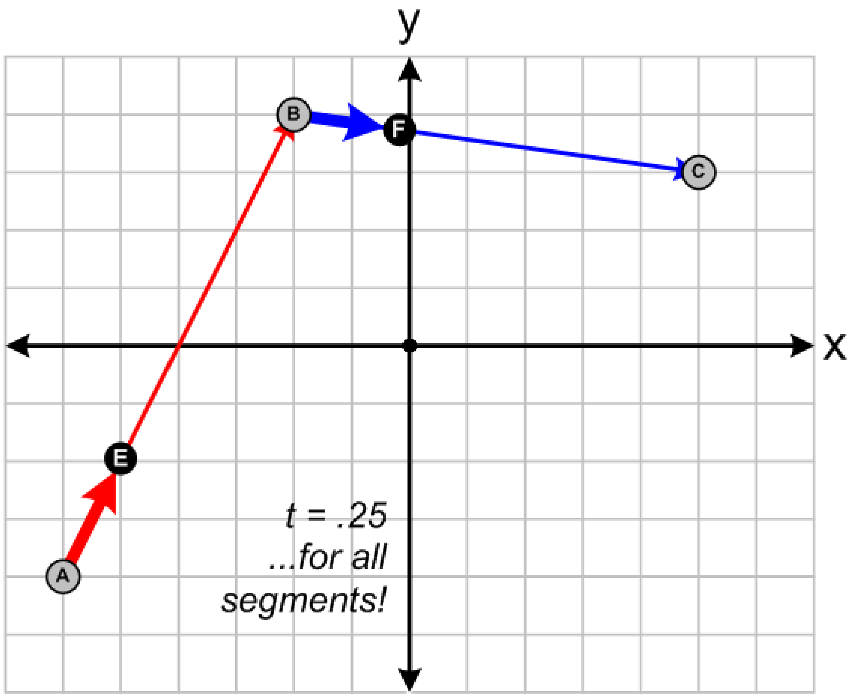
- Three **control points**: **A**, **B**, and **C**
- Two different Linear Beziers: AB and BC

Quadratic Bezier Curves



- Three **control points**: **A**, **B**, and **C**
- Two different Linear Beziers: AB and BC
- Instead of “P”, using “E” for AB and “F” for BC

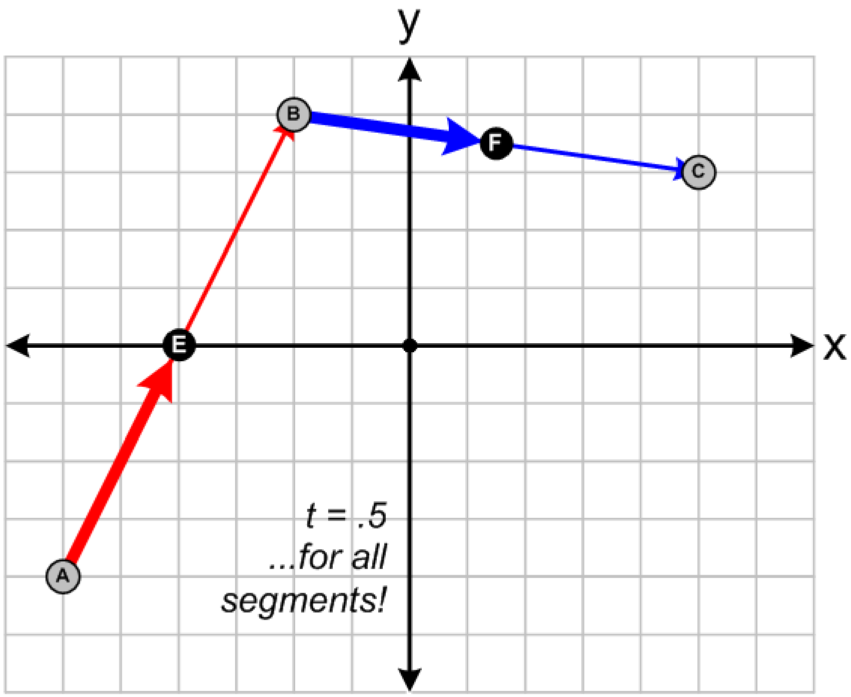
Quadratic Bezier Curves



t = .25
...for all
segments!

- Interpolate E along AB as we turn the knob
- Interpolate F along BC as we turn the knob
- Move E and F simultaneously – only one "t"!

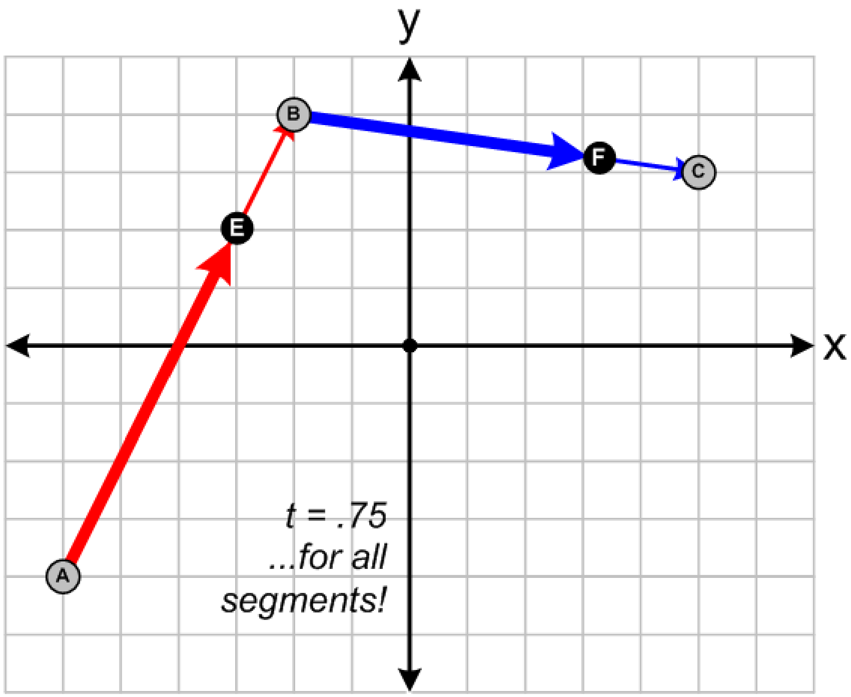
Quadratic Bezier Curves



t = .5
...for all
segments!

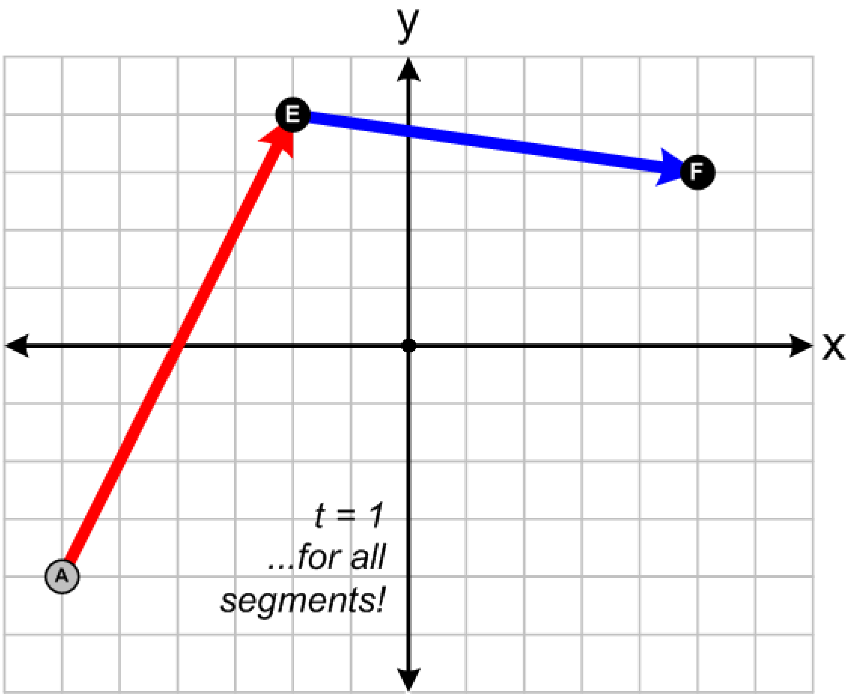
- » Interpolate E along AB as we turn the knob
- » Interpolate F along BC as we turn the knob
- » Move E and F simultaneously – only one “t”!

Quadratic Bezier Curves



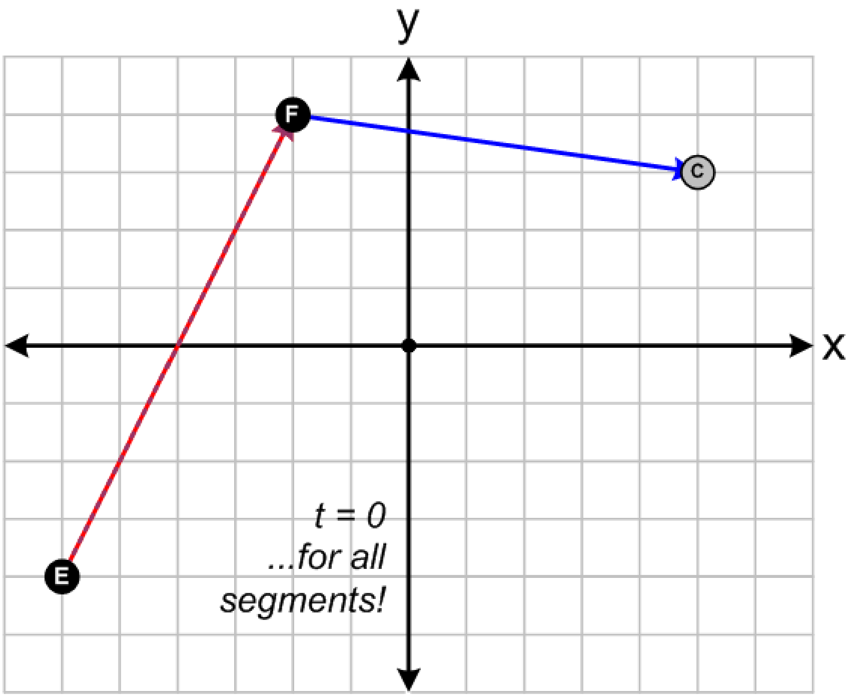
- Interpolate E along AB as we turn the knob
- Interpolate F along BC as we turn the knob
- Move E and F simultaneously – only one “t”!

Quadratic Bezier Curves



- Interpolate E along AB as we turn the knob
- Interpolate F along BC as we turn the knob
- Move E and F simultaneously – only one “t”!

Quadratic Bezier Curves

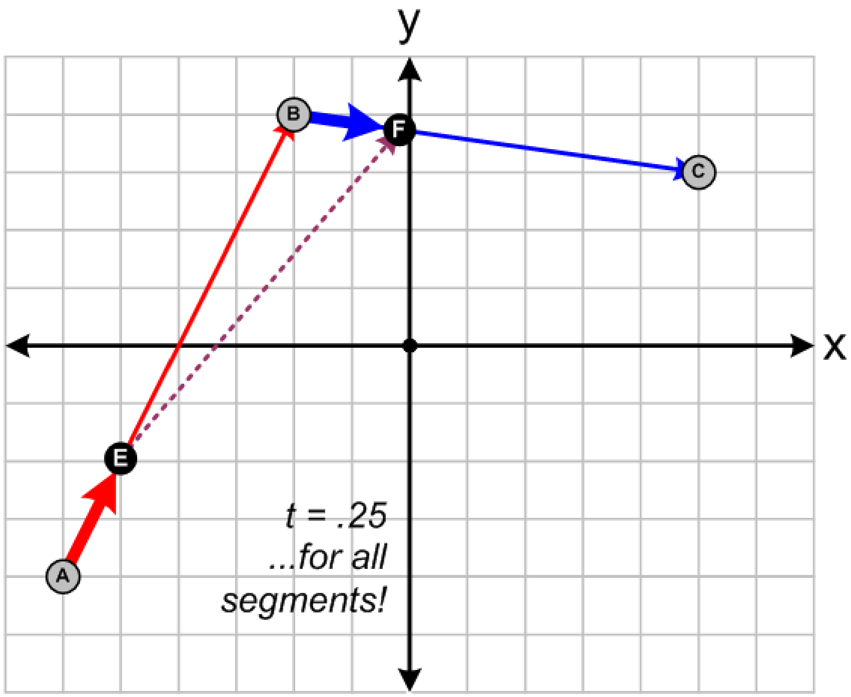


- Now let's turn the knob again...

  (from t=0 to t=1)

but **draw a line** between E and F as they move.

Quadratic Bezier Curves

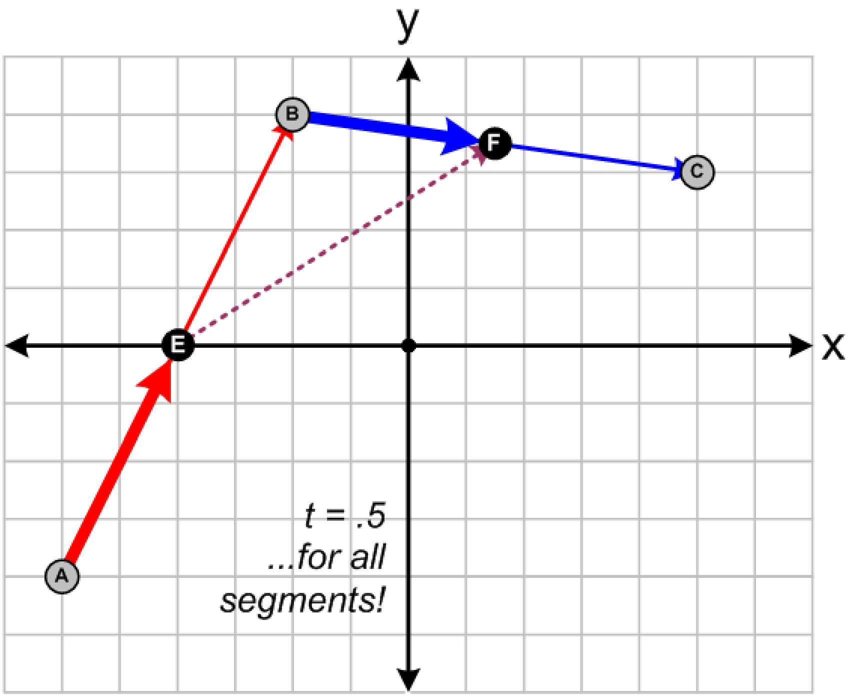


- Now let’s turn the knob again…

  (from t=0 to t=1)

but **draw a line** between E and F as they move.

Quadratic Bezier Curves

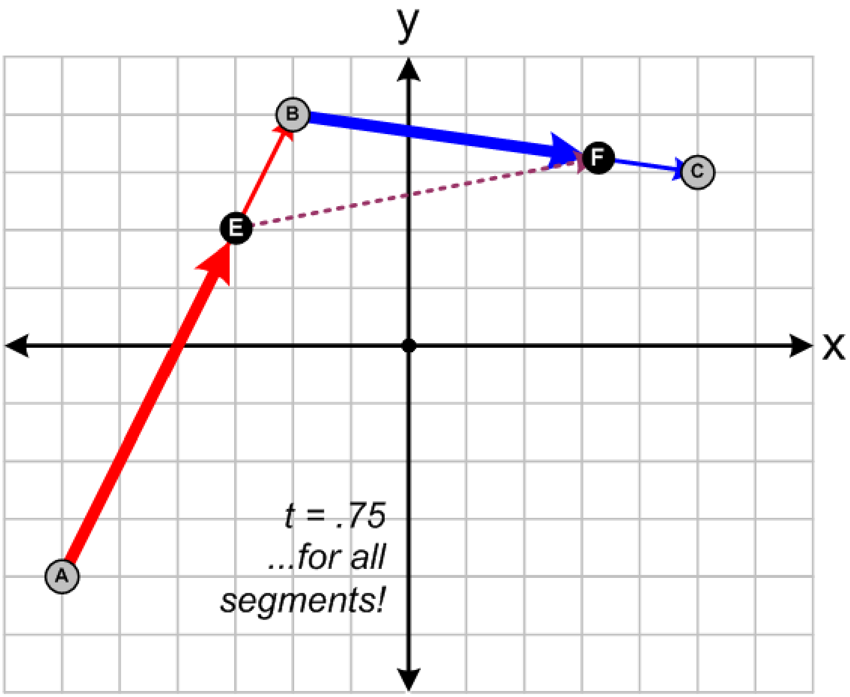


» Now let's turn the knob again…

(from t=0 to t=1)

but **draw a line** between E and F as they move.

Quadratic Bezier Curves

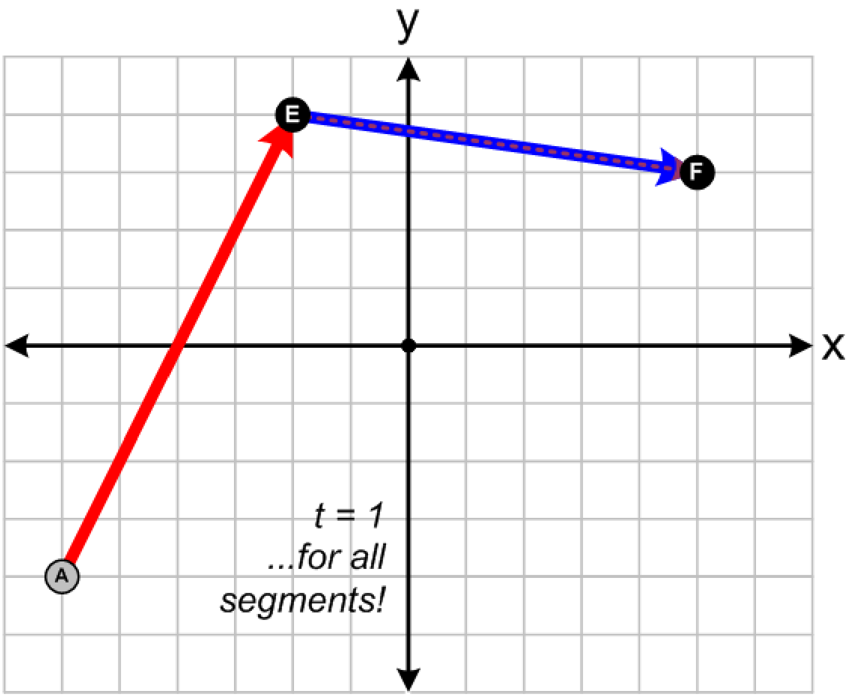


» Now let’s turn the knob again…

(from t=0 to t=1)

but **draw a line** between E and F as they move.

Quadratic Bezier Curves

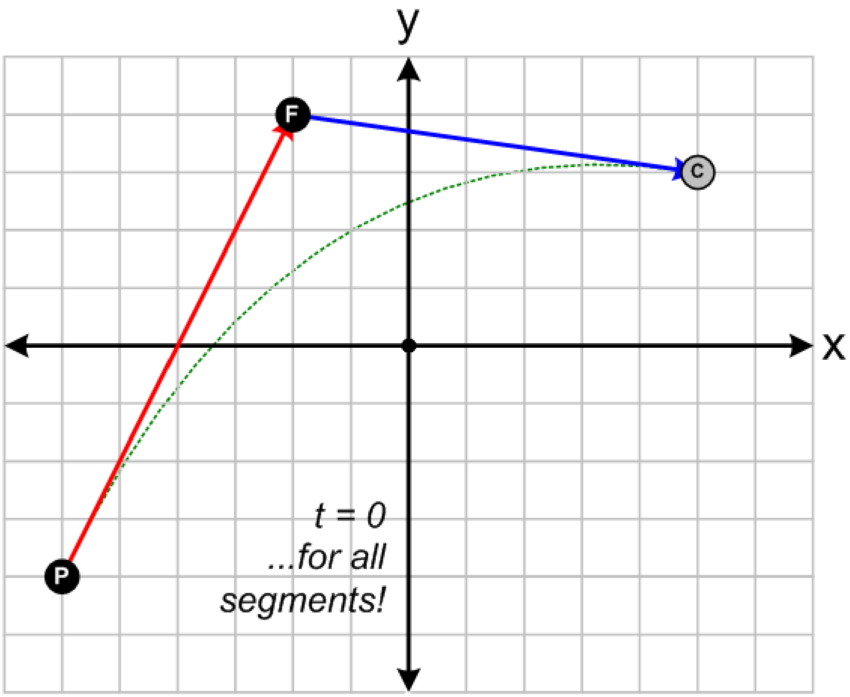


» Now let's turn the knob again...

(from t=0 to t=1)

but **draw a line** between E and F as they move.

Quadratic Bezier Curves

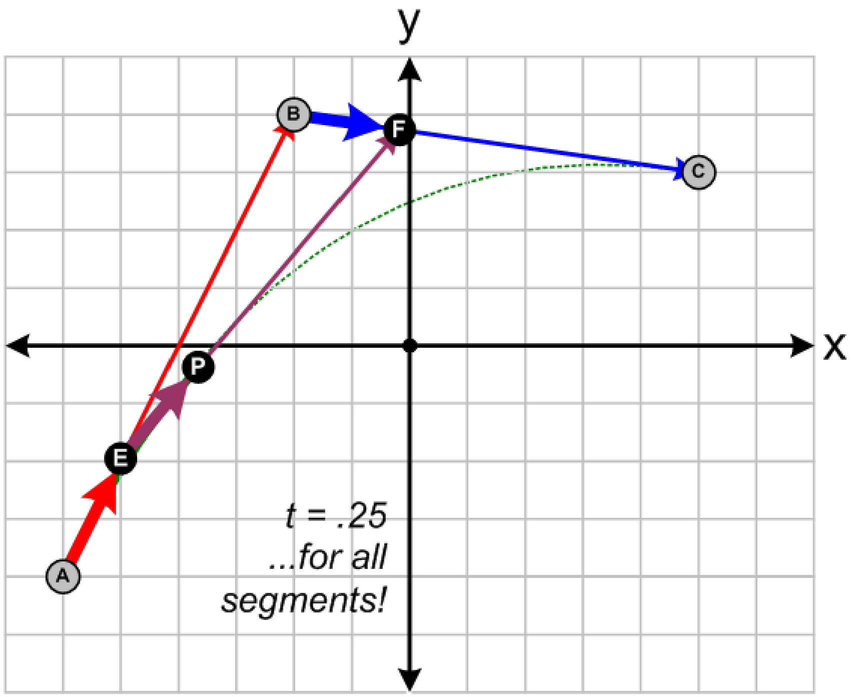


t = 0
...for all segments!

- » This time, we’ll also **interpolate P** from E to F

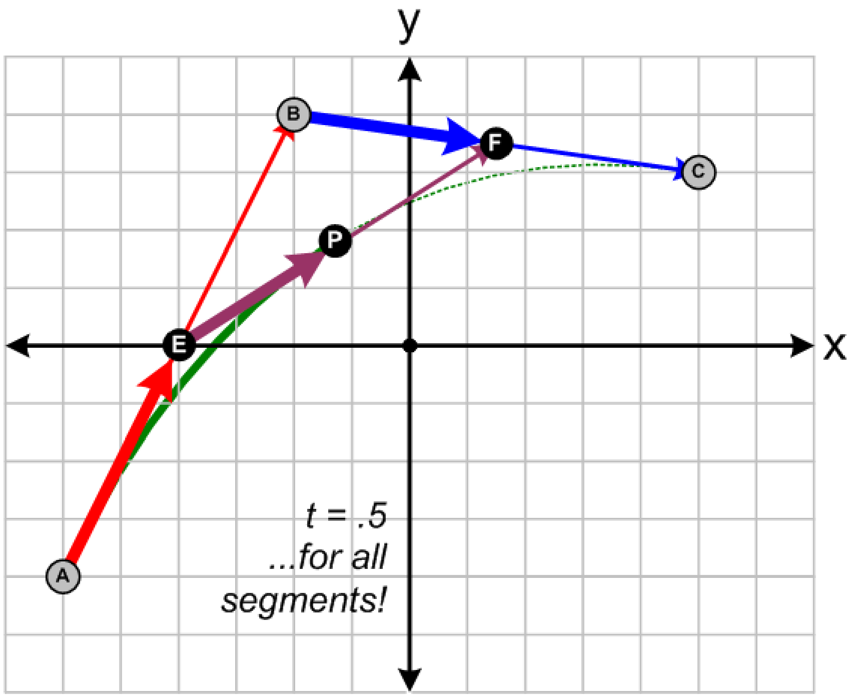
  …using the same “t” as E and F themselves
- » Watch **where P goes**!

Quadratic Bezier Curves



- This time, we'll also **interpolate P** from E to F
  …using the same "t" as E and F themselves
- Watch **where P goes**!

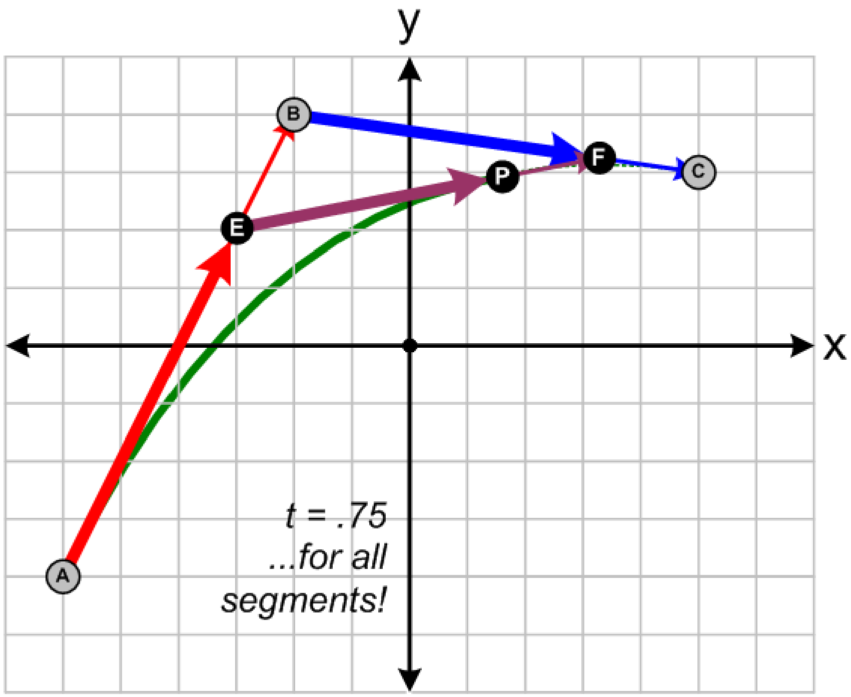
Quadratic Bezier Curves



- » This time, we'll also **interpolate P** from E to F

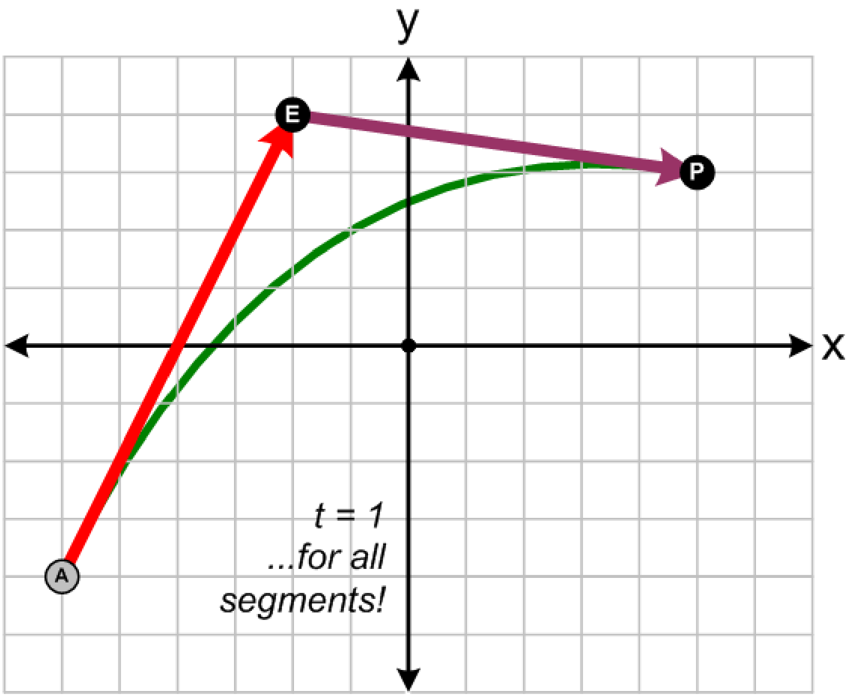
  …using the same "t" as E and F themselves
- » Watch **where P goes**!

Quadratic Bezier Curves



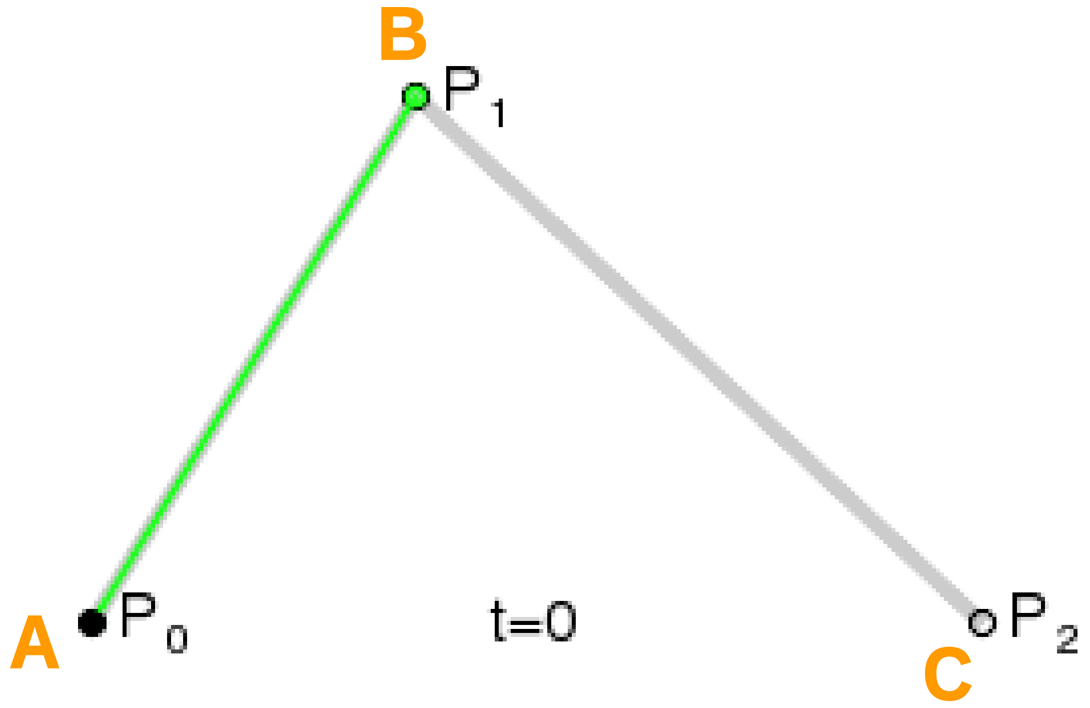
- This time, we'll also **interpolate P** from E to F
  …using the same "t" as E and F themselves
- Watch **where P goes**!

Quadratic Bezier Curves



- This time, we'll also **interpolate P** from E to F
  …using the same "t" as E and F themselves
- Watch **where P goes**!

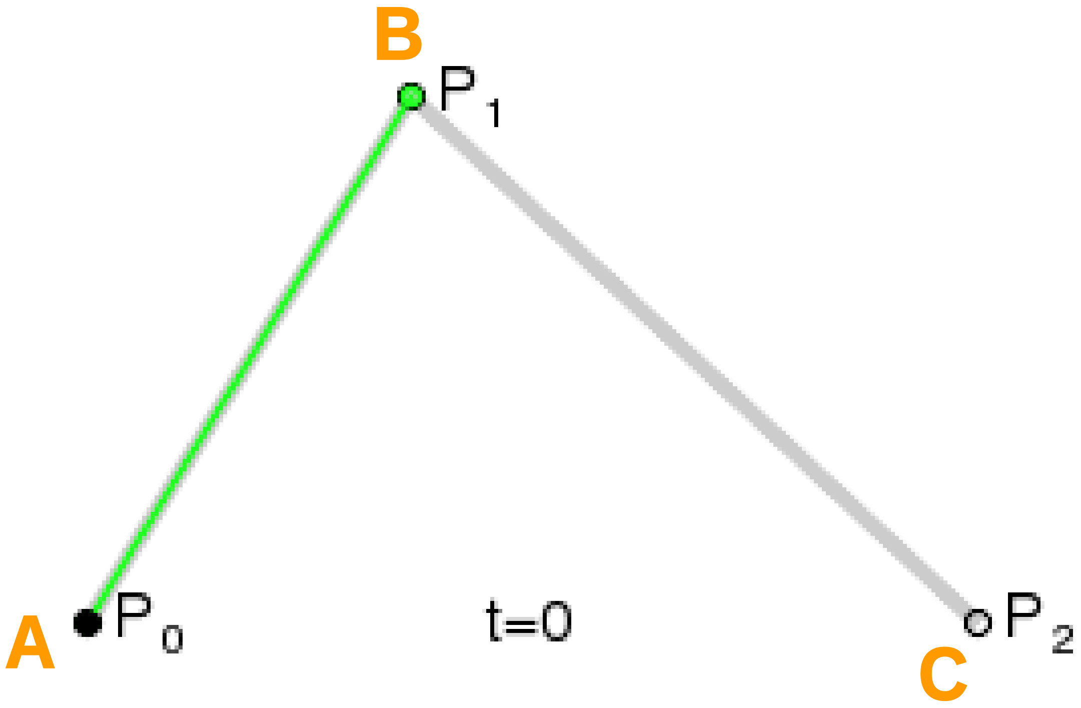
Quadratic Bezier Curves



- Note that mathematicians use

  $\mathbf{P_0}$, $\mathbf{P_1}$, $\mathbf{P_2}$ instead of **A**, **B**, **C**
- I will keep using **A**, **B**, **C** here for simplicity

Quadratic Bezier Curves

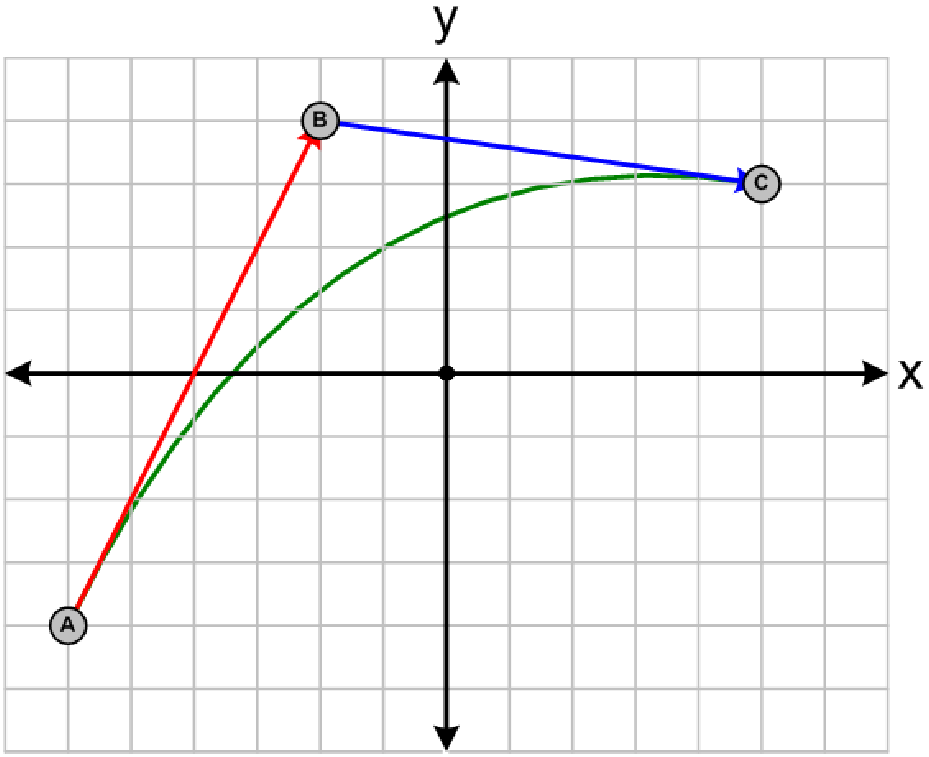


- We know P starts at **A**, and ends at **C**
- It is clearly influenced by **B**…

…but it **never actually touches B**

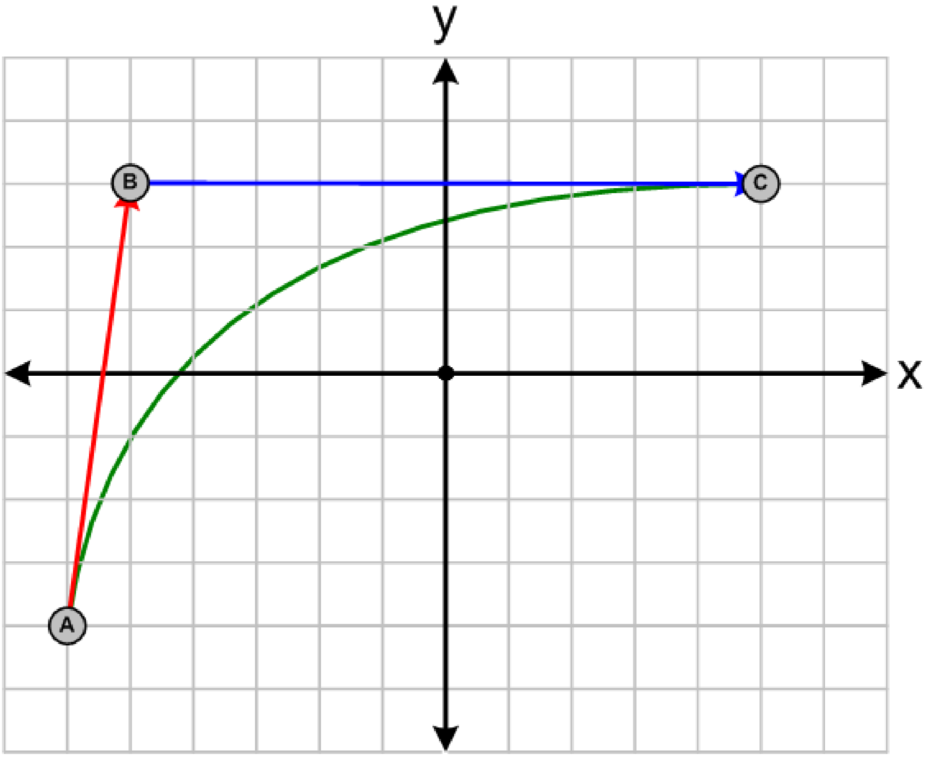
Quadratic Bezier Curves

- **B** is a **guide point** of this curve; drag it around to change the curve’s contour.



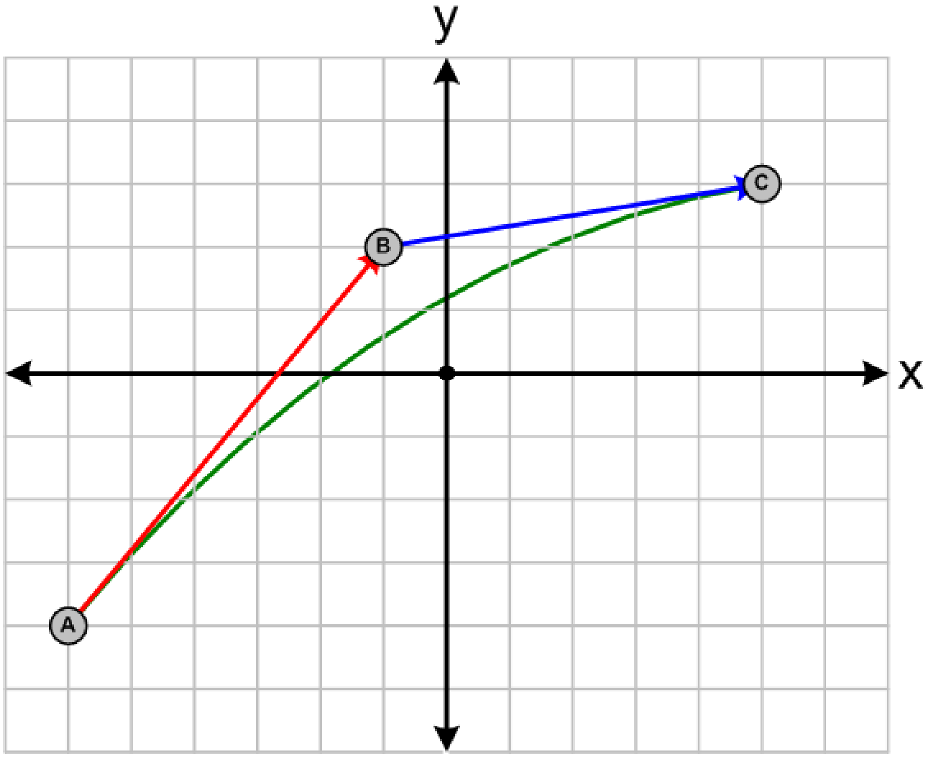
Quadratic Bezier Curves

» **B** is a **guide point** of this curve; drag it around to change the curve’s contour.



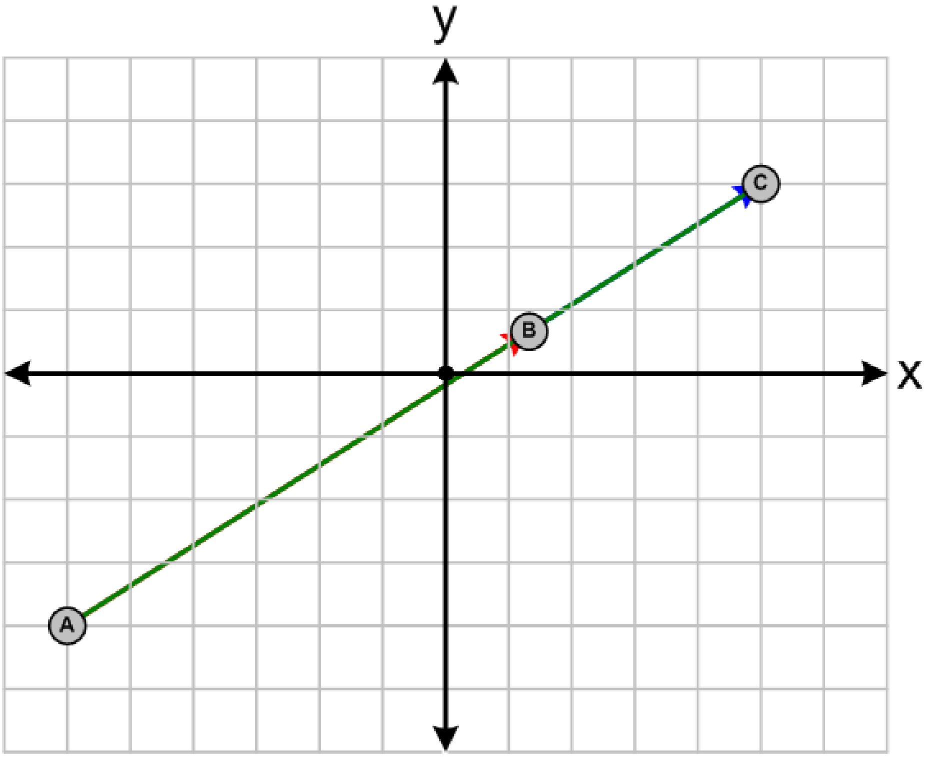
Quadratic Bezier Curves

- **B** is a **guide point** of this curve; drag it around to change the curve’s contour.

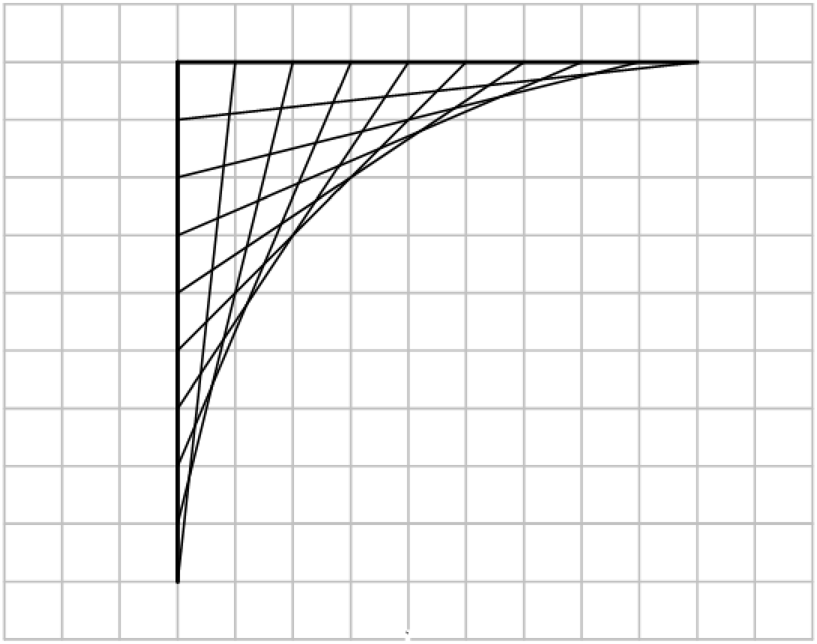


Quadratic Bezier Curves

- **B** is a **guide point** of this curve; drag it around to change the curve's contour.



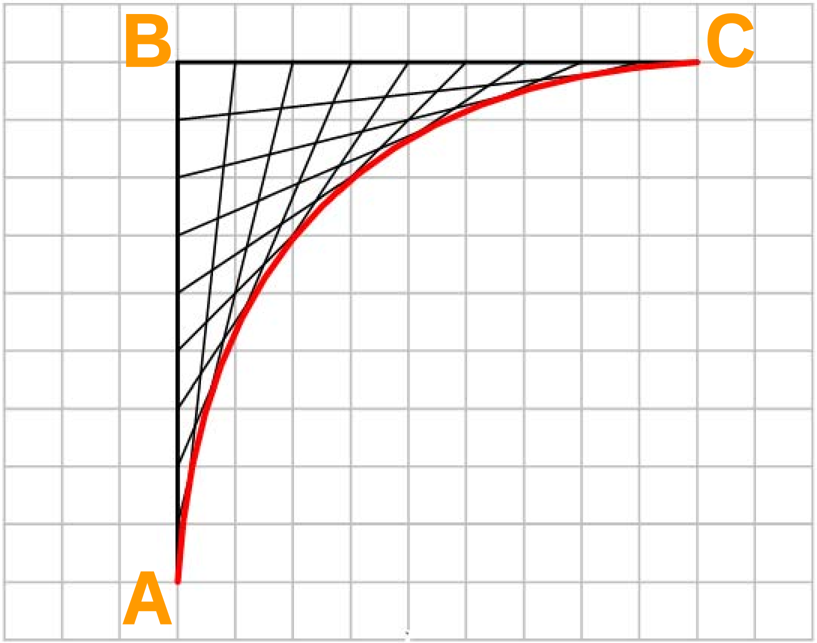
Quadratic Bezier Curves



- By the way, this is also that thing you were drawing in junior high when you were bored.

(when you weren't drawing D&D maps, that is)

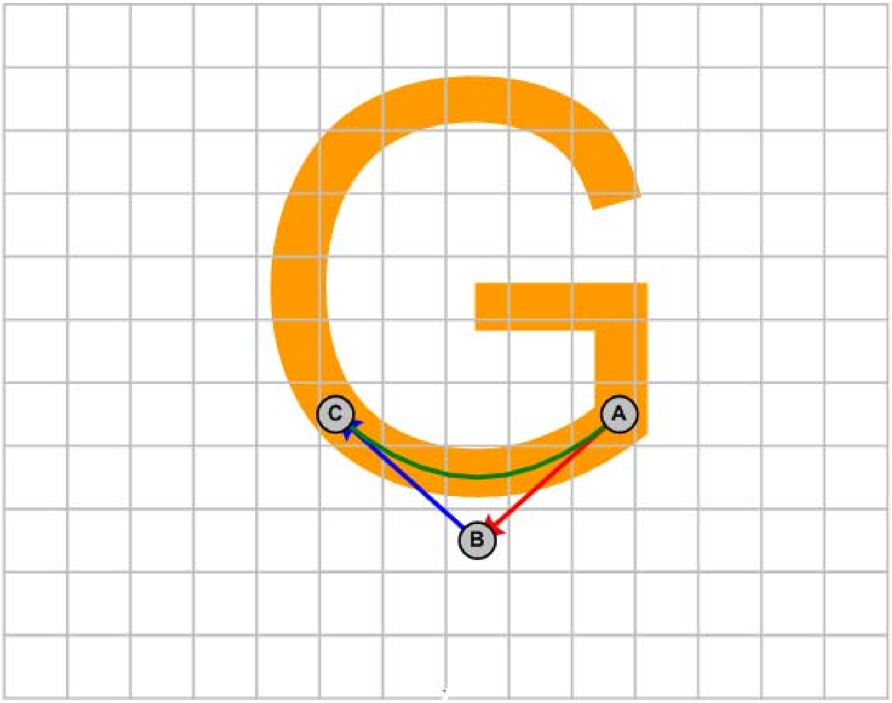
Quadratic Bezier Curves



- By the way, this is also that thing you were drawing in junior high when you were bored.

(when you weren't drawing D&D maps, that is)

Quadratic Bezier Curves



» BONUS: This is also how they make **True Type Fonts** look nice and curvy.

Quadratic Bezier Curves

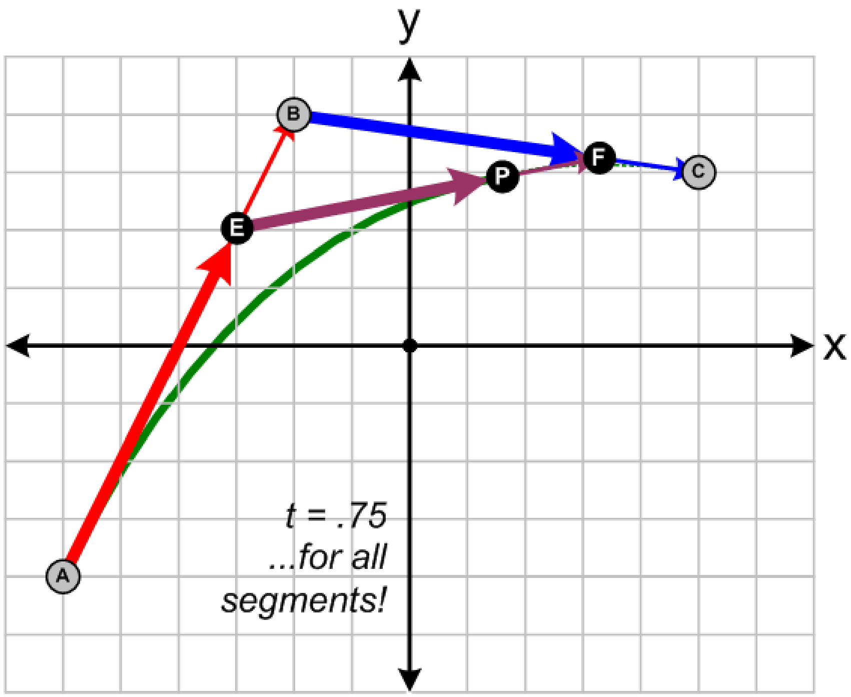
- Remember:

A Quadratic Bezier curve is just a **blend of two Linear** Bezier curves.

So the math is still pretty simple.

(Just a blend of two Linear Bezier equations.)

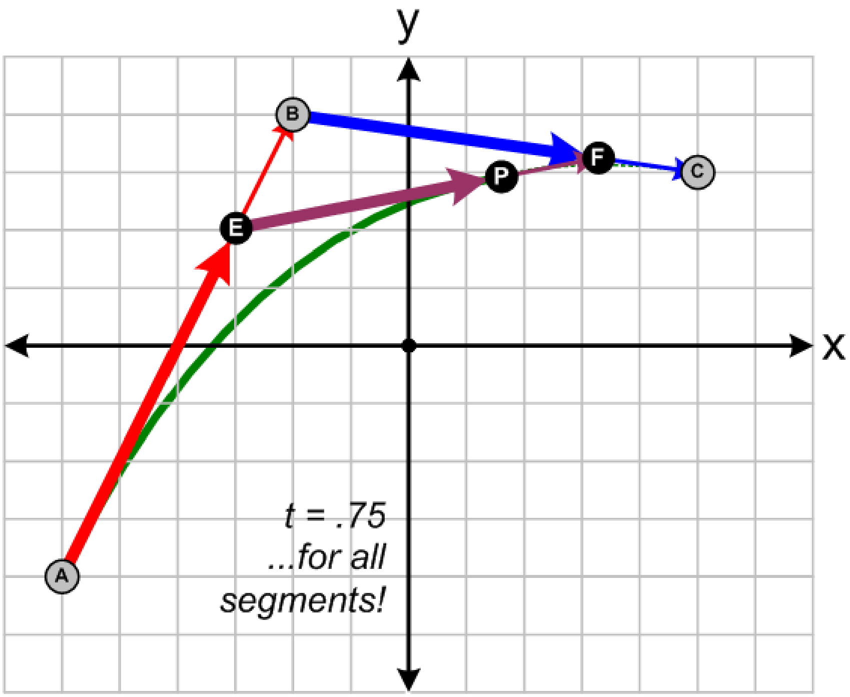
Quadratic Bezier Curves



t = .75
...for all
segments!

- $E(t) = (s * A) + (t * B)$ ← *where* ***s*** *= 1-t*
- $F(t) = (s * B) + (t * C)$
- $P(t) = (s * E) + (t * F)$ ← *technically E(t) and F(t) here*

Quadratic Bezier Curves



t = .75
...for all segments!

- » E(t) = sA + tB ← *where **s** = 1-t*
- » F(t) = sB + tC
- » P(t) = sE + tF ← *technically E(t) and F(t) here*

Quadratic Bezier Curves

- Hold on! You said “quadratic” meant we’d see a $\mathbf{t}^2$ in there somewhere.

- $E(t) = sA + tB$
- $F(t) = sB + tC$
- $P(t) = sE(t) + tF(t)$

- $P(t)$ is an interpolation from $E(t)$ to $F(t)$
- When you plug the $E(t)$ and $F(t)$ equations into the $P(t)$ equation, you get…

Quadratic Bezier Curves

» One equation to rule them all:

$P(t) = sE(t) + tF(t)$

or

$P(t) = s(sA + tB) + t(sB + tC)$

or

$P(t) = (s^2)A + (st)B + (st)B + (t^2)C$

or

$P(t) = (s^2)A + 2(st)B + (t^2)C$

(BTW, there’s our “quadratic” $\mathbf{t^2}$)

Quadratic Bezier Curves

- What if $t = 0$? (at the start of the curve)
  so then... s = 1

$P(t) = (s^2)A + 2(st)B + (t^2)C$

becomes

$P(t) = (1^2)A + 2(1*0)B + (0^2)C$

becomes

$P(t) = (1)A + 2(0)B + (0)C$

becomes

$P(t) = A$

Quadratic Bezier Curves

- What if $t = 1$? (at the end of the curve)

  so then... s = 0

$$P(t) = (s^2)A + 2(st)B + (t^2)C$$

becomes

$$P(t) = (0^2)A + 2(0*1)B + (1^2)C$$

becomes

$$P(t) = (0)A + 2(0)B + (1)C$$

becomes

$$P(t) = C$$

Quadratic Bezier Curves

» What if *t = 0.5* ? (halfway through the curve)

so then… s = 0.5 also

$$P(t) = (s^2)A + 2(st)B + (t^2)C$$

becomes

$$P(t) = (0.5^2)A + 2(0.5*0.5)B + (0.5^2)C$$

becomes

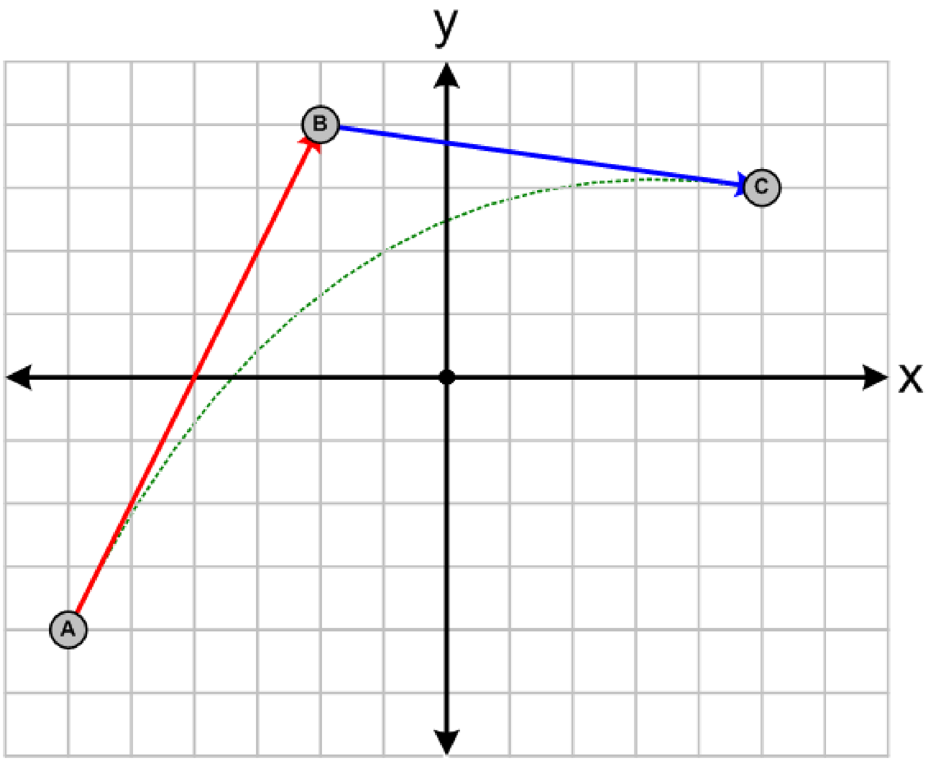
$$P(t) = (0.25)A + 2(0.25)B + (.25)C$$

becomes

$$P(t) = .25A + .50B + .25C$$

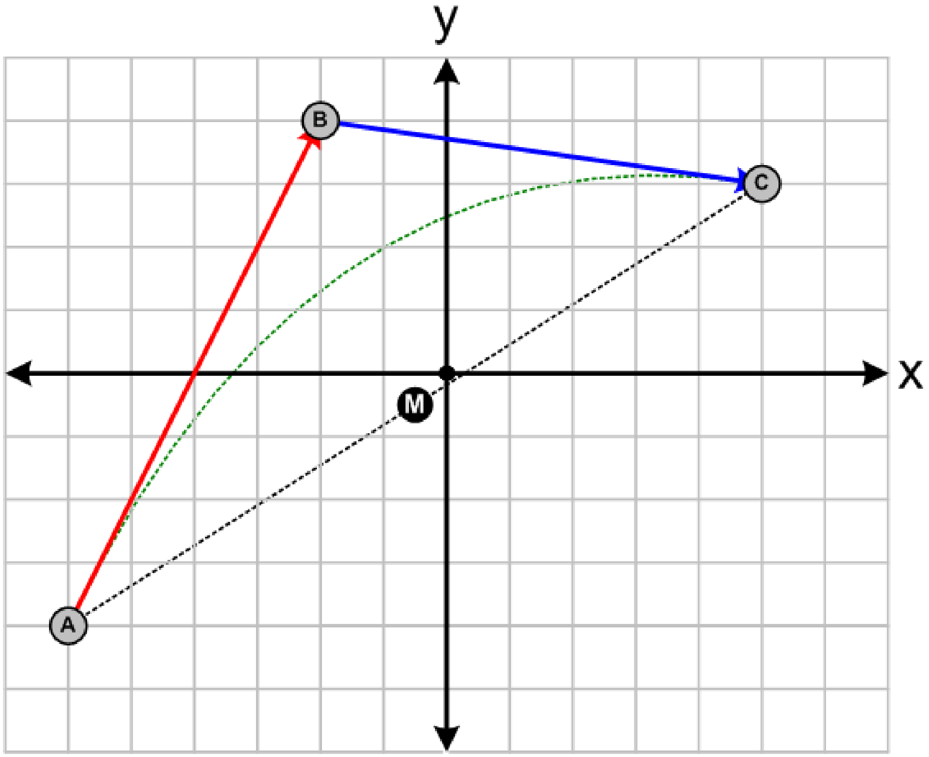
Quadratic Bezier Curves

» If we say **M** is the midpoint of the line **AC**...



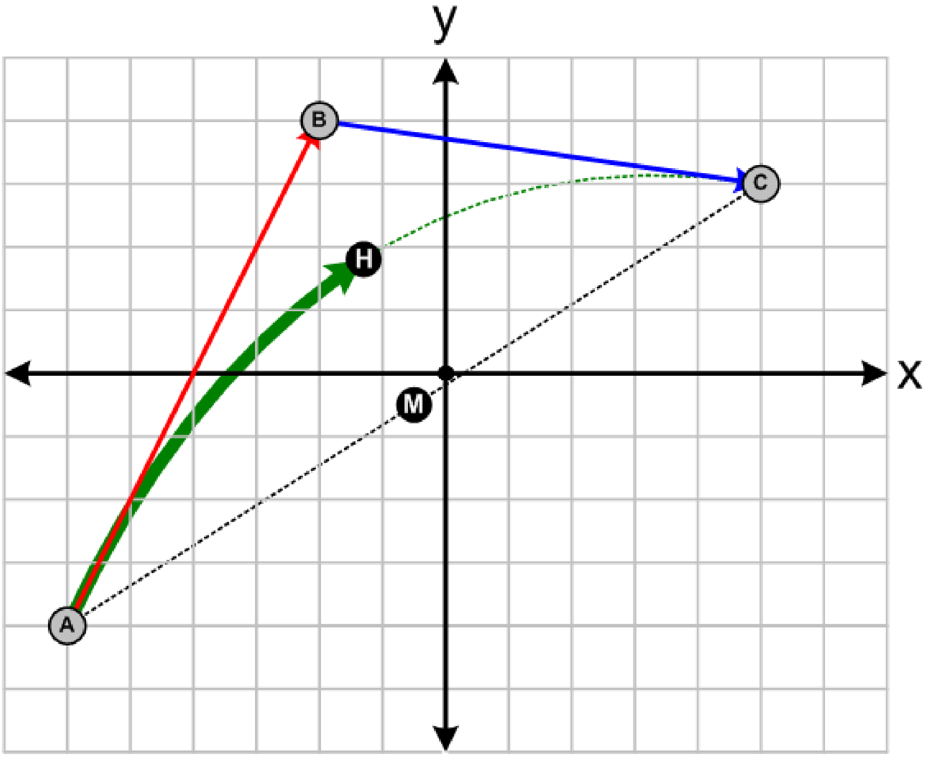
Quadratic Bezier Curves

- If we say **M** is the midpoint of the line **AC**...



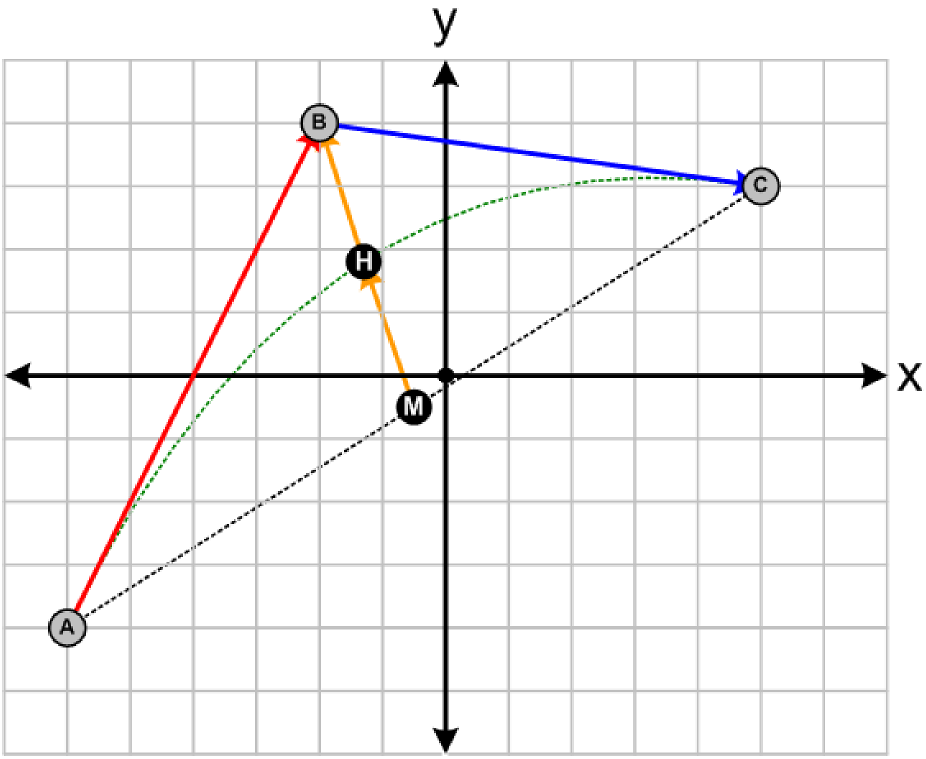
Quadratic Bezier Curves

- » And **H** is the halfway point on the curve **(where *t = 0.5*)**



Quadratic Bezier Curves

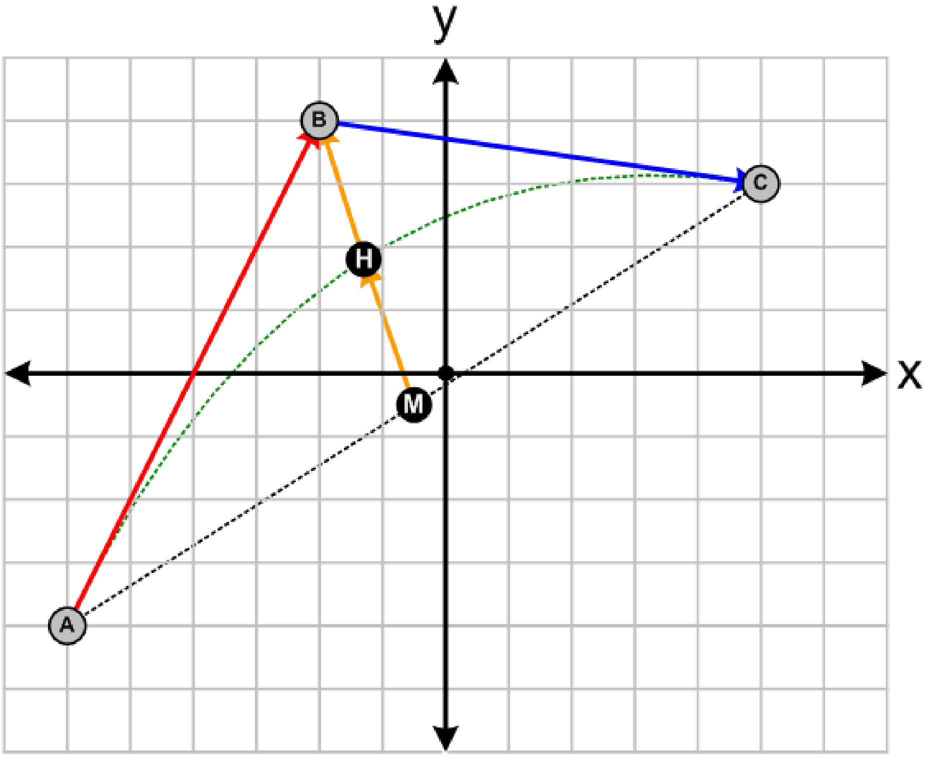
» Then **H** is also halfway from **M** to **B**



Quadratic Bezier Curves

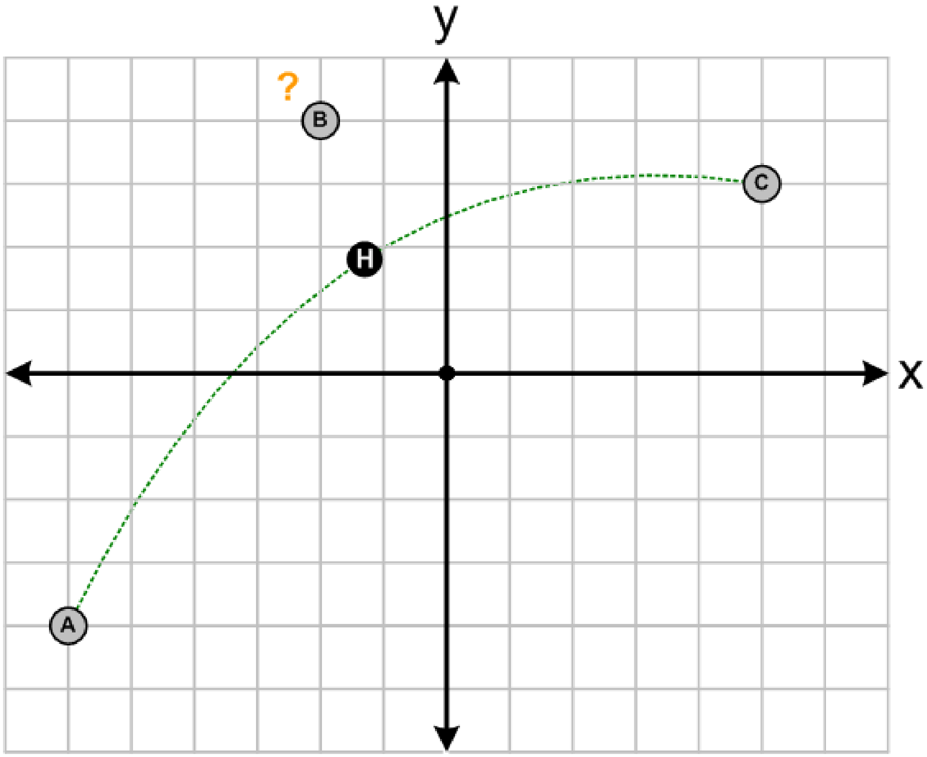
- So, let's say that we'd rather drag the halfway point (**H**) around than **B**.

(maybe because **H** is *on the curve itself)*



Quadratic Bezier Curves

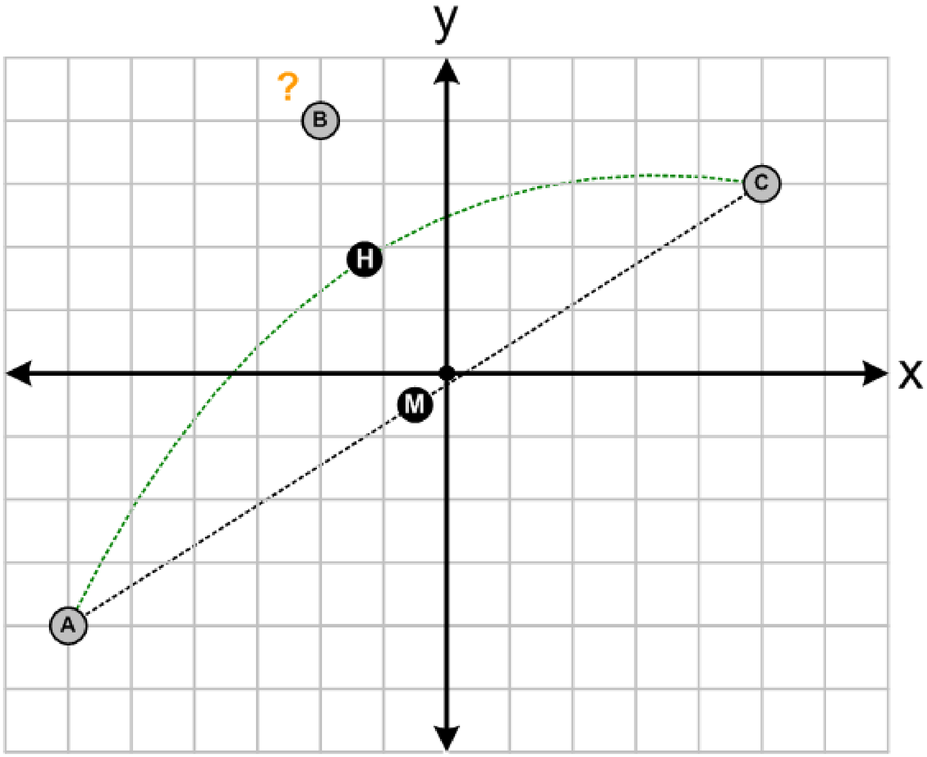
» So now we know **H**, but not **B**.
(and we also know **A** and **C**)



Quadratic Bezier Curves

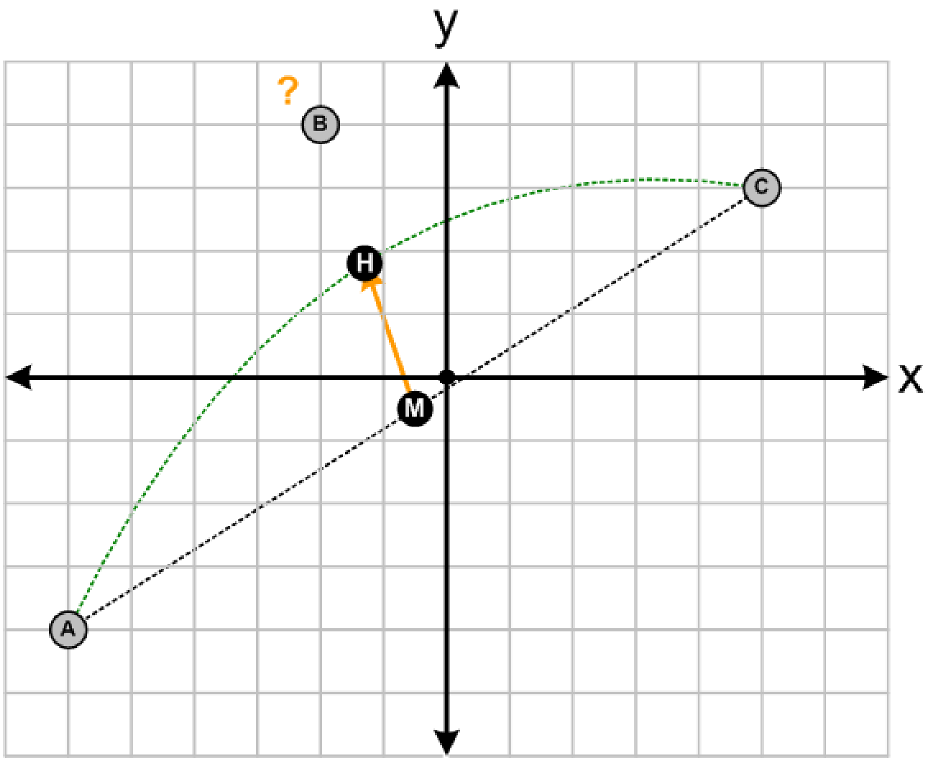
- » Start by computing **M** (midpoint of **AC**):

  M = .5**A** + .5**C**



Quadratic Bezier Curves

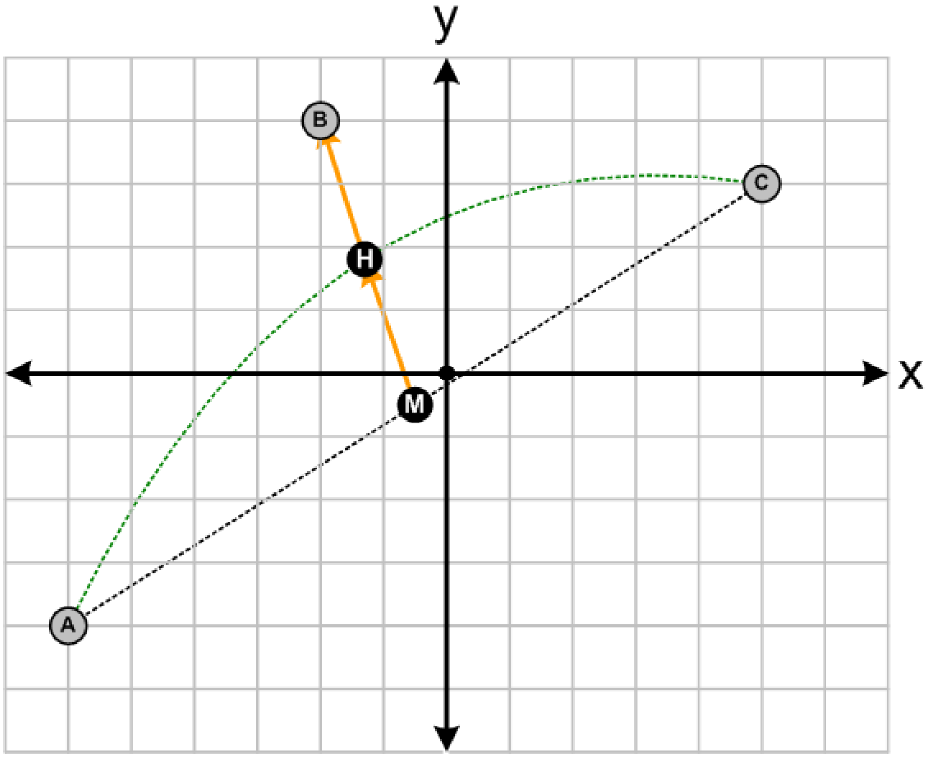
- Compute **MH** (H – M)



Quadratic Bezier Curves

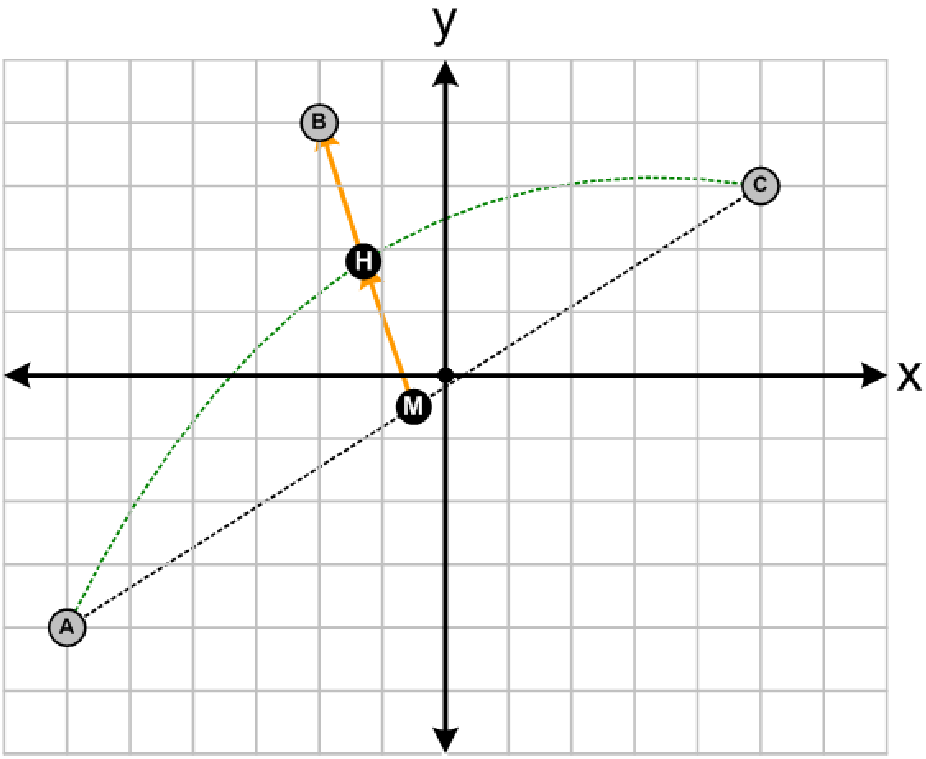
- Add **MH** to **H** to get **B**

  B = **H** + **MH**   (or 2**H** – **M**)



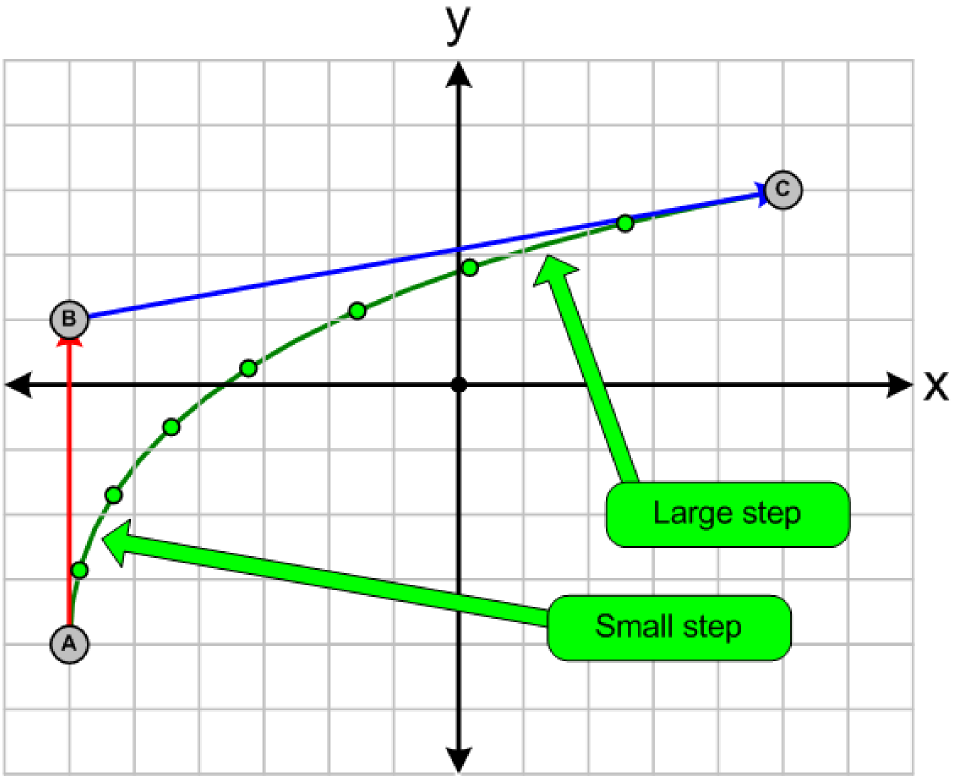
Quadratic Bezier Curves

- This is what programs like Visio do when you drag curve points, BTW.



Non-uniformity

- Be careful: most curves are not **uniform**; that is, they have variable “density” or “speed” throughout them.



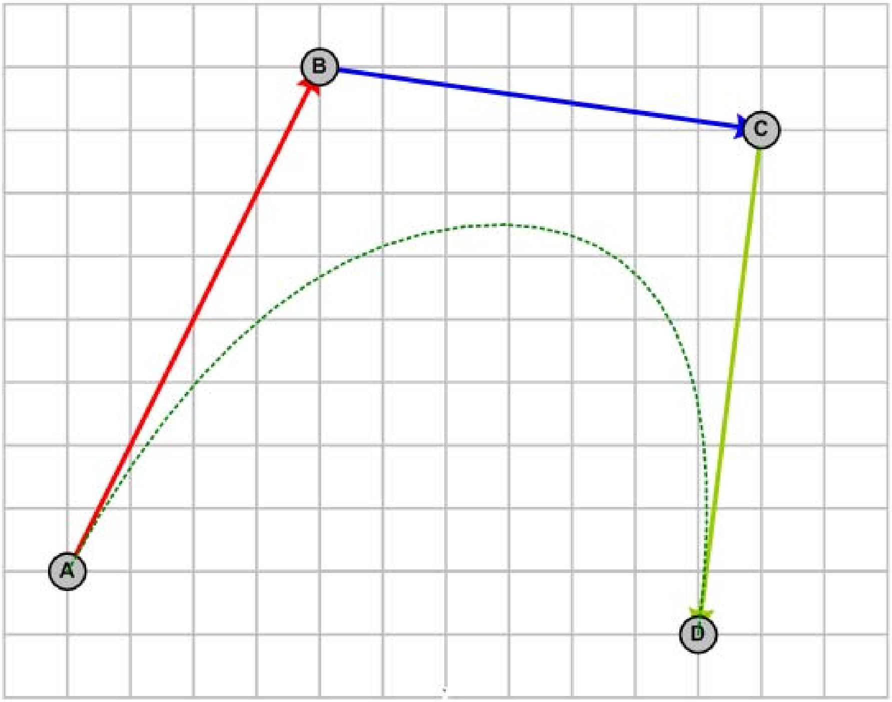
Cubic Bezier Curves

Cubic Bezier Curves

A Cubic Bezier curve is just a **blend of two Quadratic** Bezier curves.

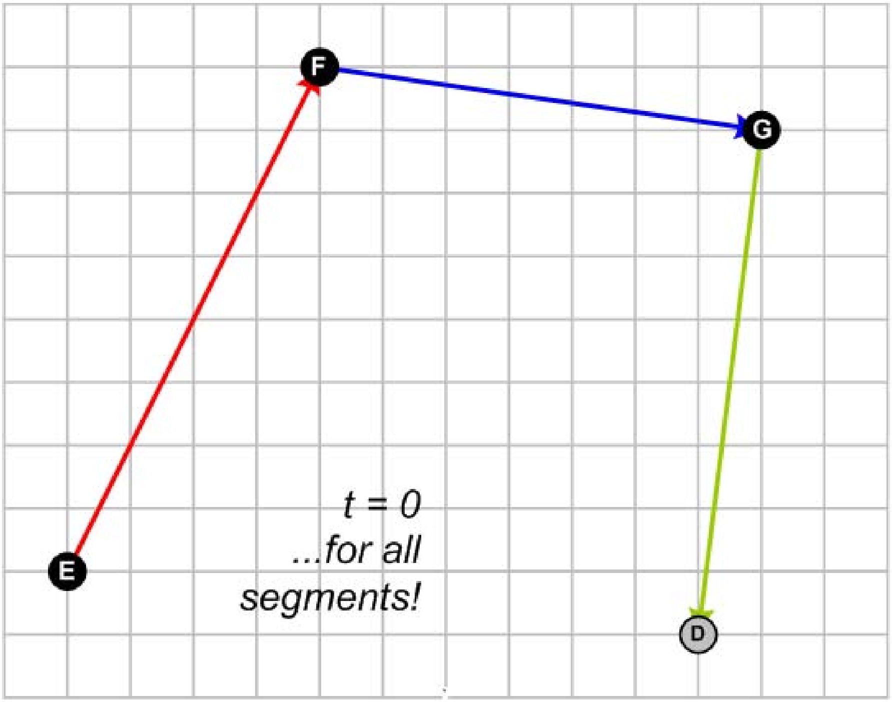
The word “cubic” means that if we sniff around the math long enough, we’ll see $\mathbf{t^3}$. (In our Linear Beziers we saw **t**; in our Quadratics we saw $\mathbf{t^2}$).

Cubic Bezier Curves



- Four **control points**: **A**, **B**, **C**, and **D**
- 2 different Quadratic Beziers: **ABC** and **BCD**
- 3 different Linear Beziers: **AB**, **BC**, and **CD**

Cubic Bezier Curves



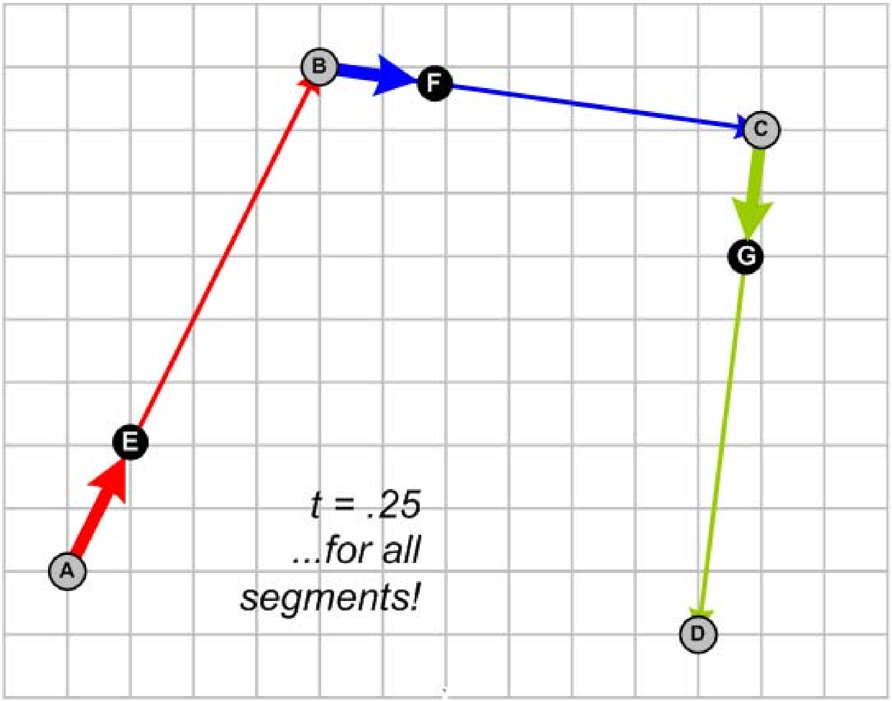
- As we turn the knob (one knob, one "t" for everyone):

  Interpolate **E** along **AB** // all three lerp simultaneously

  Interpolate **F** along **BC** // all three lerp simultaneously

  Interpolate **G** along **CD** // all three lerp simultaneously

Cubic Bezier Curves



t = .25
...for all segments!

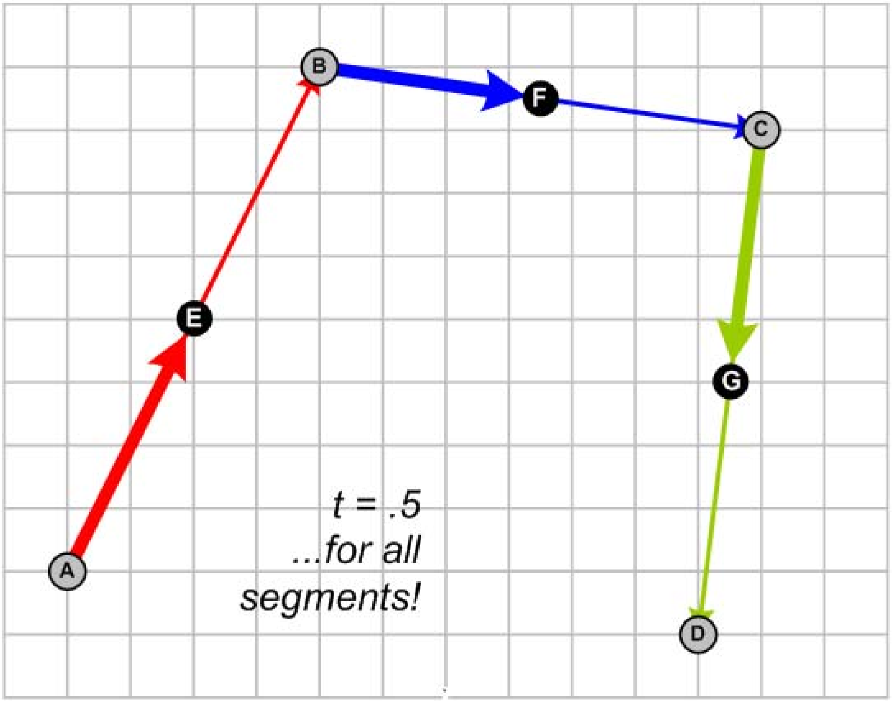
- As we turn the knob (one knob, one "t" for everyone):

  Interpolate **E** along **AB** // all three lerp simultaneously

  Interpolate **F** along **BC** // all three lerp simultaneously

  Interpolate **G** along **CD** // all three lerp simultaneously

Cubic Bezier Curves



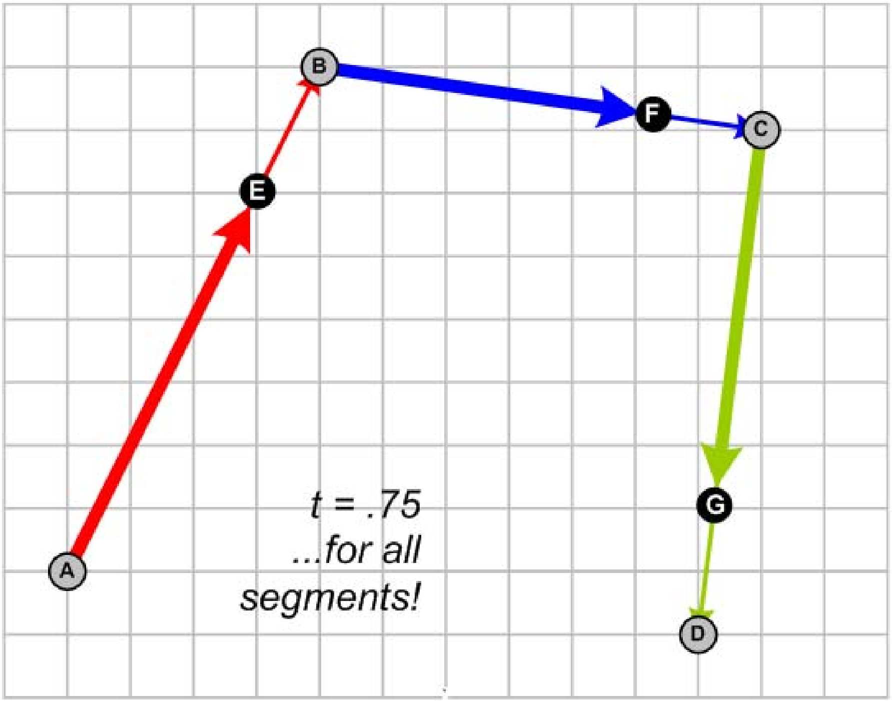
» As we turn the knob (one knob, one "t" for everyone):

Interpolate **E** along **AB** // all three lerp simultaneously

Interpolate **F** along **BC** // all three lerp simultaneously

Interpolate **G** along **CD** // all three lerp simultaneously

Cubic Bezier Curves



t = .75 ...for all segments!

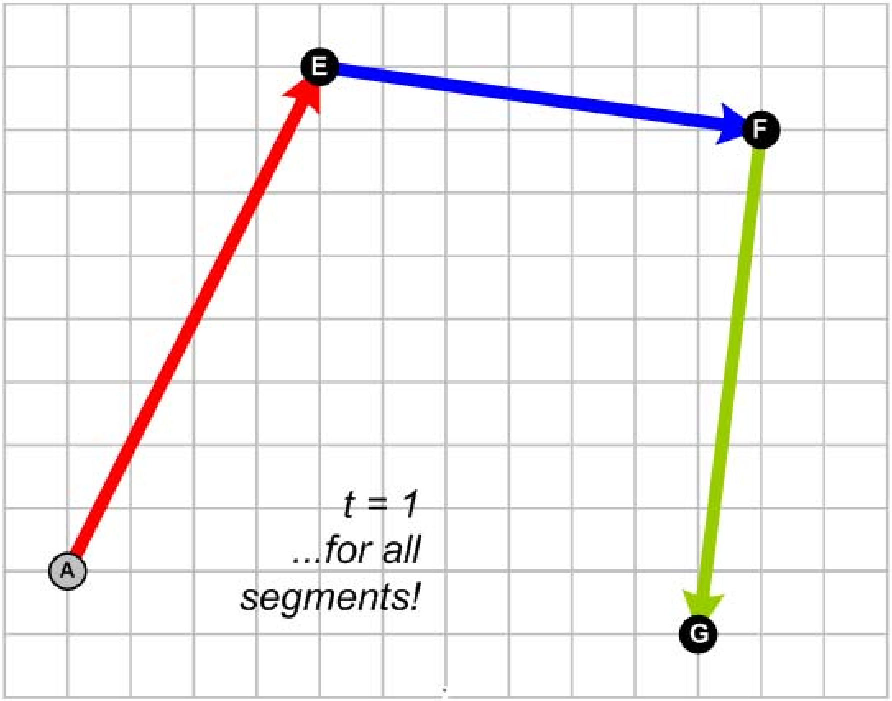
» As we turn the knob (one knob, one "t" for everyone):

Interpolate E along AB // all three lerp simultaneously

Interpolate F along BC // all three lerp simultaneously

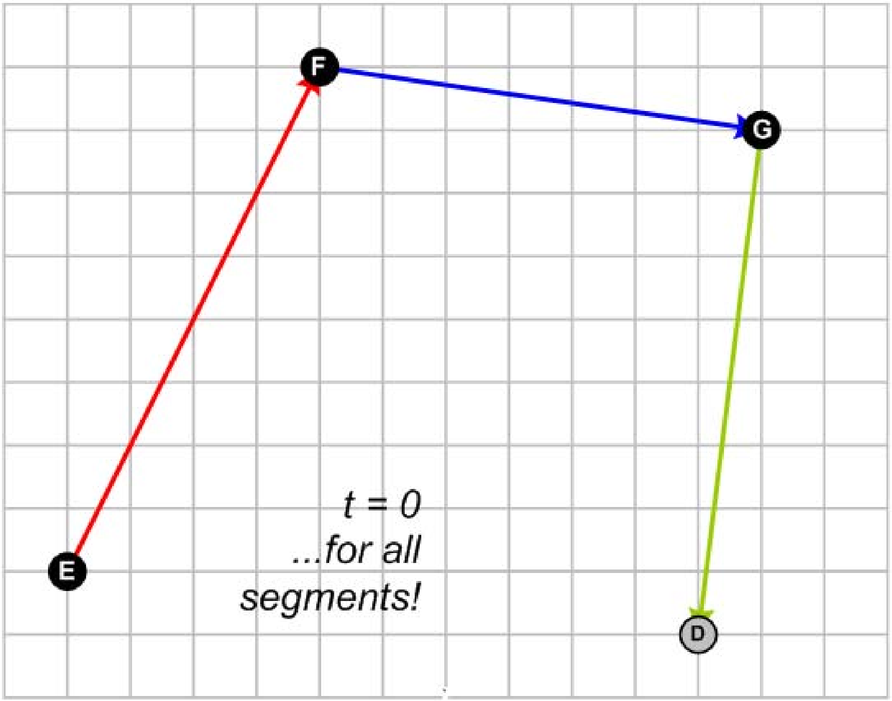
Interpolate G along CD // all three lerp simultaneously

Cubic Bezier Curves



- As we turn the knob (one knob, one "t" for everyone):
  - Interpolate **E** along **AB** // all three lerp simultaneously
  - Interpolate **F** along **BC** // all three lerp simultaneously
  - Interpolate **G** along **CD** // all three lerp simultaneously

Cubic Bezier Curves

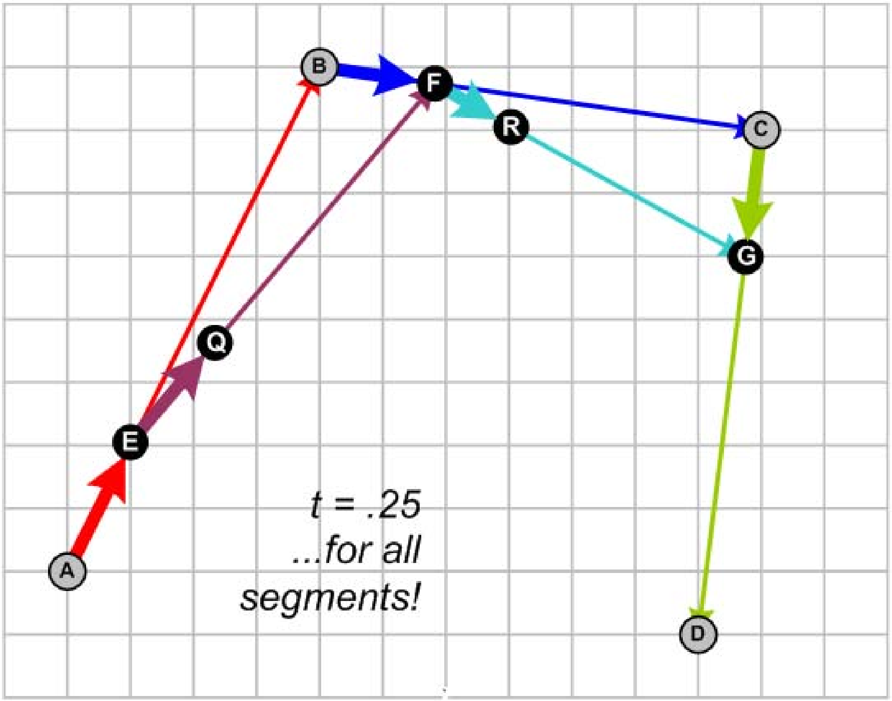


- Also:

  Interpolate **Q** along **EF** // lerp simultaneously with E,F,G

  Interpolate **R** along **FG** // lerp simultaneously with E,F,G

Cubic Bezier Curves

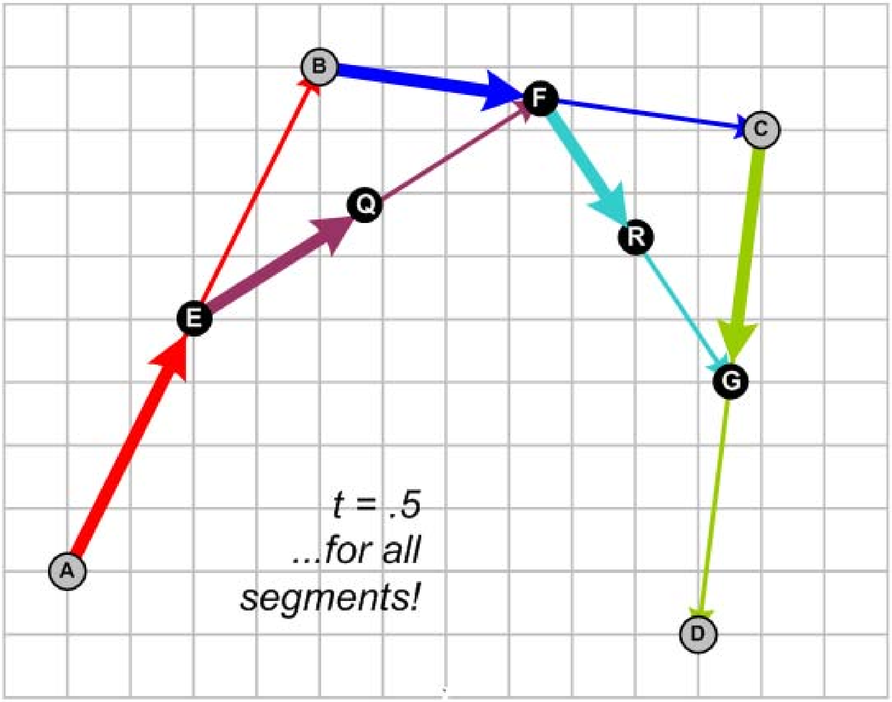


- Also:

  Interpolate **Q** along **EF** // lerp simultaneously with E,F,G

  Interpolate **R** along **FG** // lerp simultaneously with E,F,G

Cubic Bezier Curves

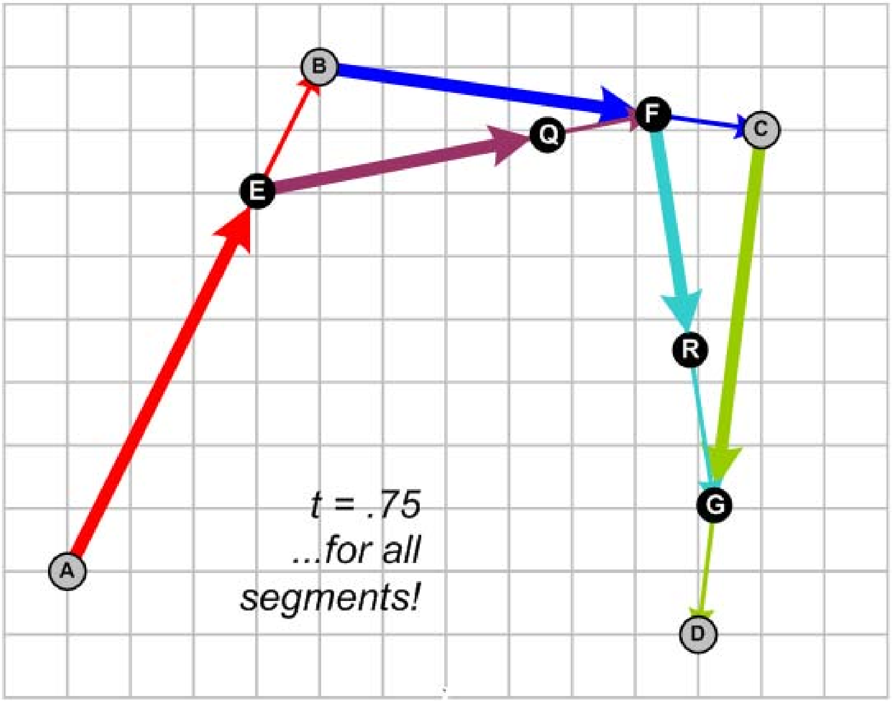


- Also:

  Interpolate **Q** along **EF**   // lerp simultaneously with E,F,G

  Interpolate **R** along **FG**   // lerp simultaneously with E,F,G

Cubic Bezier Curves

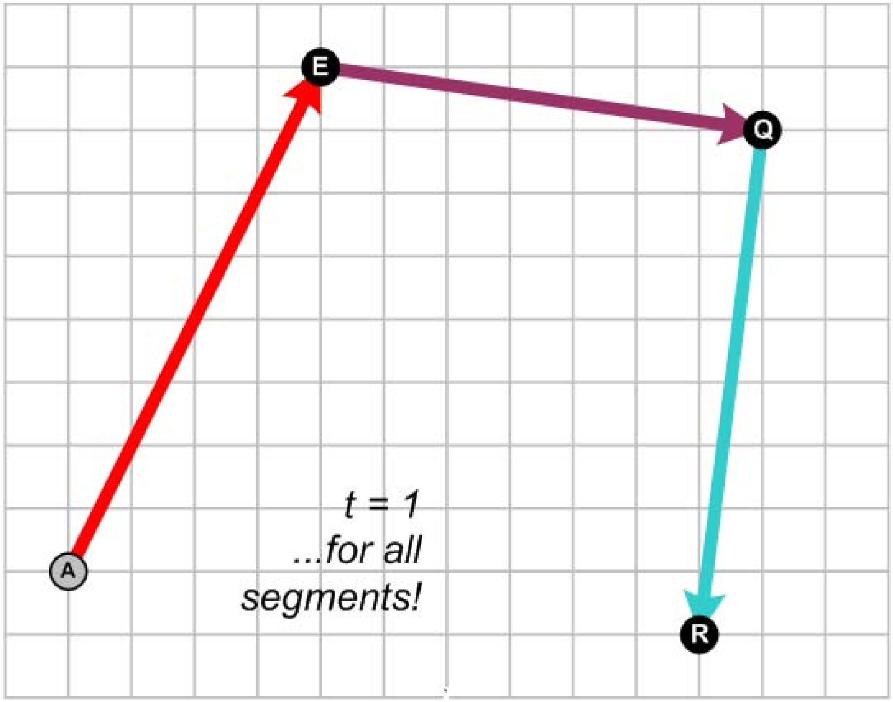


- Also:

  Interpolate **Q** along **EF** // lerp simultaneously with E,F,G

  Interpolate **R** along **FG** // lerp simultaneously with E,F,G

Cubic Bezier Curves

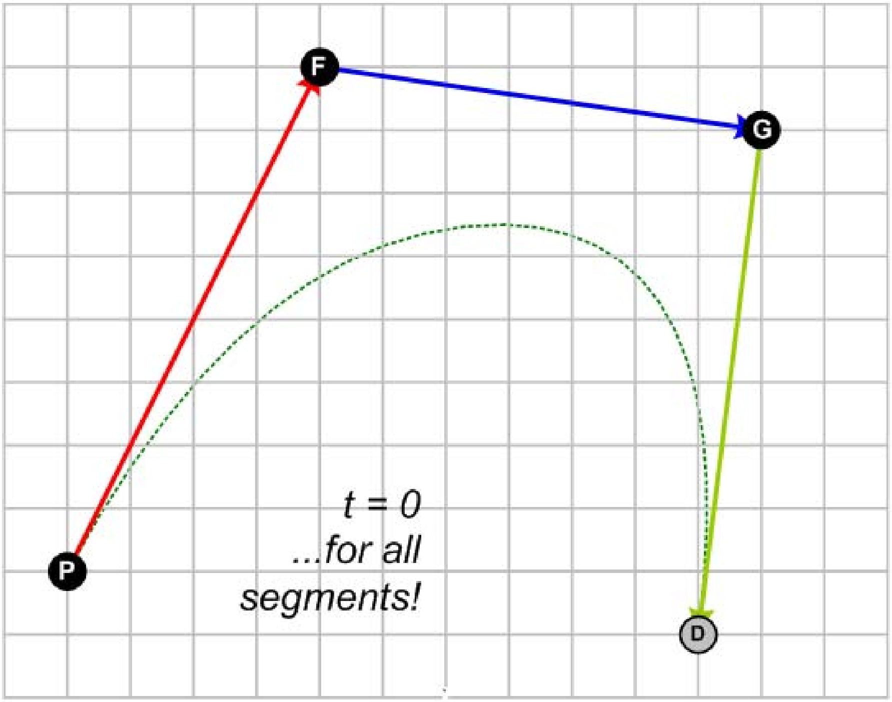


- Also:

  Interpolate **Q** along **EF** // lerp simultaneously with E,F,G

  Interpolate **R** along **FG** // lerp simultaneously with E,F,G

Cubic Bezier Curves



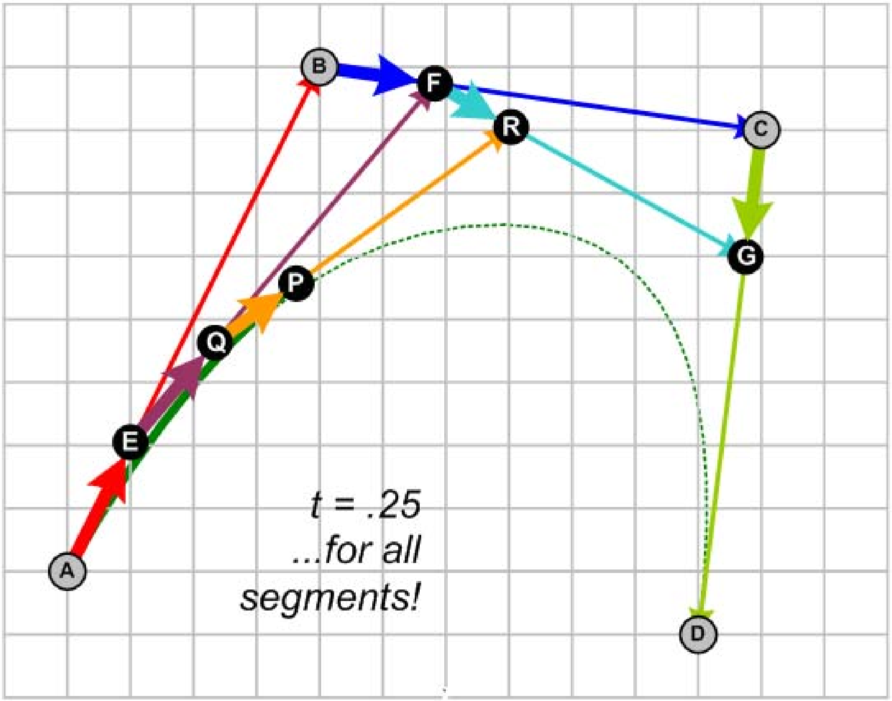
- And finally:

  Interpolate **P** along **QR**

  (simultaneously with E,F,G,Q,R)

- Again, watch **where P goes**!

Cubic Bezier Curves



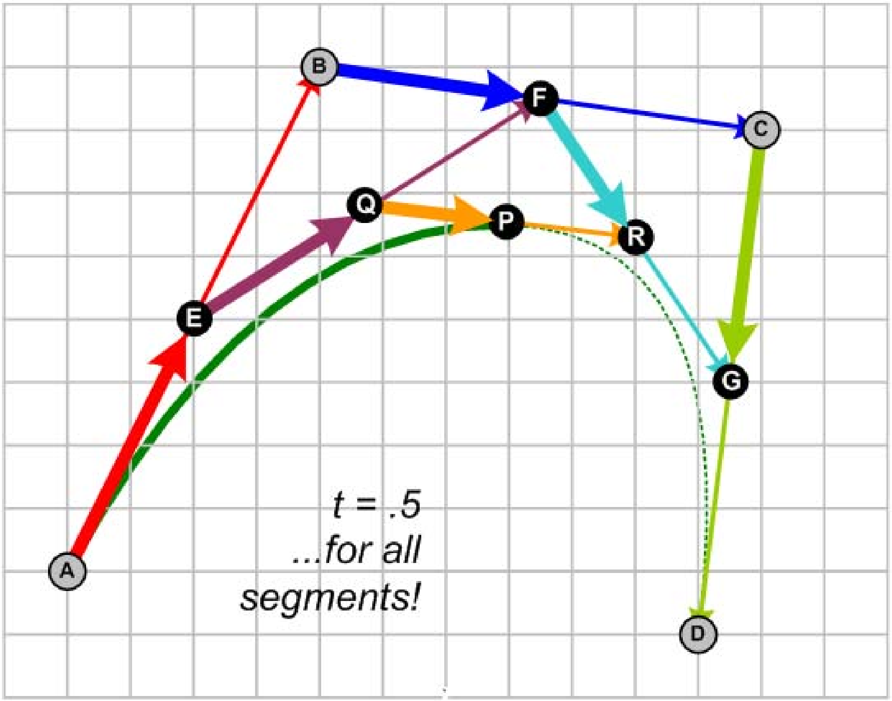
- And finally:

  Interpolate **P** along **QR**

  (simultaneously with E,F,G,Q,R)

- Again, watch **where P goes**!

Cubic Bezier Curves



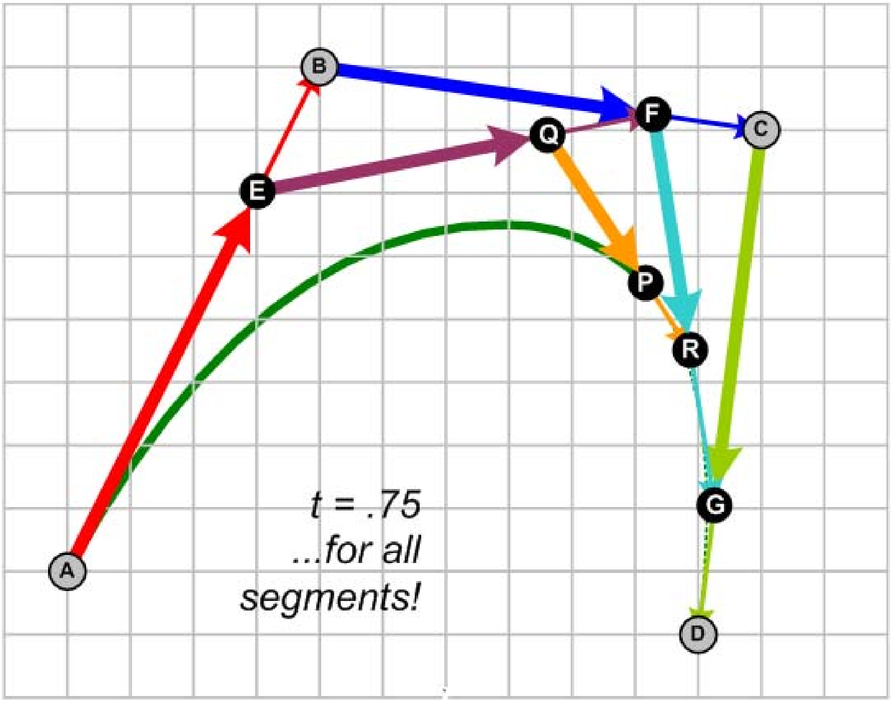
- » And finally:

  Interpolate **P** along **QR**

  (simultaneously with E,F,G,Q,R)

- » Again, watch **where P goes**!

Cubic Bezier Curves



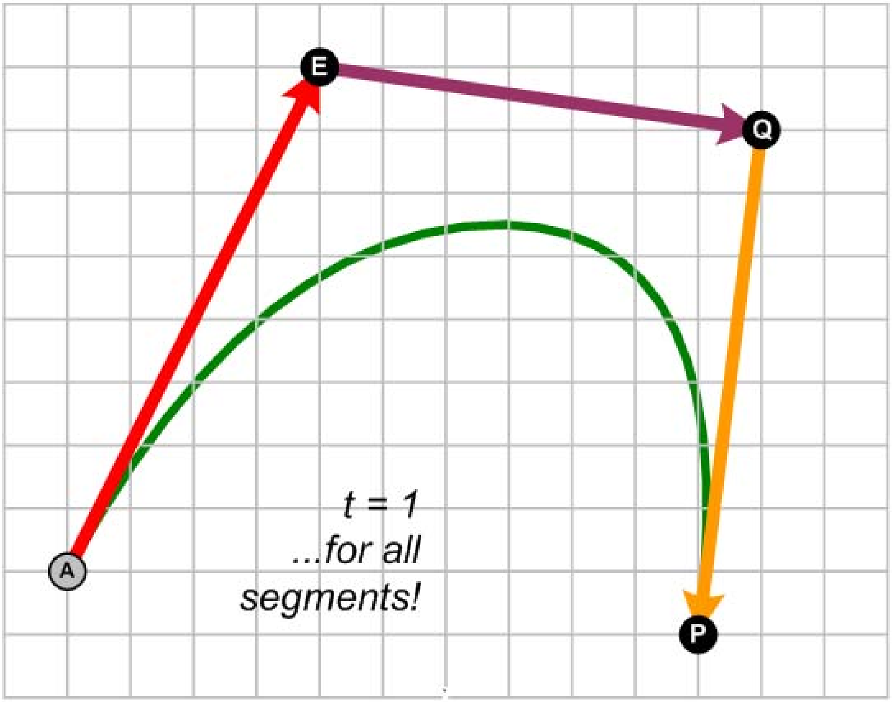
- And finally:

  Interpolate **P** along **QR**

  (simultaneously with E,F,G,Q,R)

- Again, watch **where P goes**!

Cubic Bezier Curves



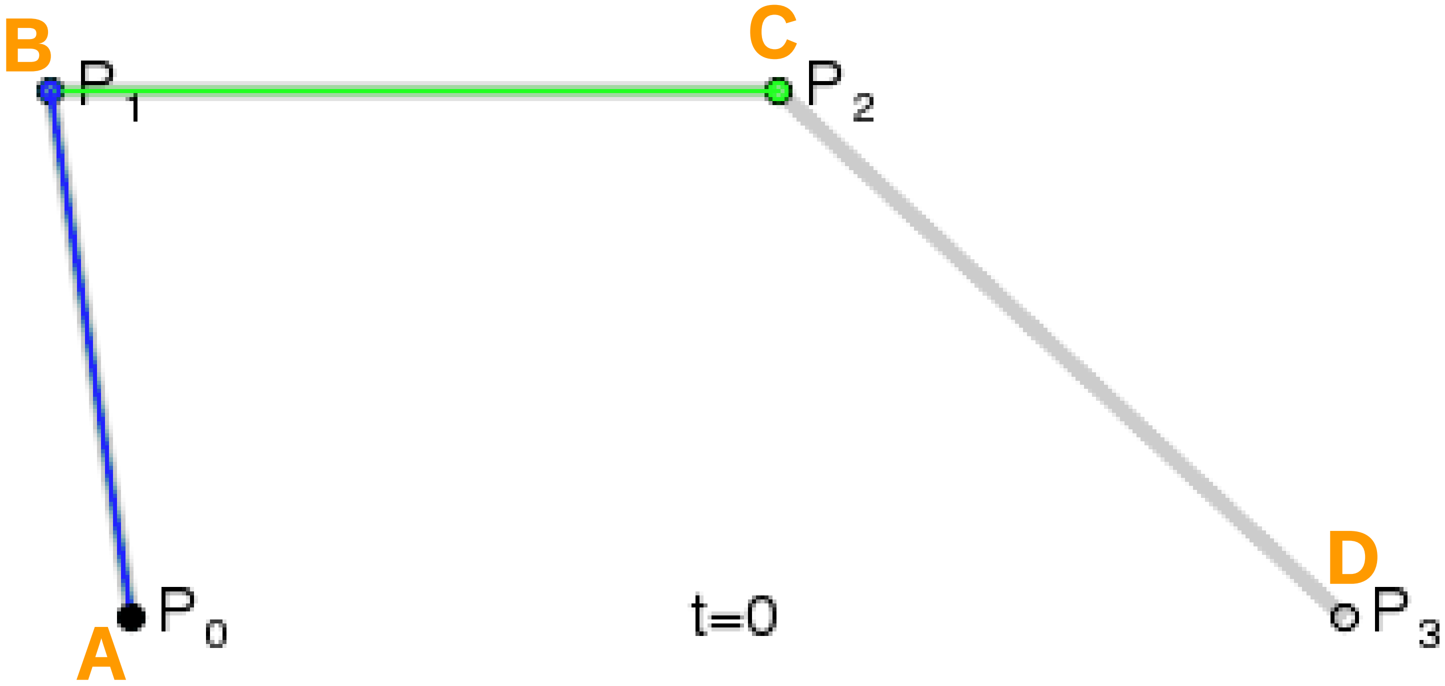
- And finally:

  Interpolate **P** along **QR**

  (simultaneously with E,F,G,Q,R)

- Again, watch **where P goes**!

Cubic Bezier Curves



- Now P starts at **A**, and ends at **D**
- It never touches **B** or **C**...

  since they are **guide points**

Cubic Bezier Curves

- Remember:

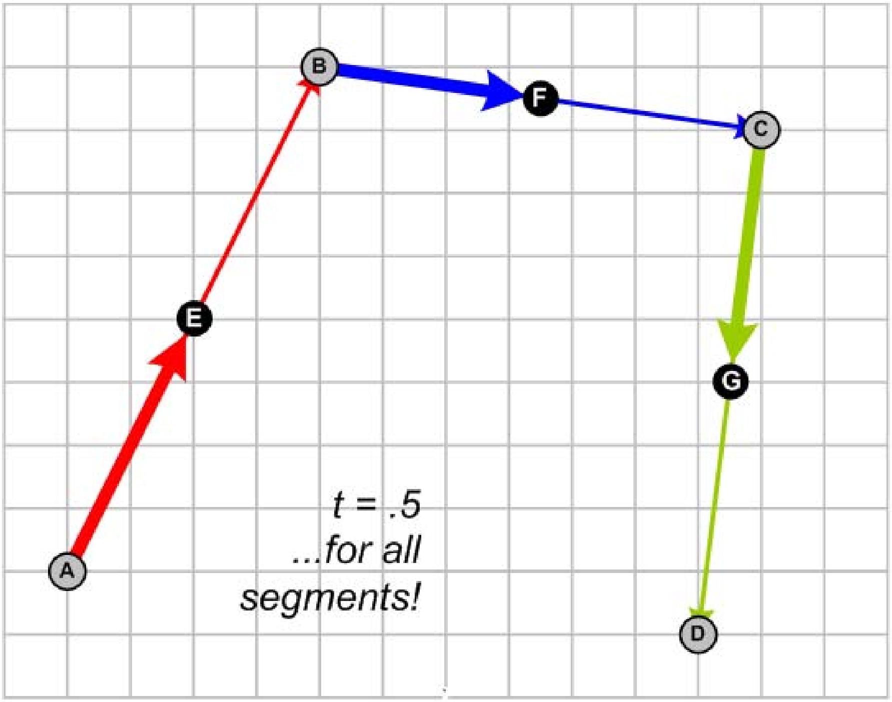
A Cubic Bezier curve is just a **blend of two Quadratic** Bezier curves.

Which are just a **blend of 3 Linear** Bezier curves.

So the math is still not too bad.

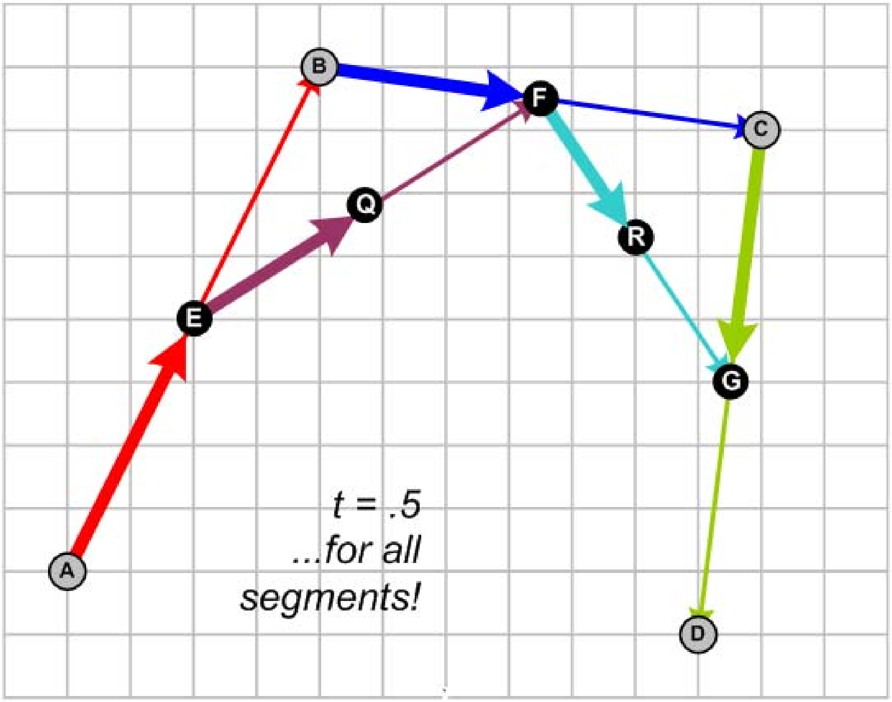
(A blend of blends of Linear Bezier equations.)

Cubic Bezier Curves



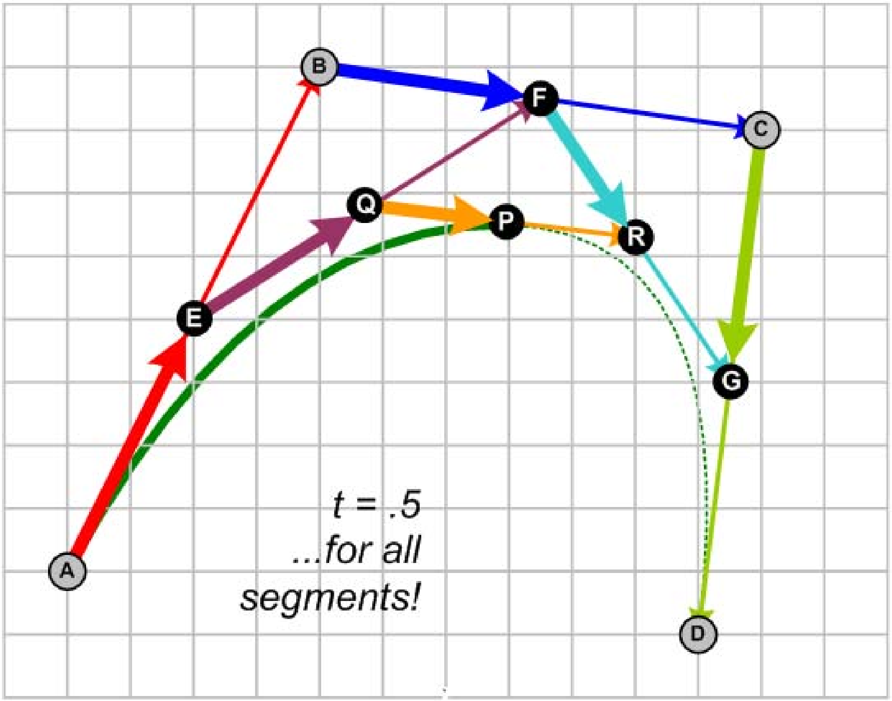
- E(t) = sA + tB ← *where* ***s*** *= 1-t*
- F(t) = sB + tC
- G(t) = sC + tD

Cubic Bezier Curves



- » And then **Q** and **R** interpolate those results…
- » **Q**(t) = s**E** + t**F**
- » **R**(t) = s**F** + t**G**

Cubic Bezier Curves



» And lastly **P** interpolates from **Q** to **R**

» **P**(t) = s**Q** + t**R**

Cubic Bezier Curves

- $E(t) = sA + tB$ // Linear Bezier (blend of A and B)
- $F(t) = sB + tC$ // Linear Bezier (blend of B and C)
- $G(t) = sC + tD$ // Linear Bezier (blend of C and D)

- $Q(t) = sE + tF$ // Quadratic Bezier (blend of E and F)
- $R(t) = sF + tG$ // Quadratic Bezier (blend of F and G)

- $P(t) = sQ + tR$ // Cubic Bezier (blend of Q and R)

- Okay! So let’s combine these all together…

Cubic Bezier Curves

- Do some hand-waving mathemagic here...

...and we get **one equation to rule them all**:

$$\mathbf{P(t)} = (s^3)\mathbf{A} + 3(s^2t)\mathbf{B} + 3(st^2)\mathbf{C} + (t^3)\mathbf{D}$$

(BTW, there's our "cubic" $\mathbf{t^3}$)

Cubic Bezier Curves

- Let’s compare the three Bezier equations (Linear, Quadratic, Cubic):

$$\mathbf{P(t)} = (s)\mathbf{A} + (t)\mathbf{B}$$

$$\mathbf{P(t)} = (s^2)\mathbf{A} + 2(st)\mathbf{B} + (t^2)\mathbf{C}$$

$$\mathbf{P(t)} = (s^3)\mathbf{A} + 3(s^2t)\mathbf{B} + 3(st^2)\mathbf{C} + (t^3)\mathbf{D}$$

- There’s some nice symmetry here...

Cubic Bezier Curves

- Write in all of the numeric coefficients...
- Express each term as powers of **s** and **t**

$$P(t) = \mathbf{1}(s^1t^0)\mathbf{A} + \mathbf{1}(s^0t^1)\mathbf{B}$$

$$P(t) = \mathbf{1}(s^2t^0)\mathbf{A} + \mathbf{2}(s^1t^1)\mathbf{B} + \mathbf{1}(s^0t^2)\mathbf{C}$$

$$P(t) = \mathbf{1}(s^3t^0)\mathbf{A} + \mathbf{3}(s^2t^1)\mathbf{B} + \mathbf{3}(s^1t^2)\mathbf{C} + \mathbf{1}(s^0t^3)\mathbf{D}$$

Cubic Bezier Curves

- Write in all of the numeric coefficients...
- Express each term as powers of **s** and **t**

$$P(t) = \mathbf{1}(s^{1}t^{0})\mathbf{A} + \mathbf{1}(s^{0}t^{1})\mathbf{B}$$

$$P(t) = \mathbf{1}(s^{2}t^{0})\mathbf{A} + \mathbf{2}(s^{1}t^{1})\mathbf{B} + \mathbf{1}(s^{0}t^{2})\mathbf{C}$$

$$P(t) = \mathbf{1}(s^{3}t^{0})\mathbf{A} + \mathbf{3}(s^{2}t^{1})\mathbf{B} + \mathbf{3}(s^{1}t^{2})\mathbf{C} + \mathbf{1}(s^{0}t^{3})\mathbf{D}$$

- Note: “s” exponents count down

Cubic Bezier Curves

- » Write in all of the numeric coefficients...
- » Express each term as powers of **s** and **t**

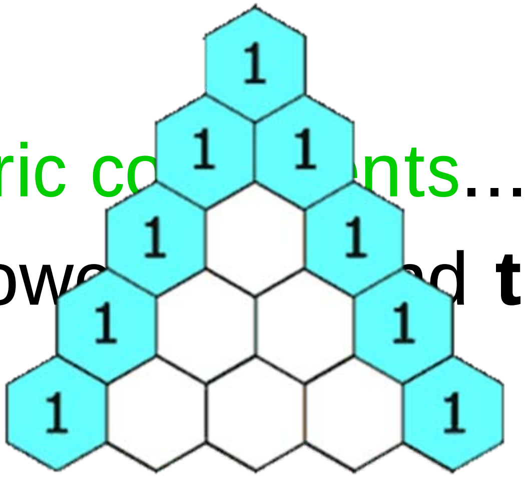
$$P(t) = \mathbf{1}(s^1\mathbf{t^0})\mathbf{A} + \mathbf{1}(s^0\mathbf{t^1})\mathbf{B}$$

$$P(t) = \mathbf{1}(s^2\mathbf{t^0})\mathbf{A} + \mathbf{2}(s^1\mathbf{t^1})\mathbf{B} + \mathbf{1}(s^0\mathbf{t^2})\mathbf{C}$$

$$P(t) = \mathbf{1}(s^3\mathbf{t^0})\mathbf{A} + \mathbf{3}(s^2\mathbf{t^1})\mathbf{B} + \mathbf{3}(s^1\mathbf{t^2})\mathbf{C} + \mathbf{1}(s^0\mathbf{t^3})\mathbf{D}$$

- » Note: “s” exponents count down
- » Note: “t” exponents count up

Cubic Bezier Curves



- Write in all of the numeric co nts…
- Express each term as pow d **t**

$$P(t) = \mathbf{1}(s^1t^0)\mathbf{A} + \mathbf{1}(s^0t^1)\mathbf{B}$$

$$P(t) = \mathbf{1}(s^2t^0)\mathbf{A} + \mathbf{2}(s^1t^1)\mathbf{B} + \mathbf{1}(s^0t^2)\mathbf{C}$$

$$P(t) = \mathbf{1}(s^3t^0)\mathbf{A} + \mathbf{3}(s^2t^1)\mathbf{B} + \mathbf{3}(s^1t^2)\mathbf{C} + \mathbf{1}(s^0t^3)\mathbf{D}$$

- Note: numeric coefficients are from Pascal’s Triangle…

Cubic Bezier Curves

» What if *t = 0.5* ? (halfway through the curve)
so then... s = 0.5 also

$$P(t) = (s^3)A + 3(s^2t)B + 3(st^2)C + (t^3)D$$

becomes

$$P(t) = (.5^3)A + 3(.5^2 * .5)B + 3(.5 * .5^2)C + (.5^3)D$$

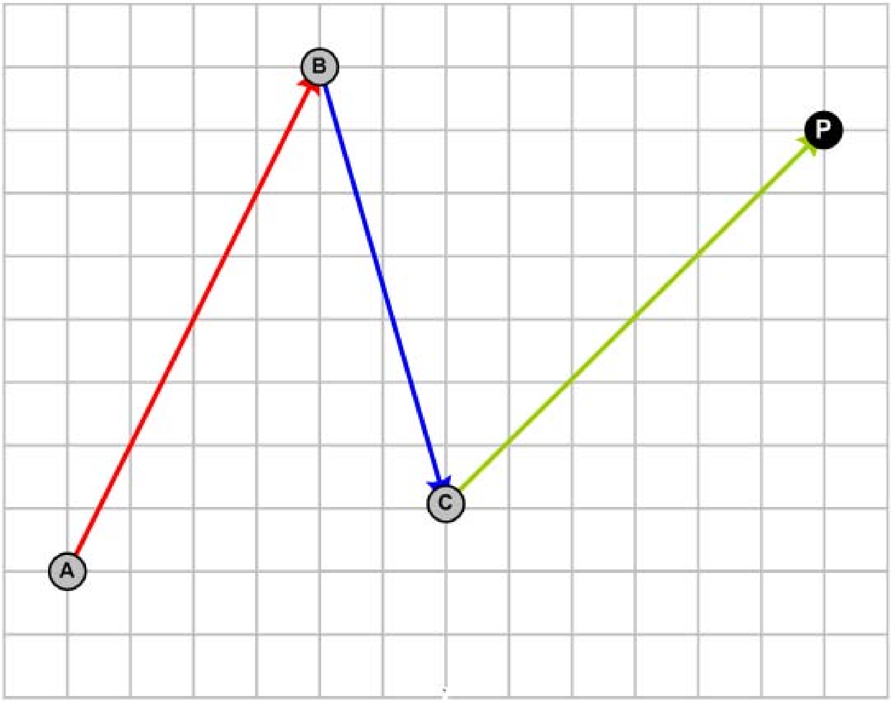
becomes

$$P(t) = (.125)A + 3(.125)B + 3(.125)C + (.125)D$$

becomes

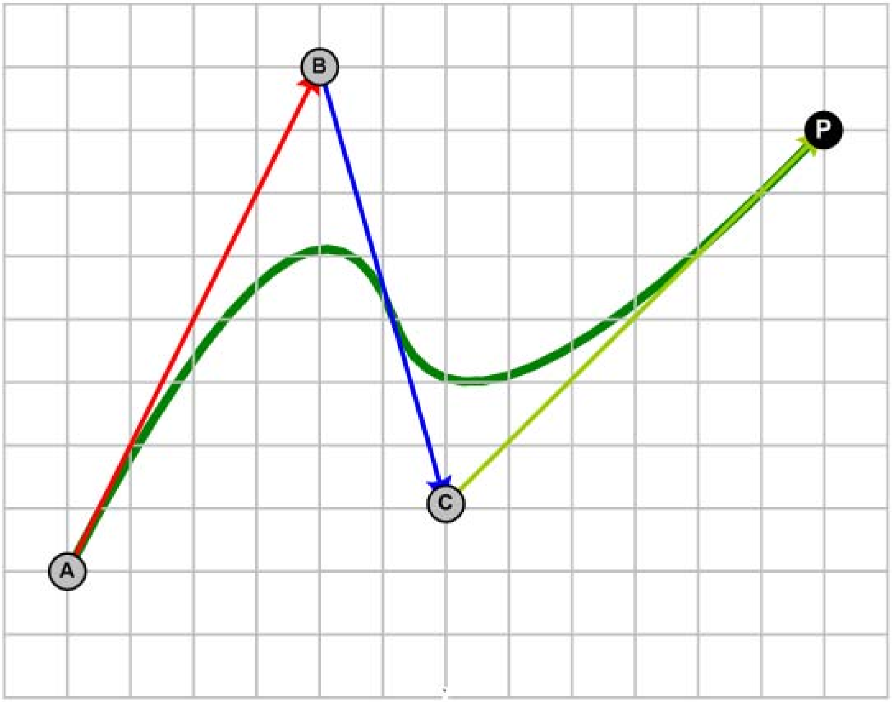
$$P(t) = .125A + .375B + .375C + .125D$$

Cubic Bezier Curves



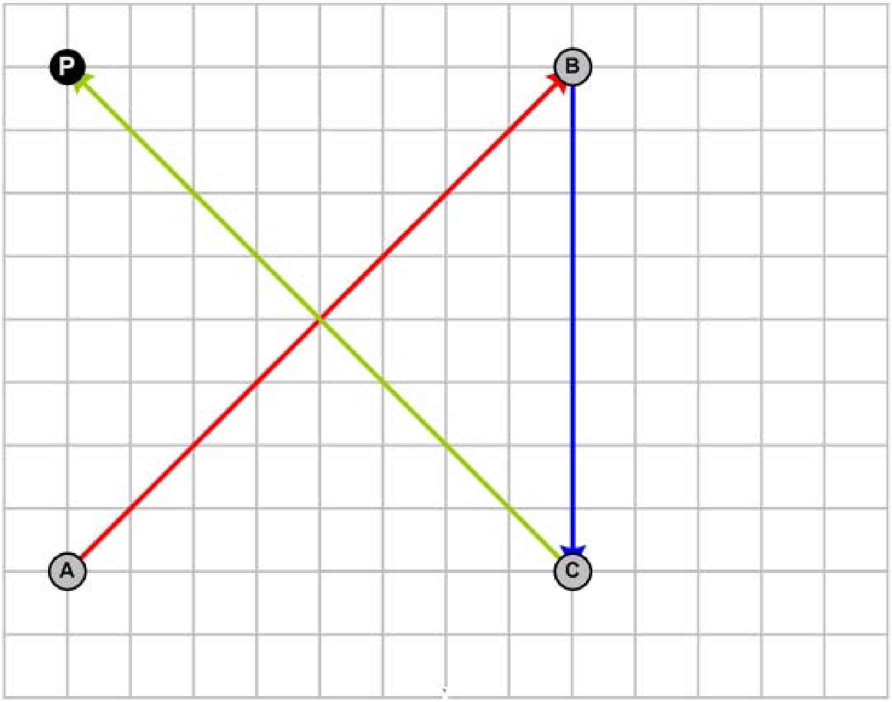
- Cubic Bezier Curves can also be “S-shaped”, if their control points are “twisted” as pictured here.

Cubic Bezier Curves



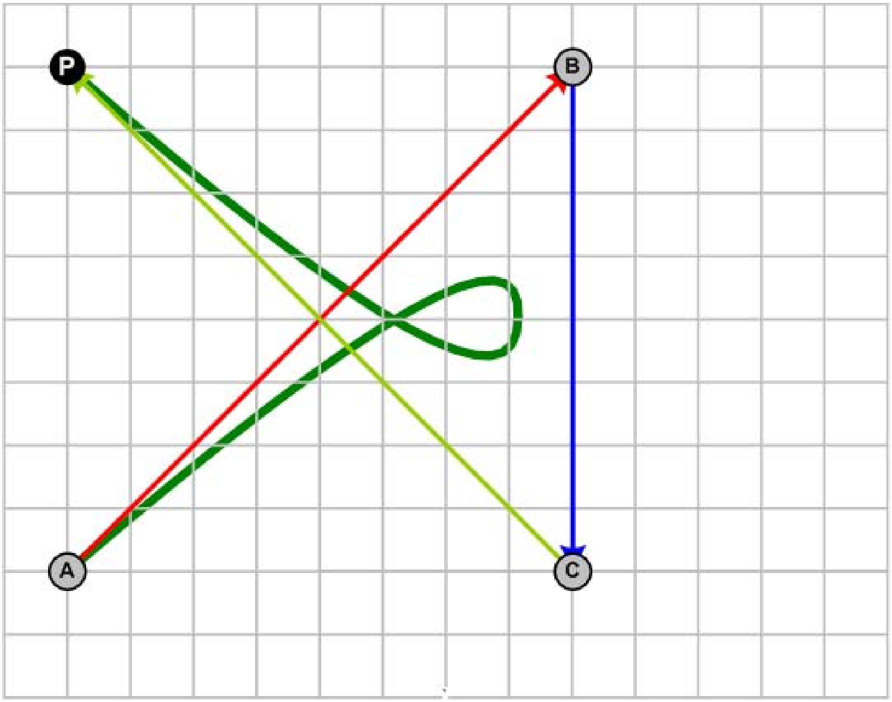
- Cubic Bezier Curves can also be “S-shaped”, if their control points are “twisted” as pictured here.

Cubic Bezier Curves



- They can also loop back around in extreme cases.

Cubic Bezier Curves



- They can also loop back around in extreme cases.

Cubic Bezier Curves

Seen in lots of places:

- Photoshop
- GIMP
- PostScript
- Flash
- AfterEffects
- 3DS Max
- Metafont

- Understable Disc Golf flight path, from above

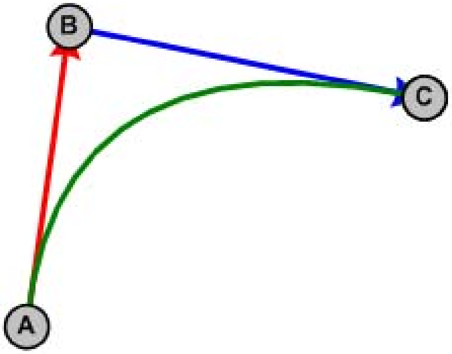
Splines

Splines

- Okay, enough of Curves already.
- So… what’s a Spline?

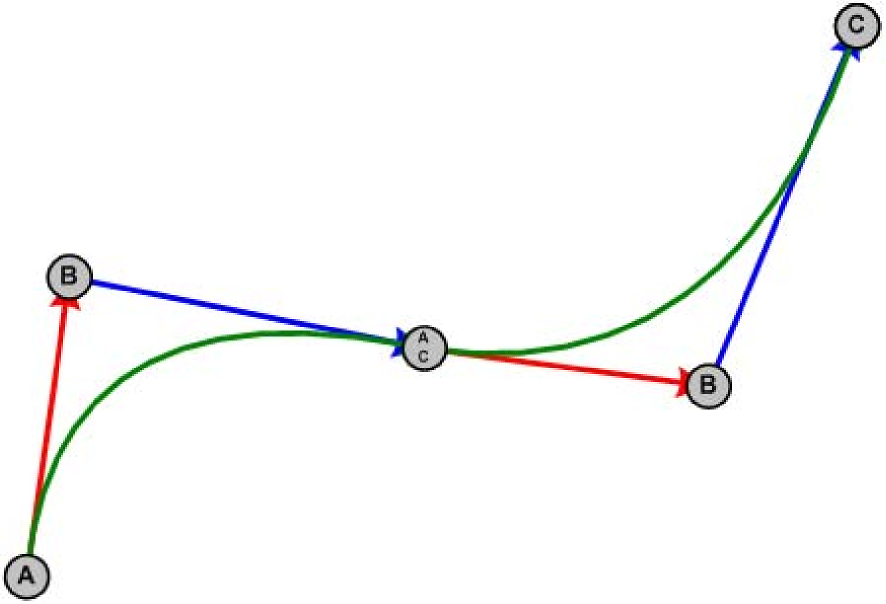
Splines

A **spline** is a chain of curves joined end-to-end.



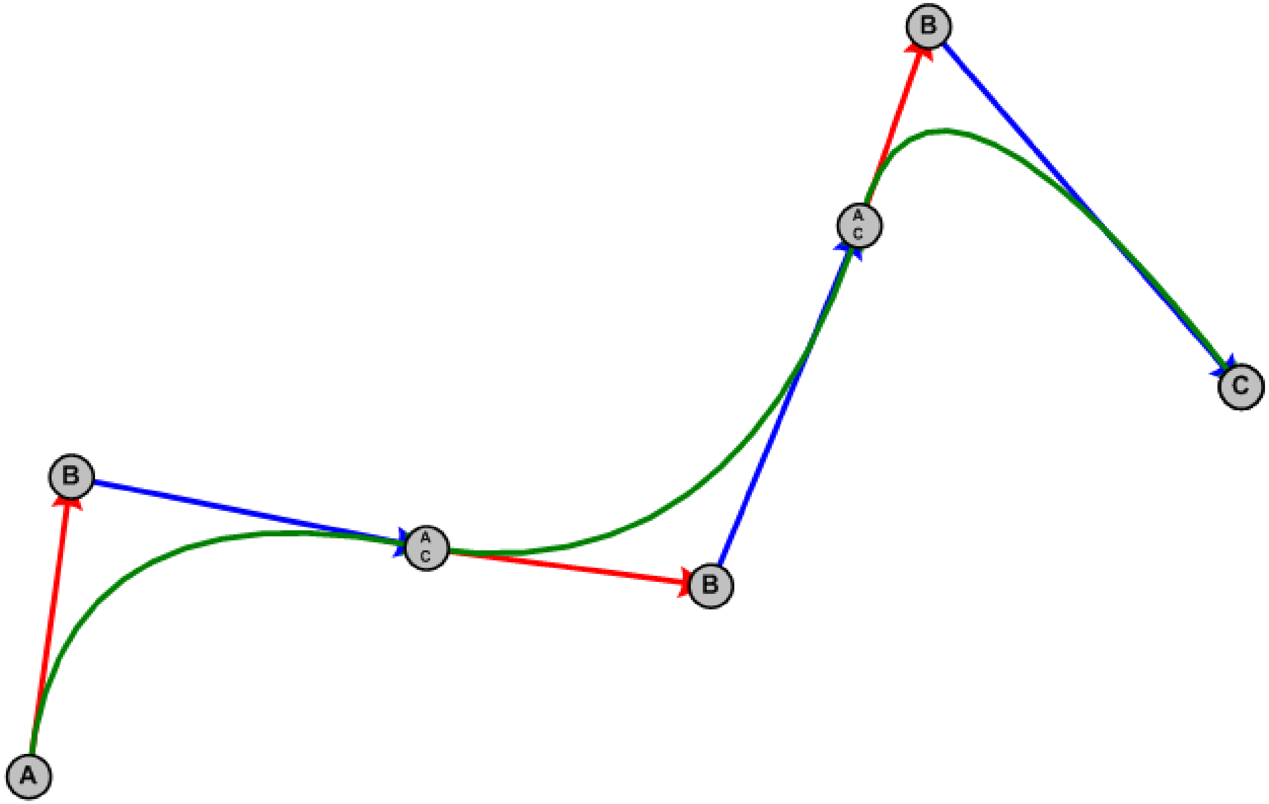
Splines

A **spline** is a chain of curves joined end-to-end.



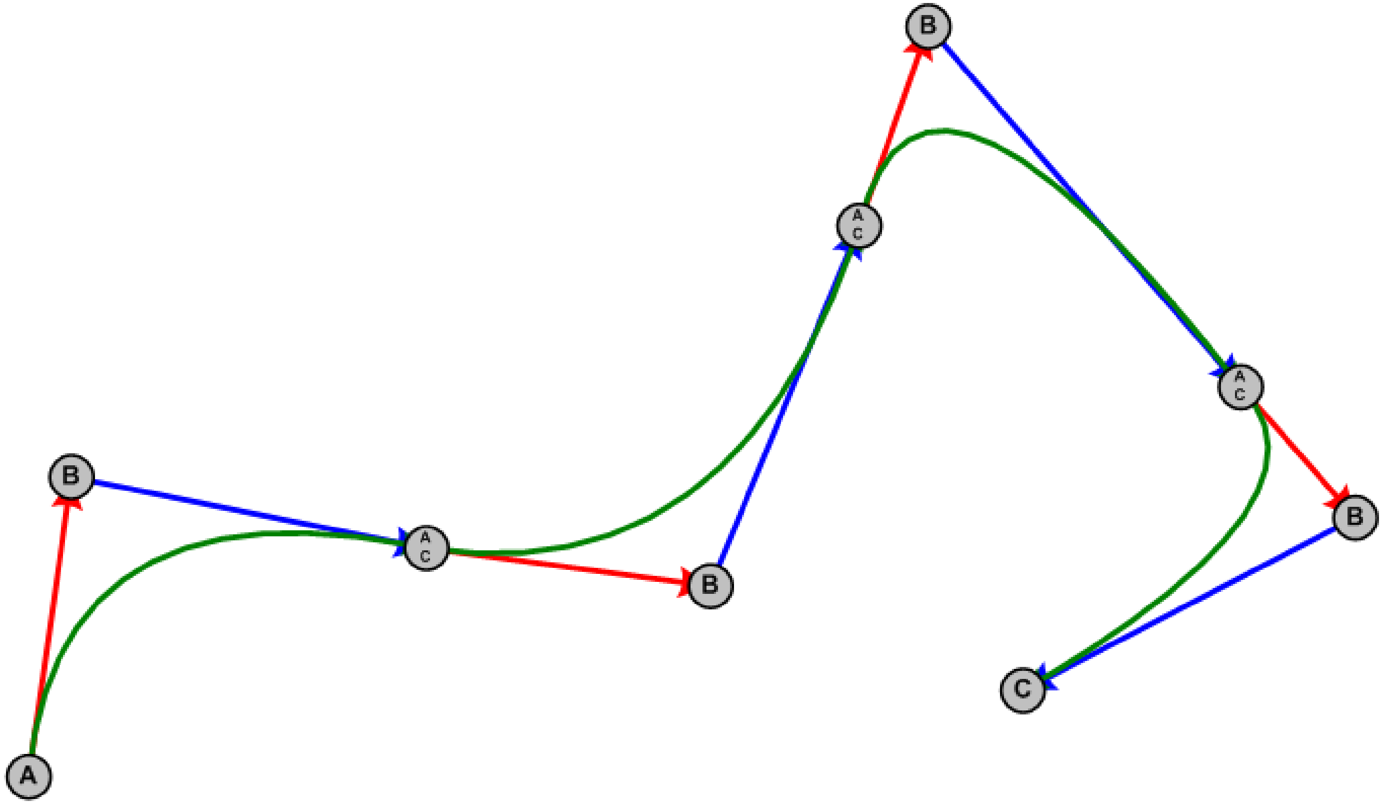
Splines

A **spline** is a chain of curves joined end-to-end.



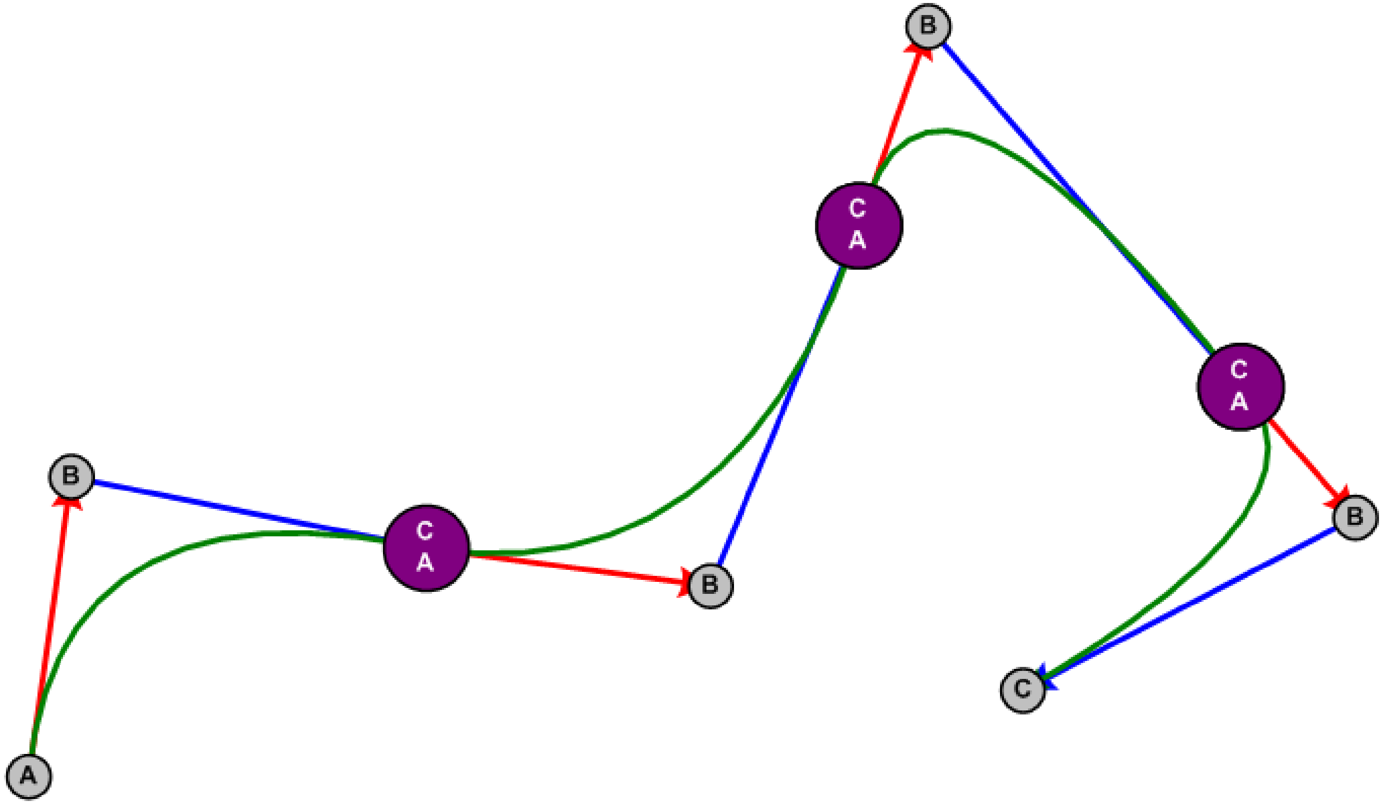
Splines

A **spline** is a chain of curves joined end-to-end.



Splines

- Curve end/start points (welds) are **knots**



Splines

- Think of **two** different **t**s:

  spline's t: Zero at start of spline, keeps increasing until the end of the spline chain

  local curve's t: Resets to 0 at start of each curve (at each knot).

- Conventionally, the local curve's t is

```
fmod( spline_t, 1.0 )
```

Splines

For a spline of 4 curve-pieces:

» Interpolate **spline_t** from 0.0 to 4.0

» If **spline_t** is 2.67, then we are:

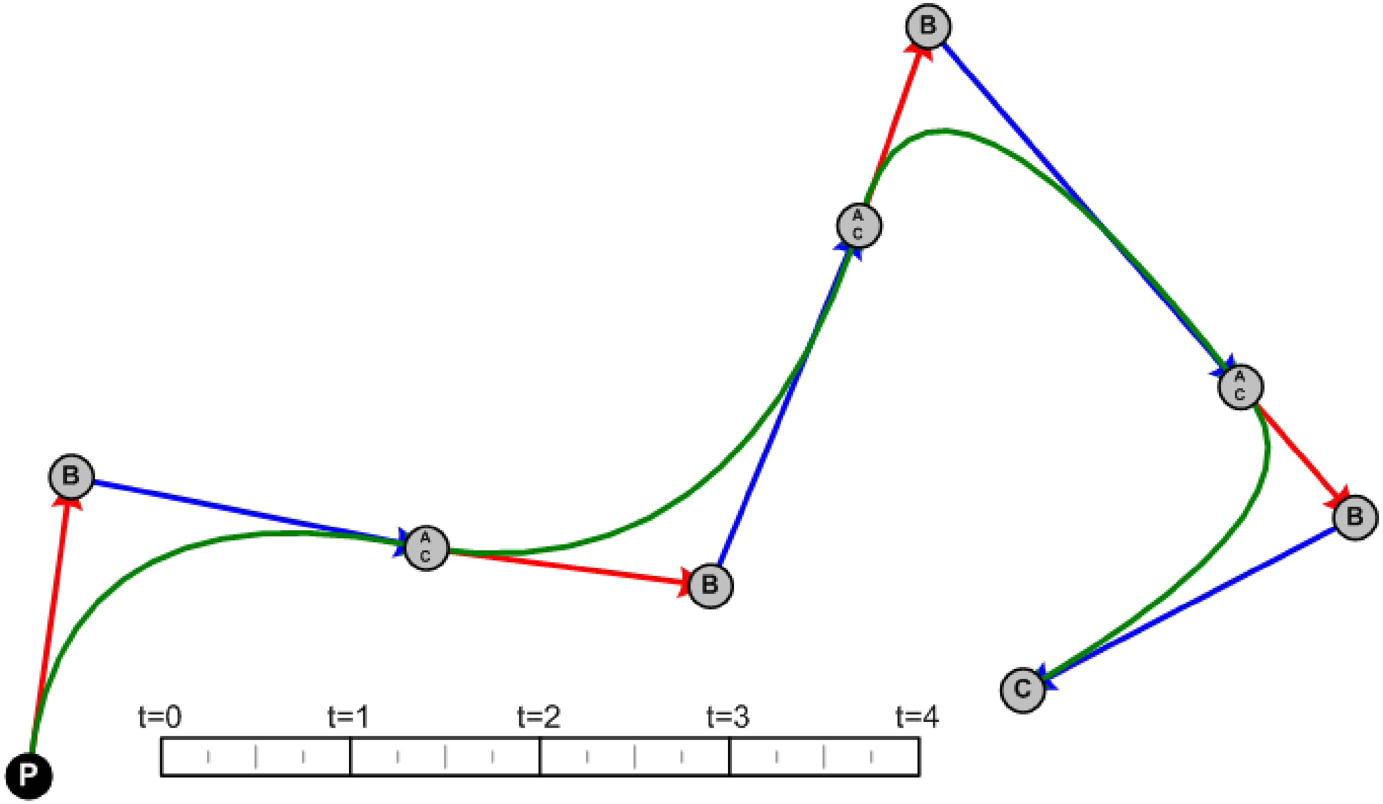
67% through this curve (**local_t** = .67)

In the third curve section (0,1,**2**,3)

» Plug **local_t** into third curve equation

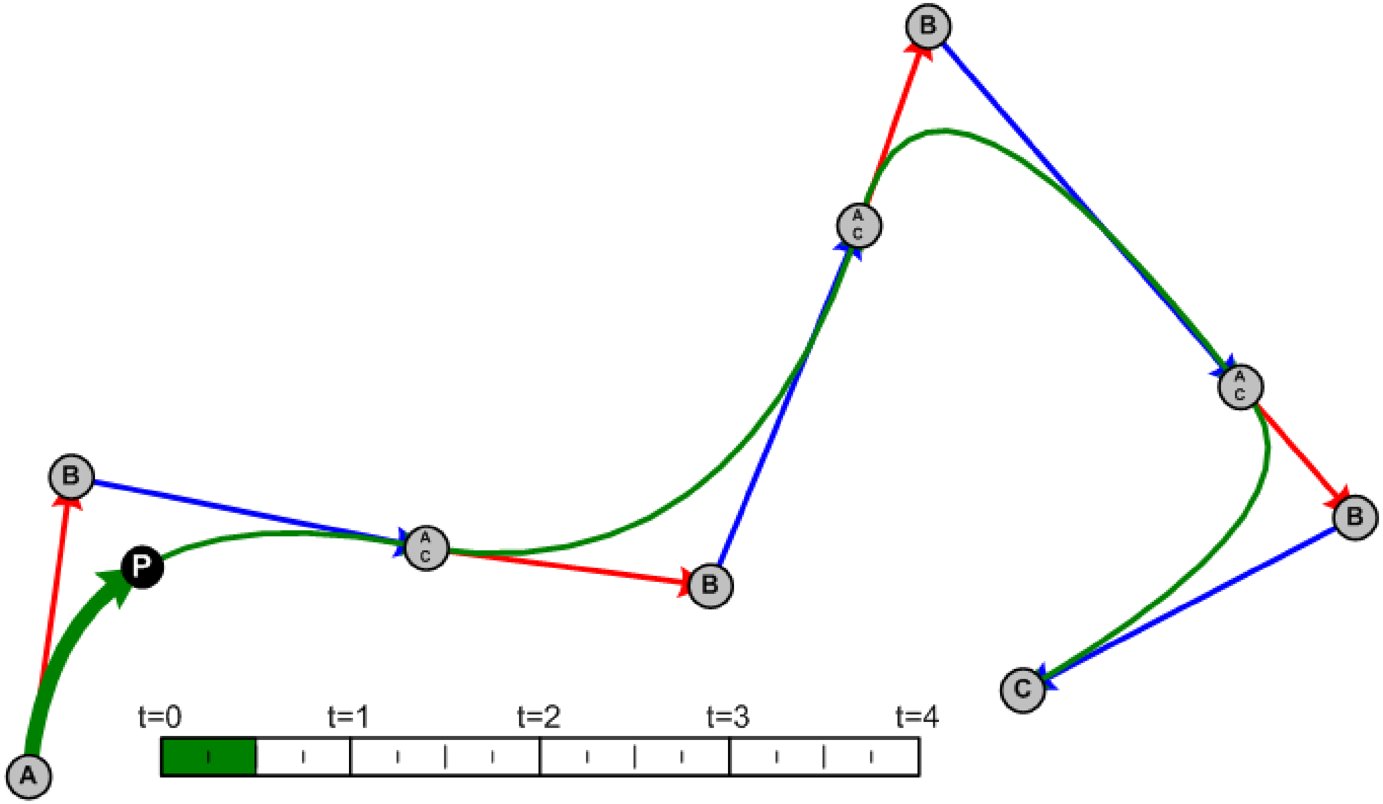
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



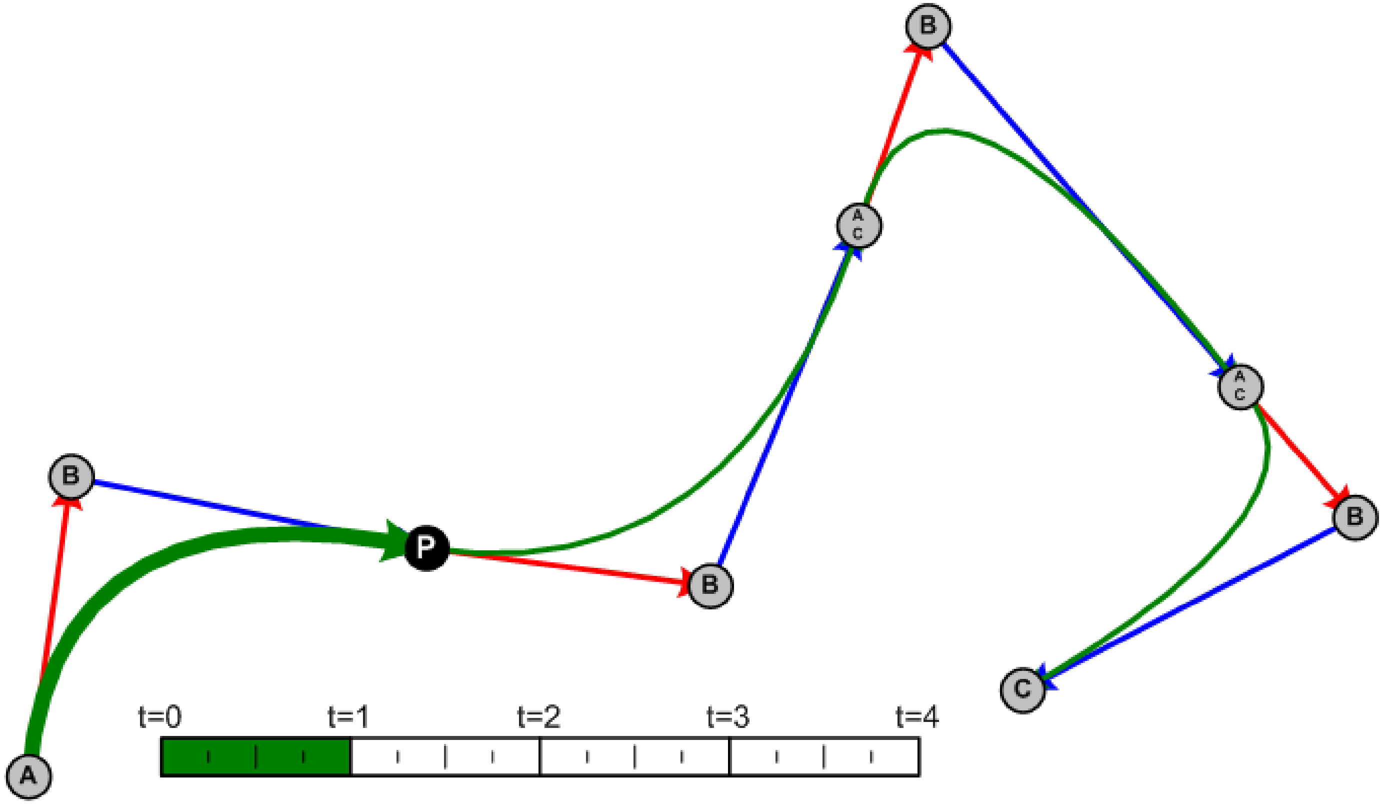
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



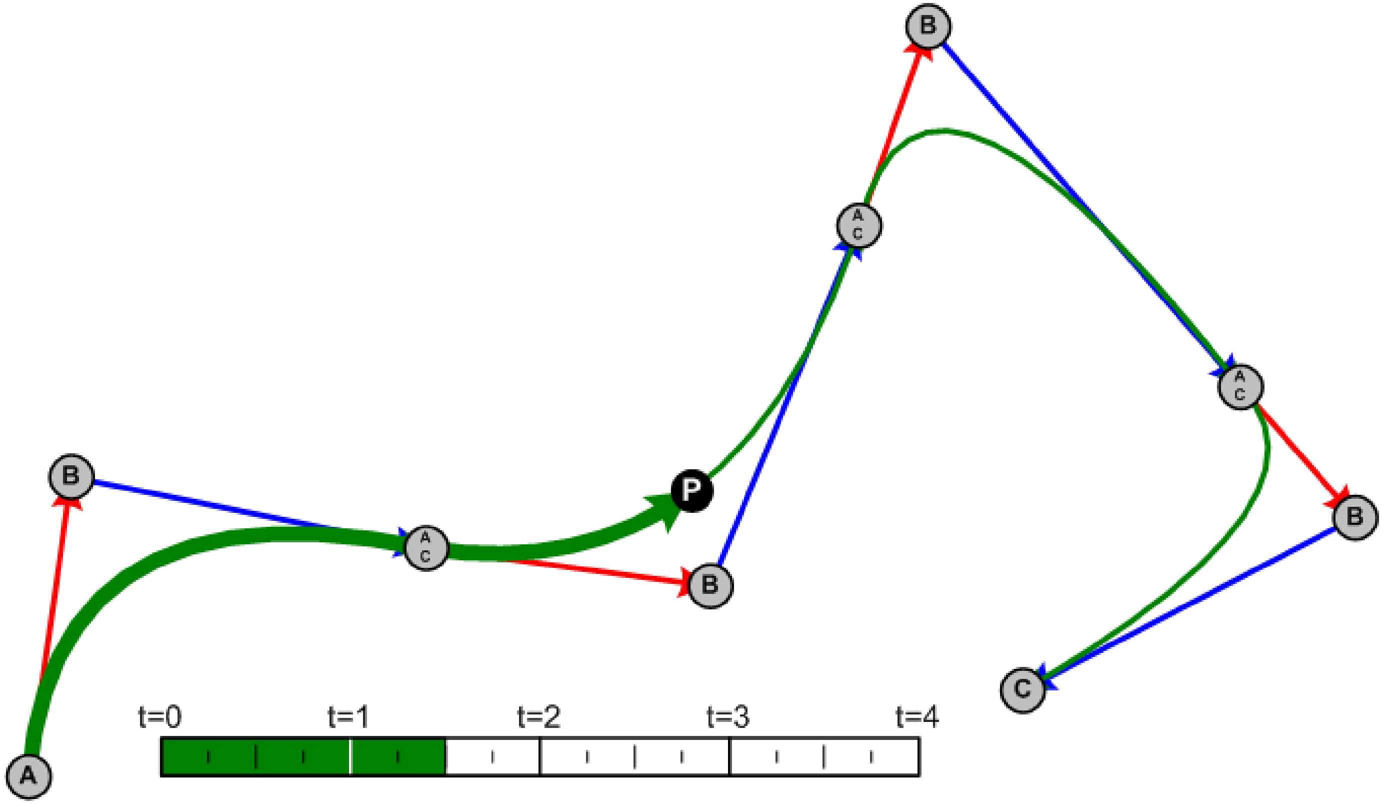
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



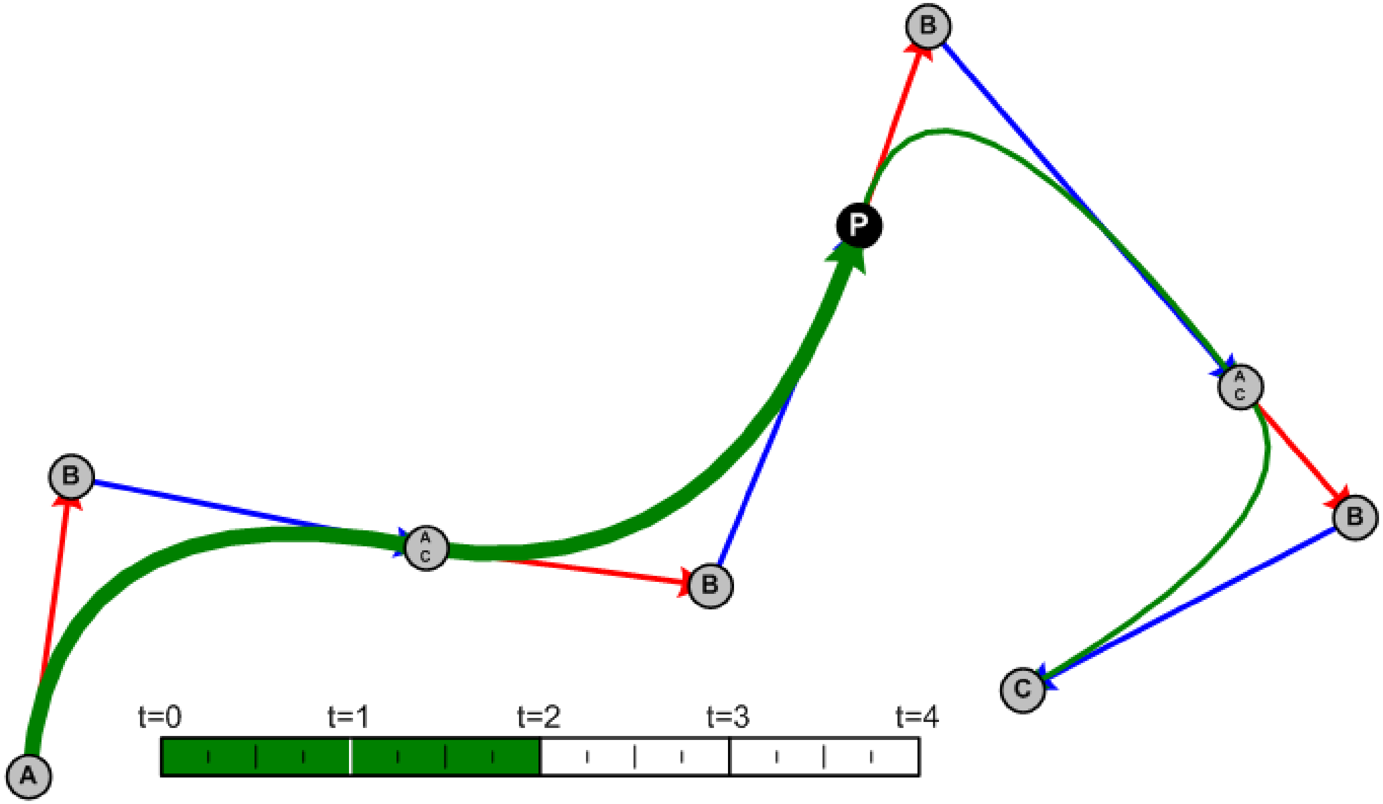
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



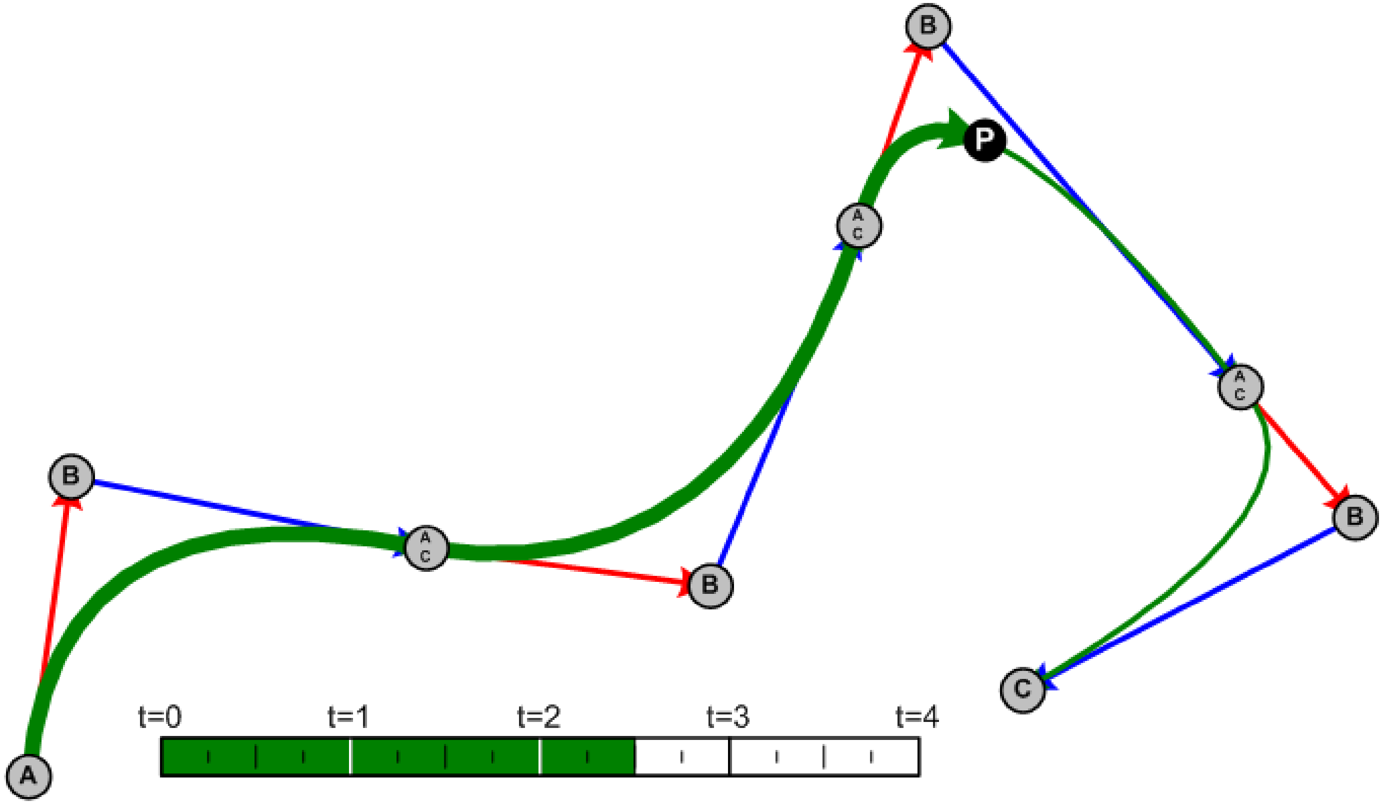
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



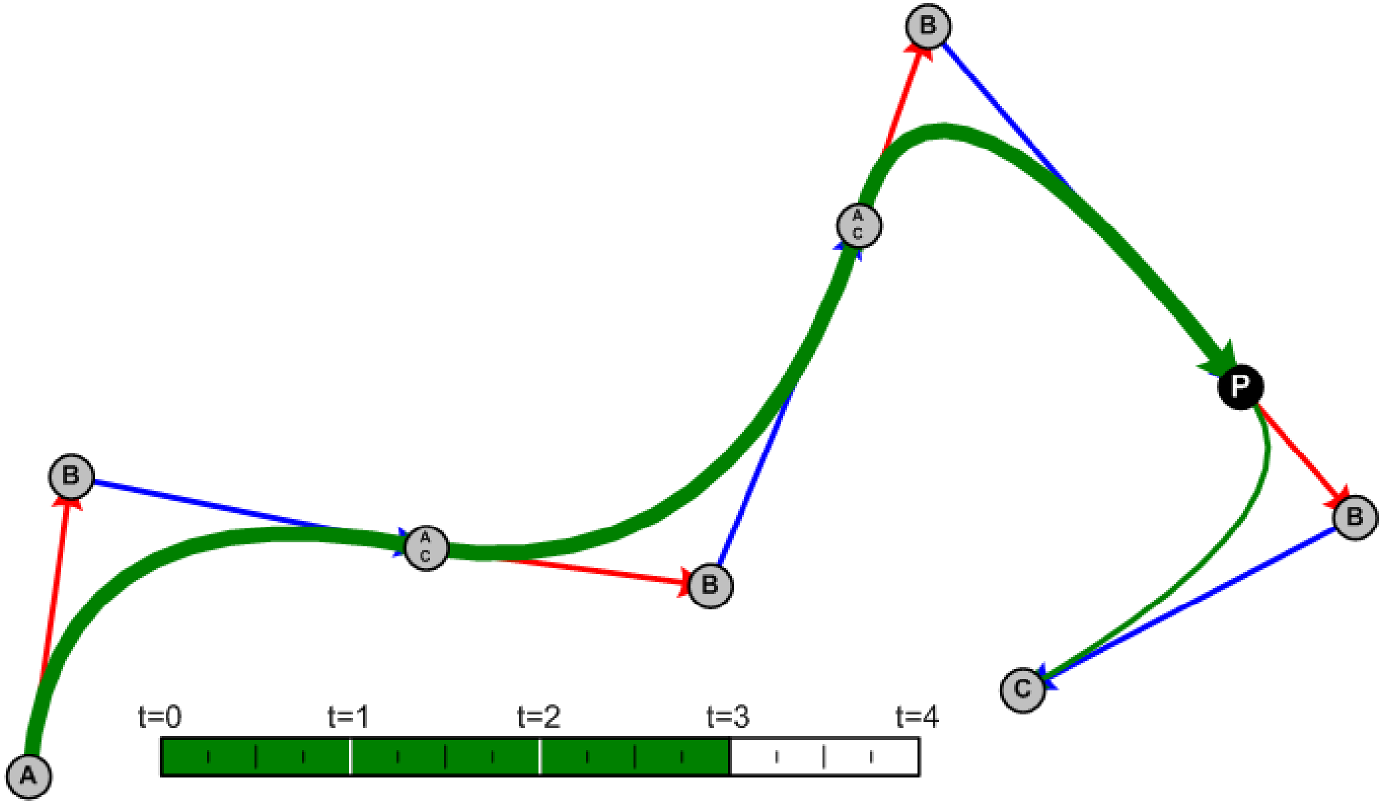
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



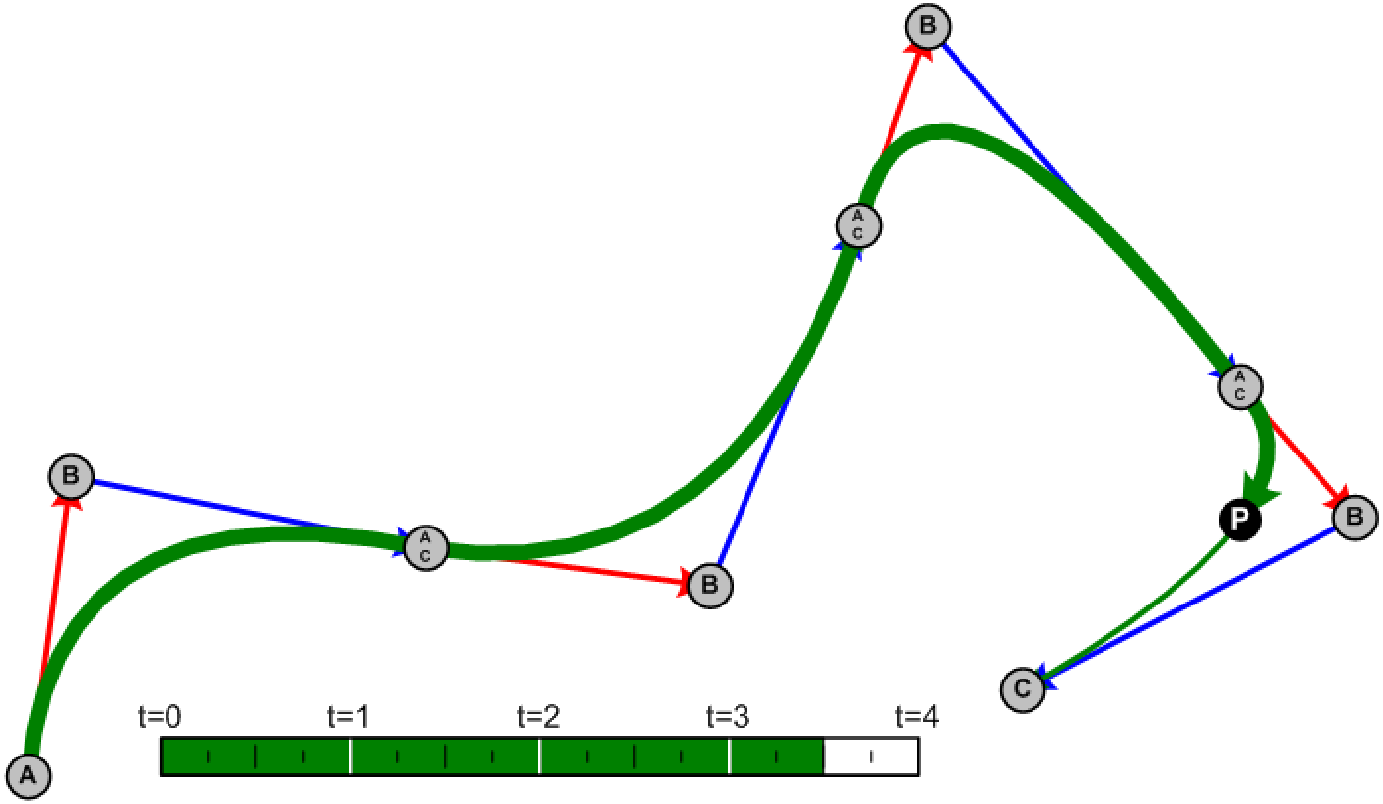
Splines

- Interpolating **spline_t** from 0.0 to 4.0...



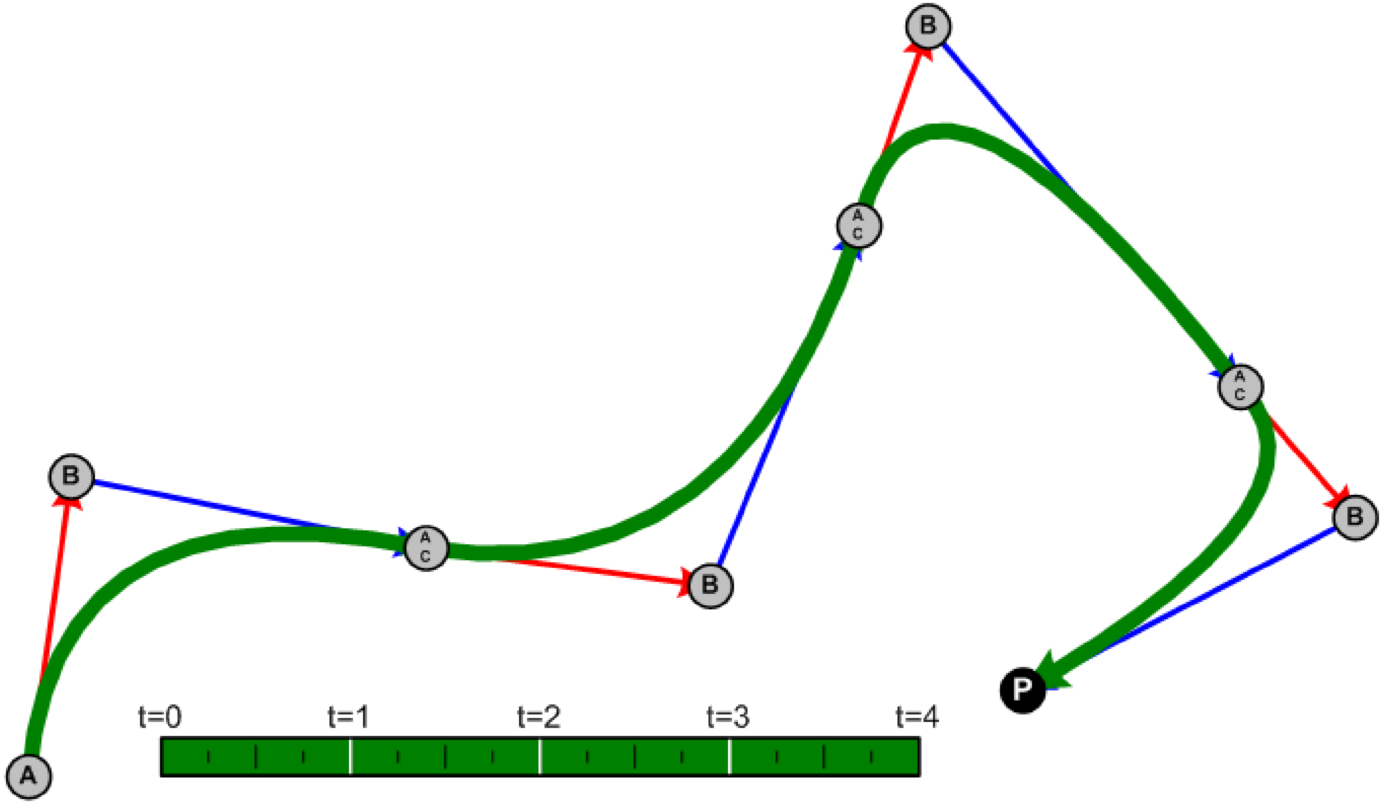
Splines

» Interpolating **spline_t** from 0.0 to 4.0...



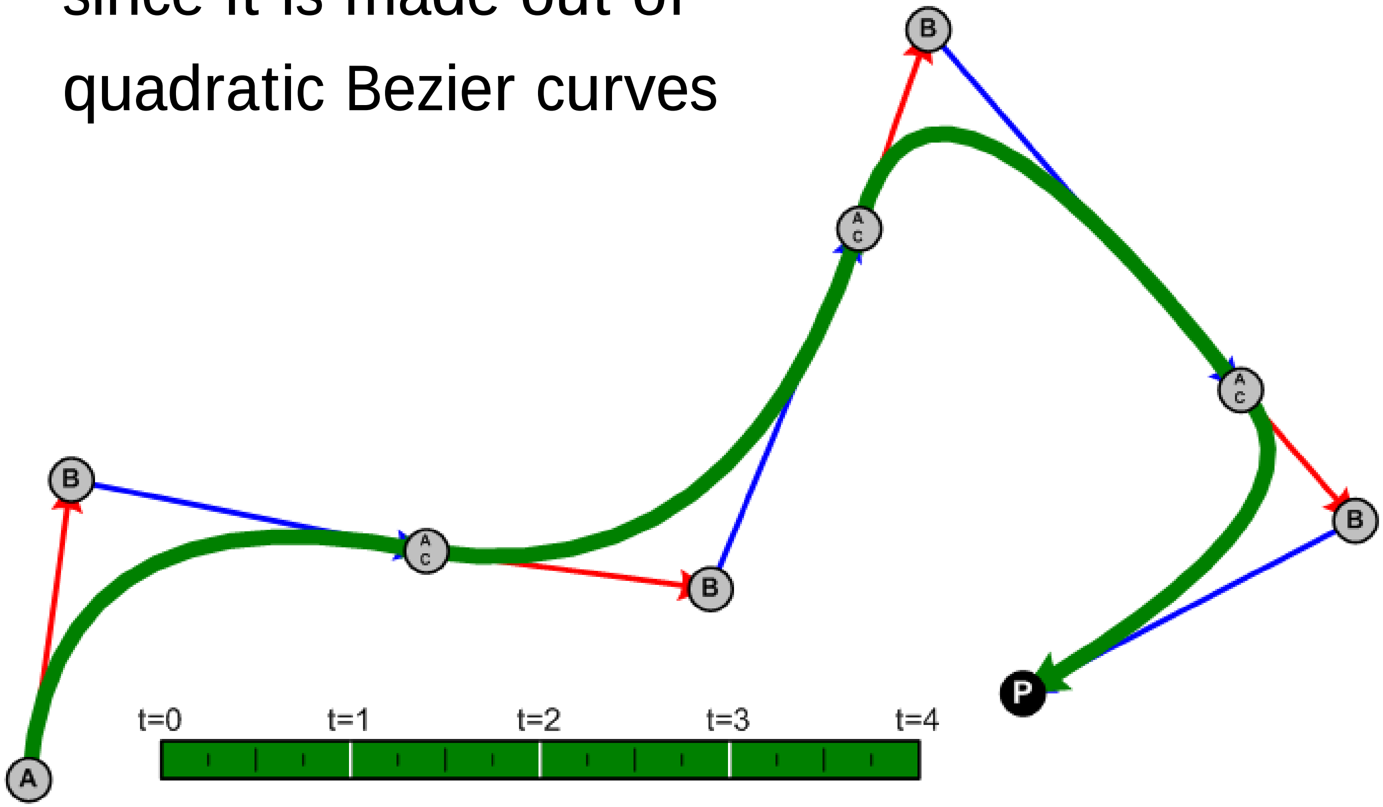
Splines

- Interpolating **spline_t** from 0.0 to 4.0...

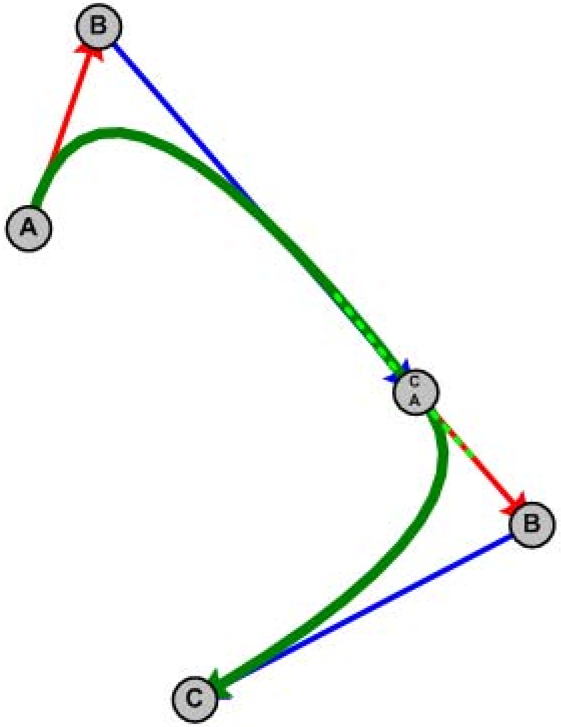


Quadratic Bezier Splines

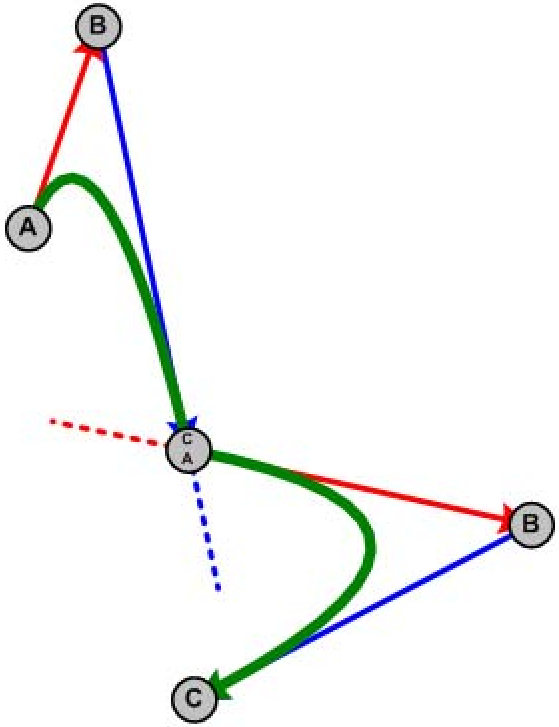
- This spline is a **quadratic Bezier spline**, since it is made out of quadratic Bezier curves



Continuity

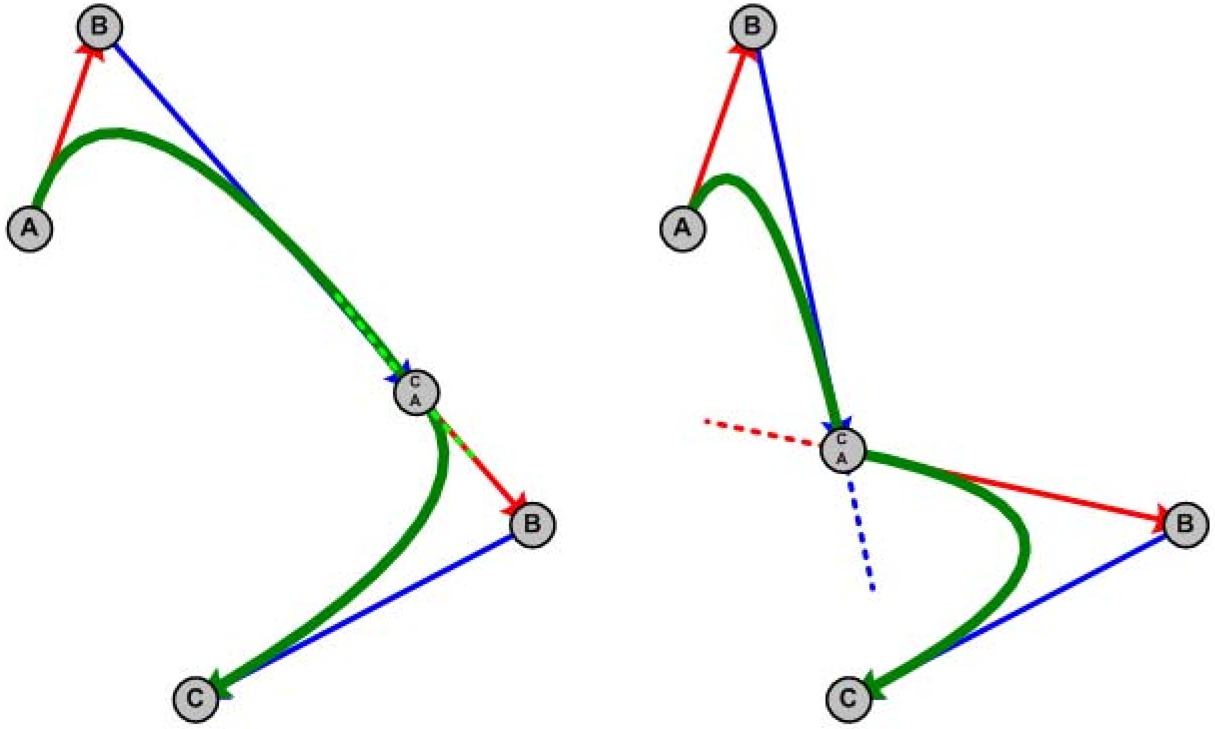


- Good continuity (C^1); connected **and** aligned



- Poor continuity (C^0); connected but not aligned

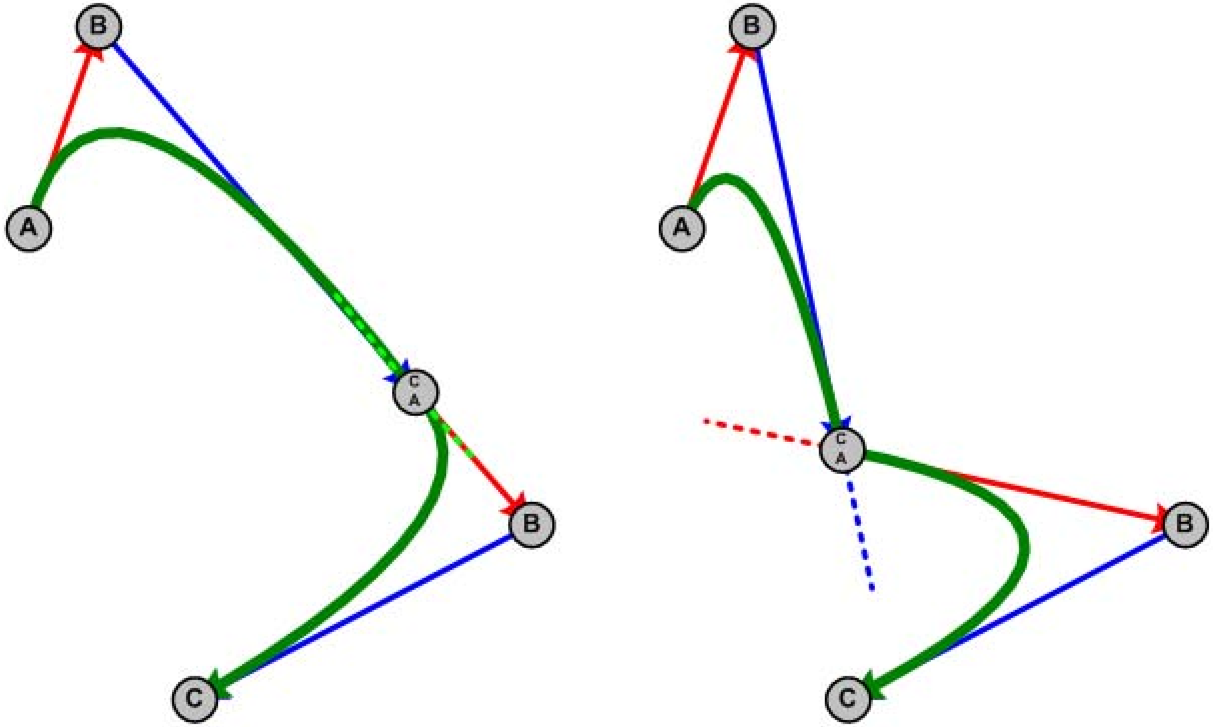
Continuity



- To ensure good continuity (C^1), make BC of first curve colinear (in line with) AB of second curve.

(derivative is continuous across entire spline)

Continuity

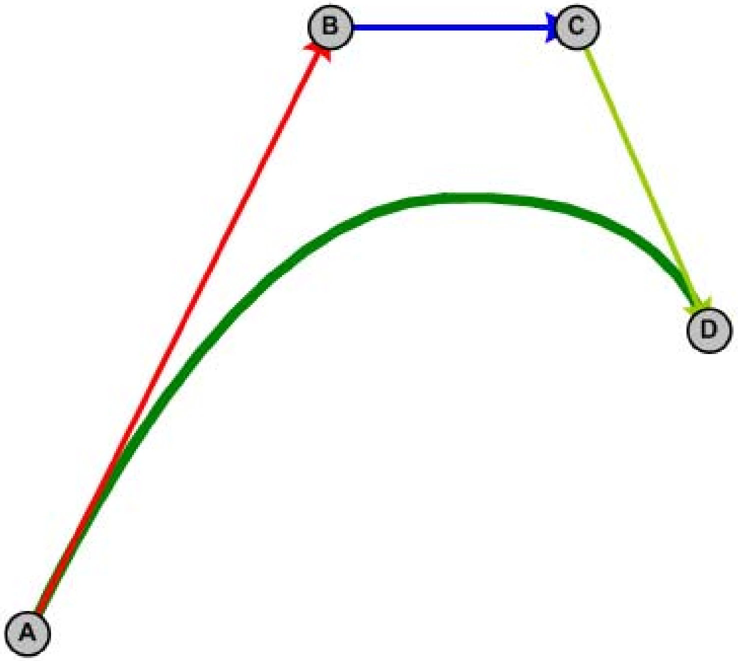


» Excellent continuity (C^2) is when speed/density matches on either side of each knot.

(second derivative is continuous across entire spline)

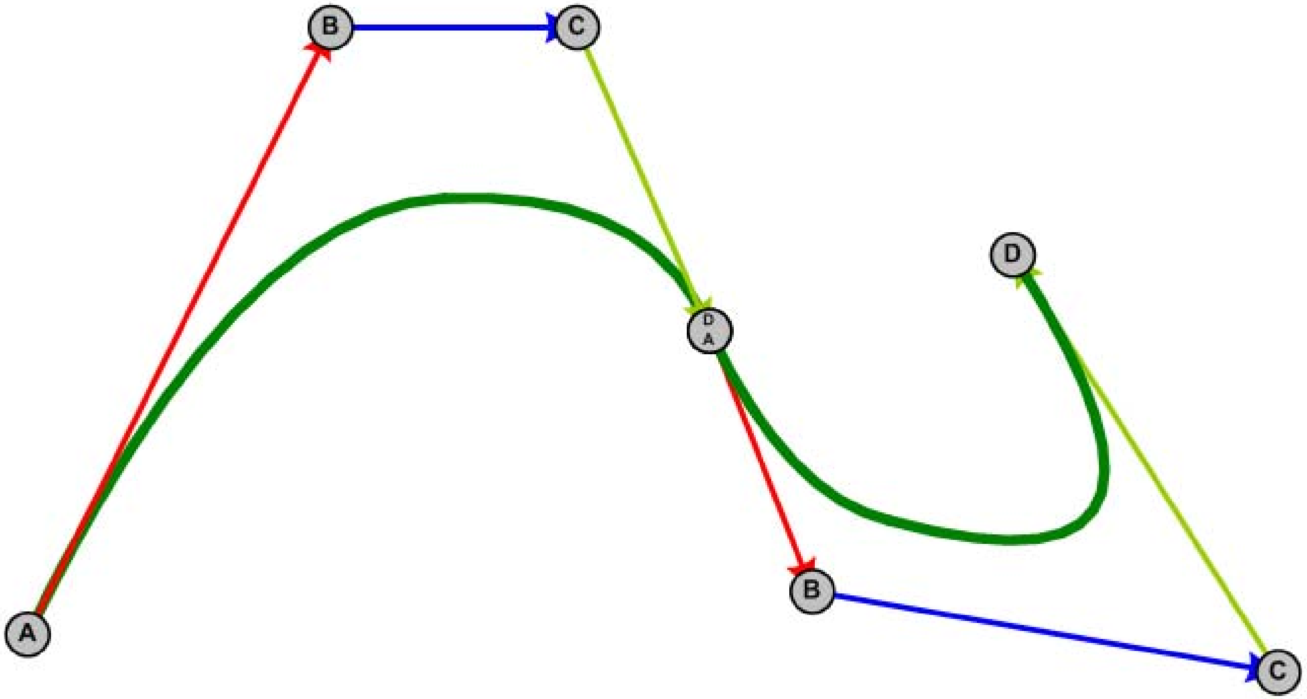
Cubic Bezier Splines

- We can build a **cubic Bezier spline** instead by using cubic Bezier curves.



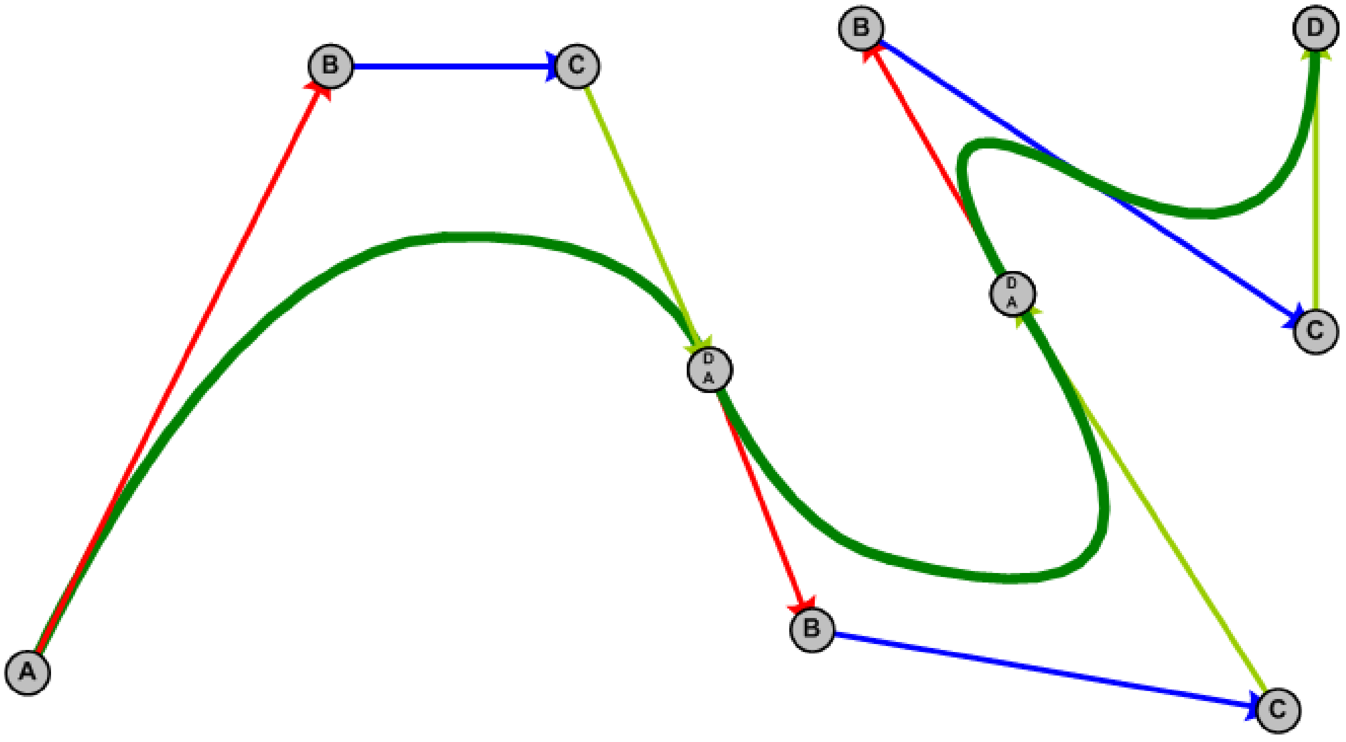
Cubic Bezier Splines

- We can build a **cubic Bezier spline** instead by using cubic Bezier curves.



Cubic Bezier Splines

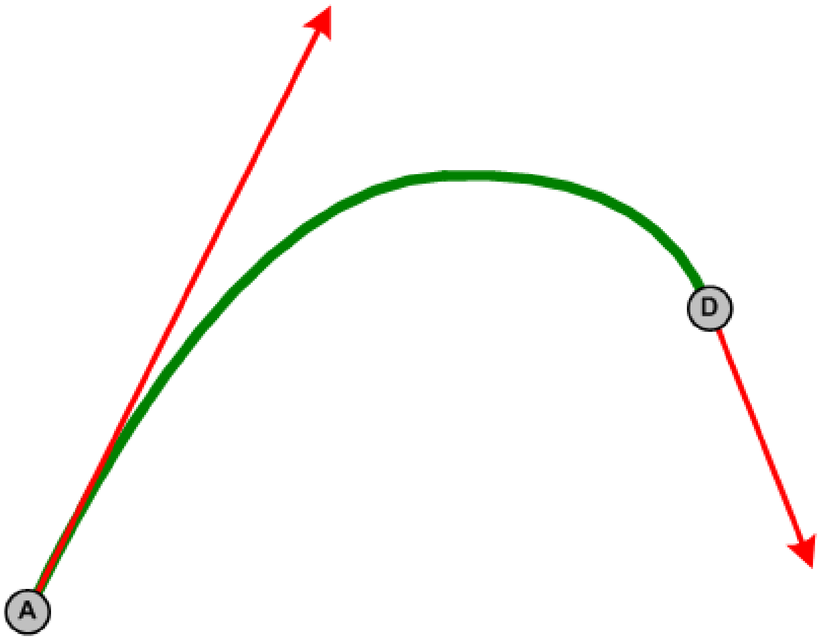
- We can build a **cubic Bezier spline** instead by using cubic Bezier curves.



Cubic Hermite Splines

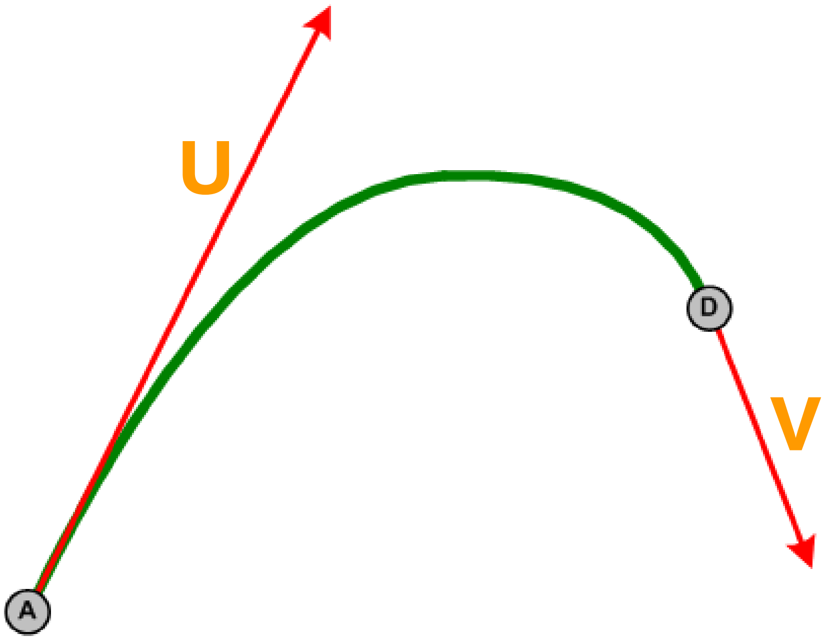
Cubic Hermite Splines

- A cubic Hermite spline is very similar to a cubic Bezier spline.



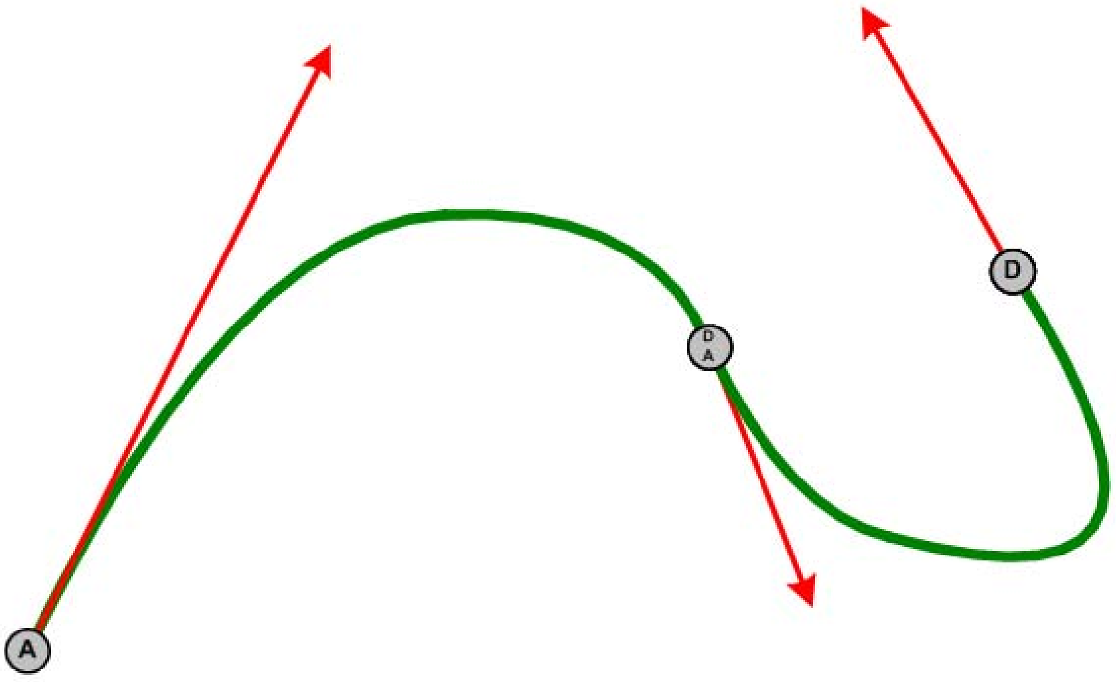
Cubic Hermite Splines

- However, we do not specify the **B** and **C** guide points.
- Instead, we give the velocity at point **A** (as **U**), and the velocity at **D** (as **V**) for each **cubic Hermite curve**.



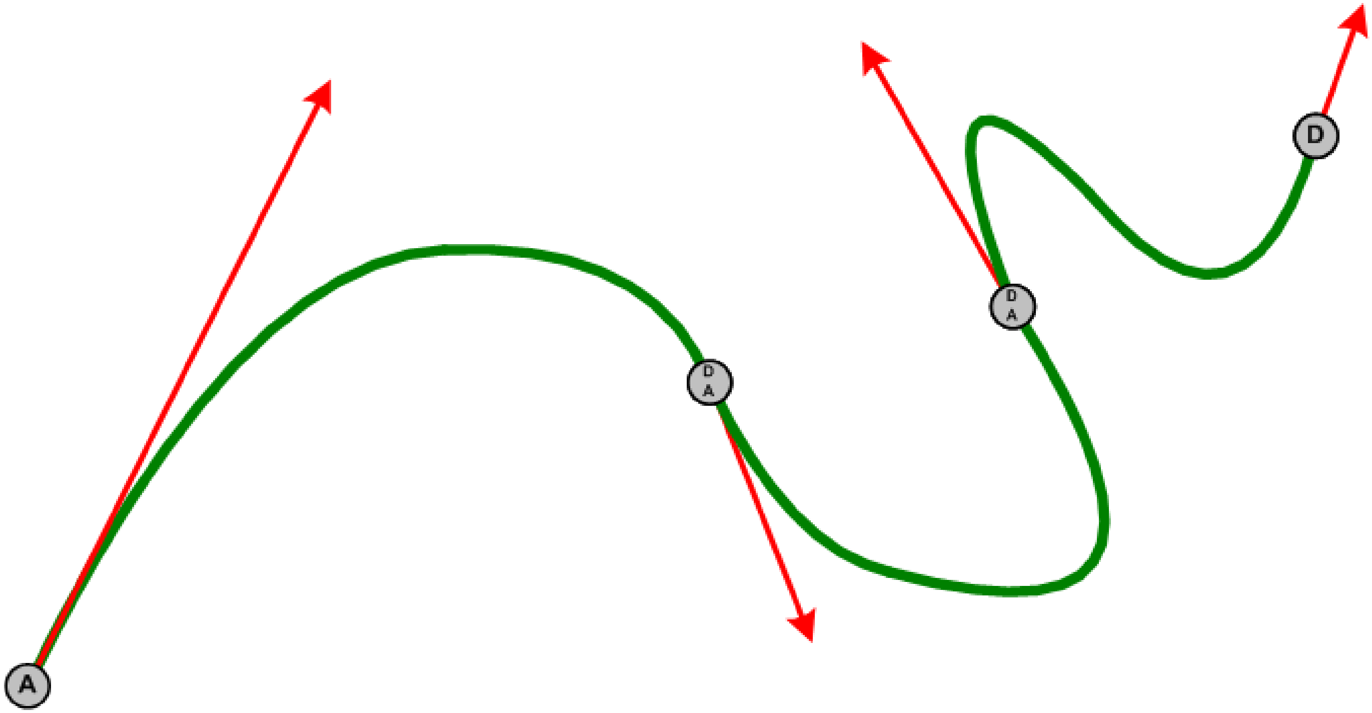
Cubic Hermite Splines

- To ensure connectedness (C^0), **D** from curve #0 is again welded on top of **A** from curve #1 (at a knot).



Cubic Hermite Splines

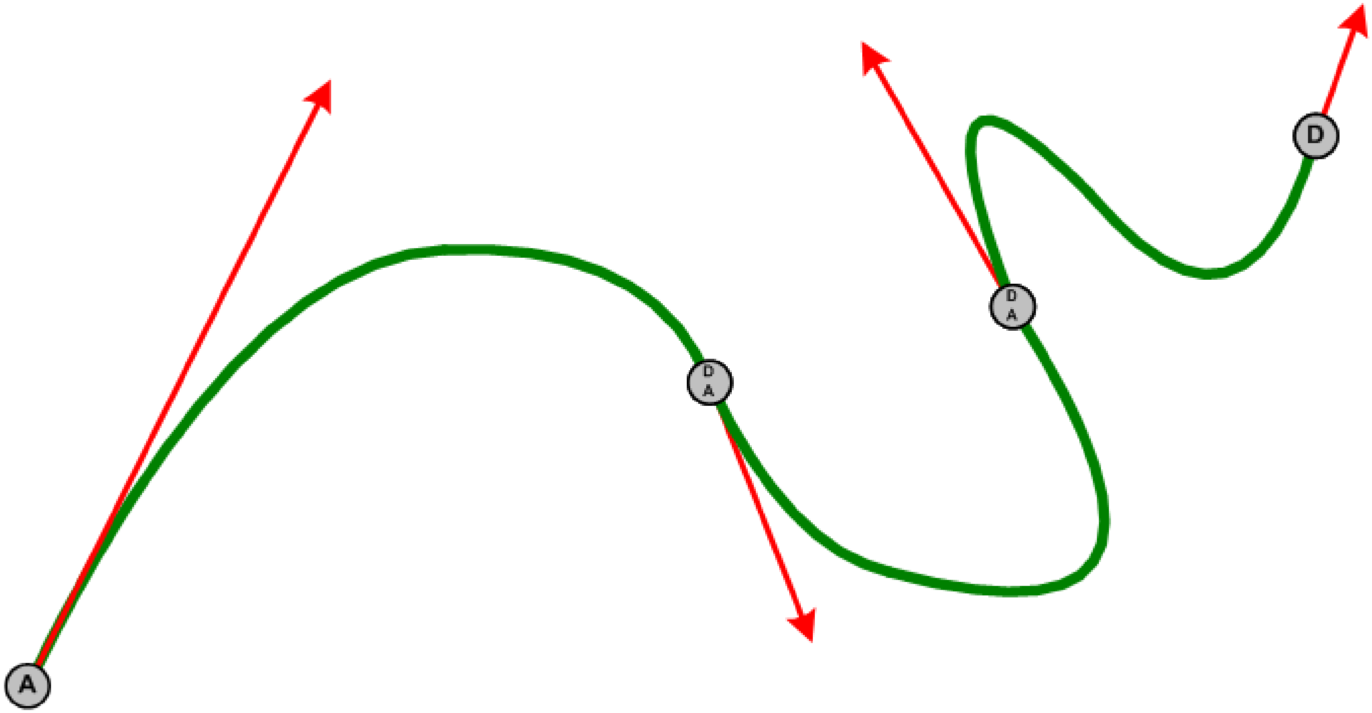
- To ensure smoothness (C^1), velocity into **D** (**V**) must match velocity's direction out of the next curve's **A** (**U**).



Cubic Hermite Splines

- For best continuity (C^2), velocity into **D** (**V**) must match direction **and magnitude** for the next curve's **A** (**U**).

(Hermite splines usually do match velocity magnitudes)



Cubic Hermite Splines

- Hermite curves, and Hermite splines, are also parametric and work basically the same way as Bezier curves: plug in “t” and go!

- The formula for **cubic Hermite curve** is:

$$\mathbf{P(t)} = s^2(1+2t)\mathbf{A} + t^2(1+2s)\mathbf{D} + s^2t\mathbf{U} + st^2\mathbf{V}$$

Cubic Hermite Splines

- Cubic Hermite and Bezier curves can be converted back and forth.
- To convert from cubic Hermite to Bezier:

$B = A + (U/3)$

$C = D - (V/3)$

- To convert from cubic Bezier to Hermite:

$U = 3(B - A)$

$V = 3(D - C)$

Catmull-Rom Splines

Catmull-Rom Splines

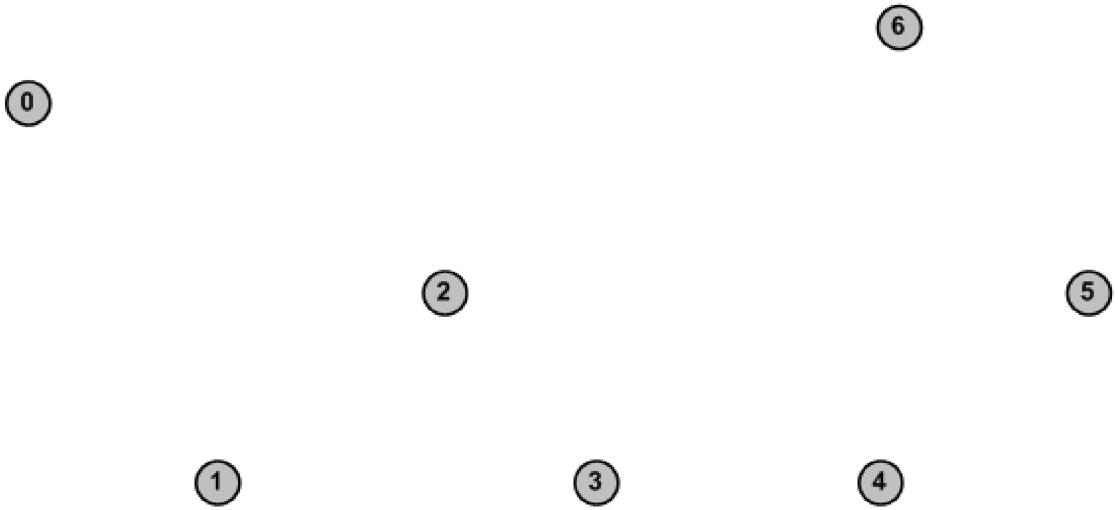
- A **Catmull-Rom spline** is just a cubic Hermite spline with special values chosen for the velocities at the start (**U**) and end (**V**) points of each section.

- You can also think of Catmull-Rom not as a type of spline, but as a technique for building cubic Hermite splines.

- Best application: curve-pathing through points

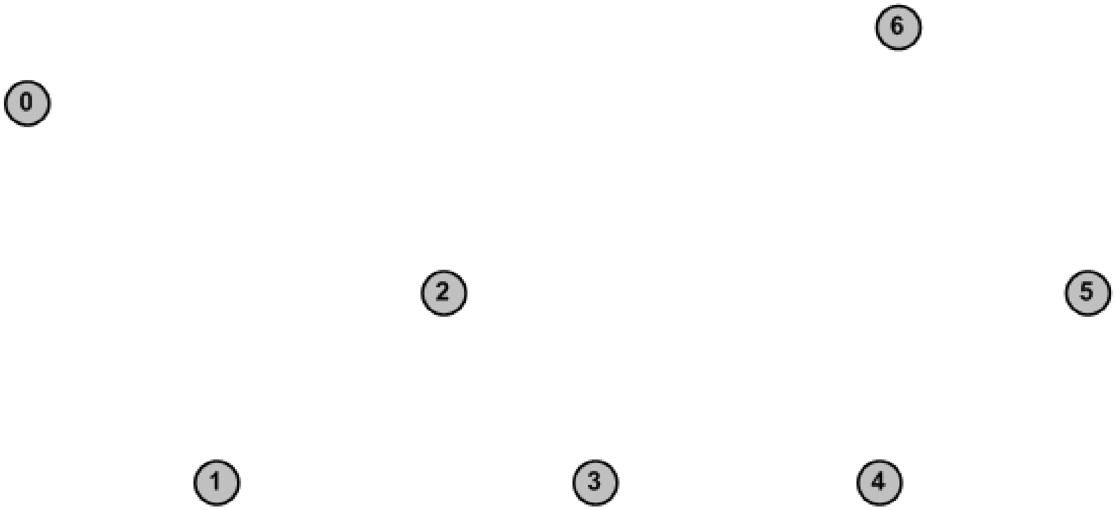
Catmull-Rom Splines

» Start with a series of points (spline start, spline end, and interior knots)



Catmull-Rom Splines

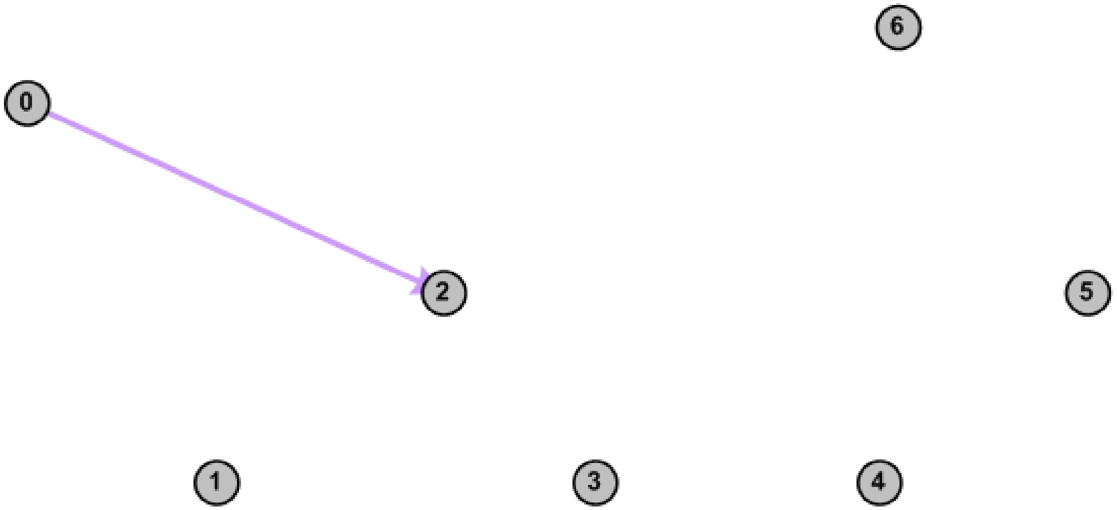
» 1. Assume **U** and **V** velocities are zero at start and end of spline (points 0 and 6 here).



Catmull-Rom Splines

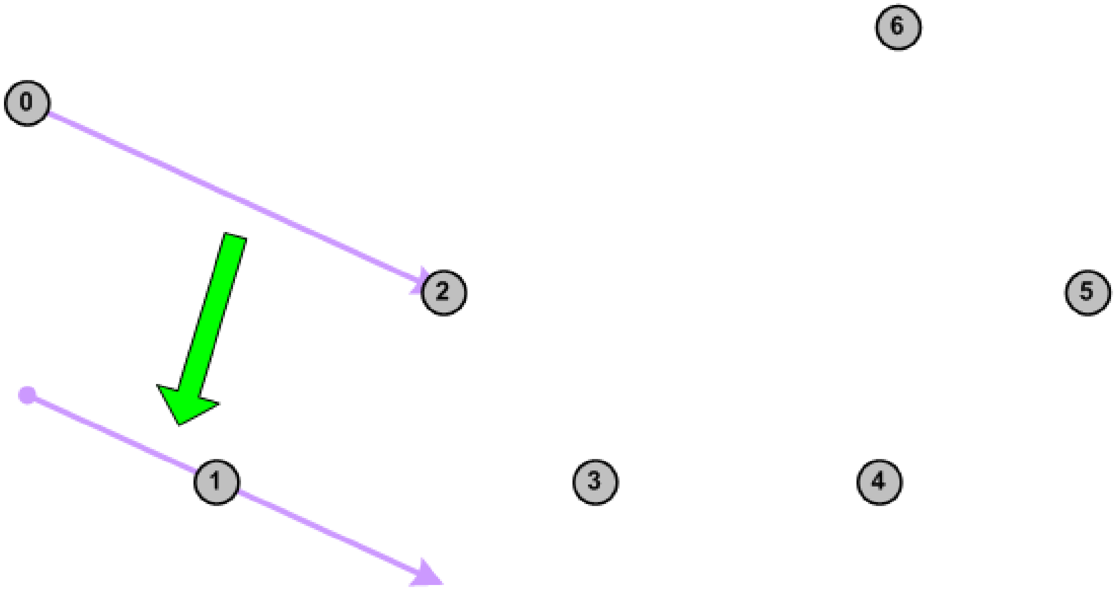
- 2. Compute a vector from point 0 to point 2.

  $(Vec_{0\_to\_2} = P_2 - P_0)$



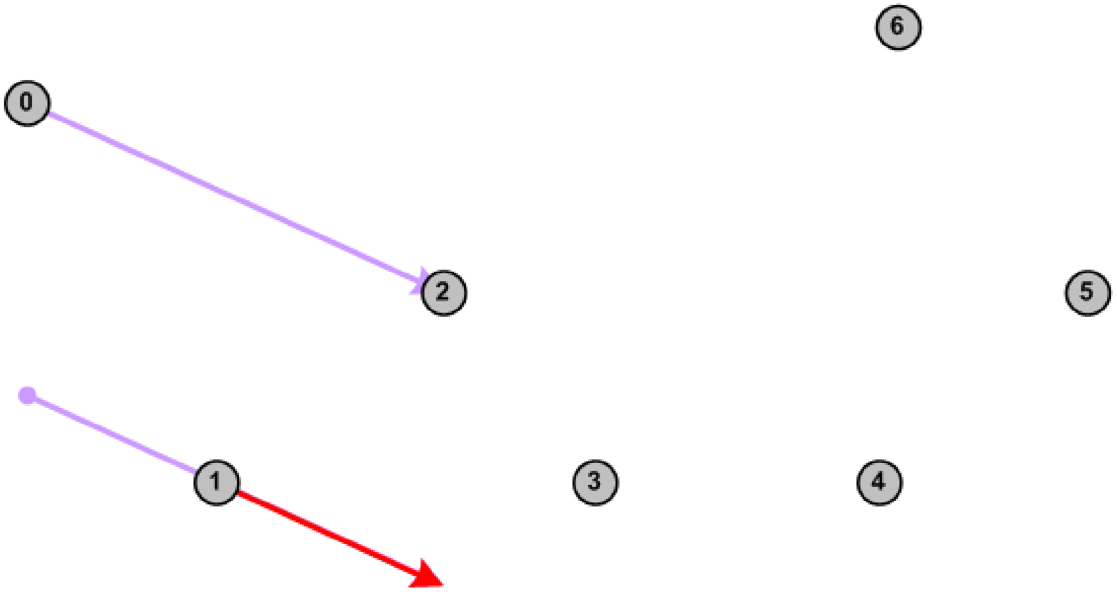
Catmull-Rom Splines

- That will be our tangent for point 1.



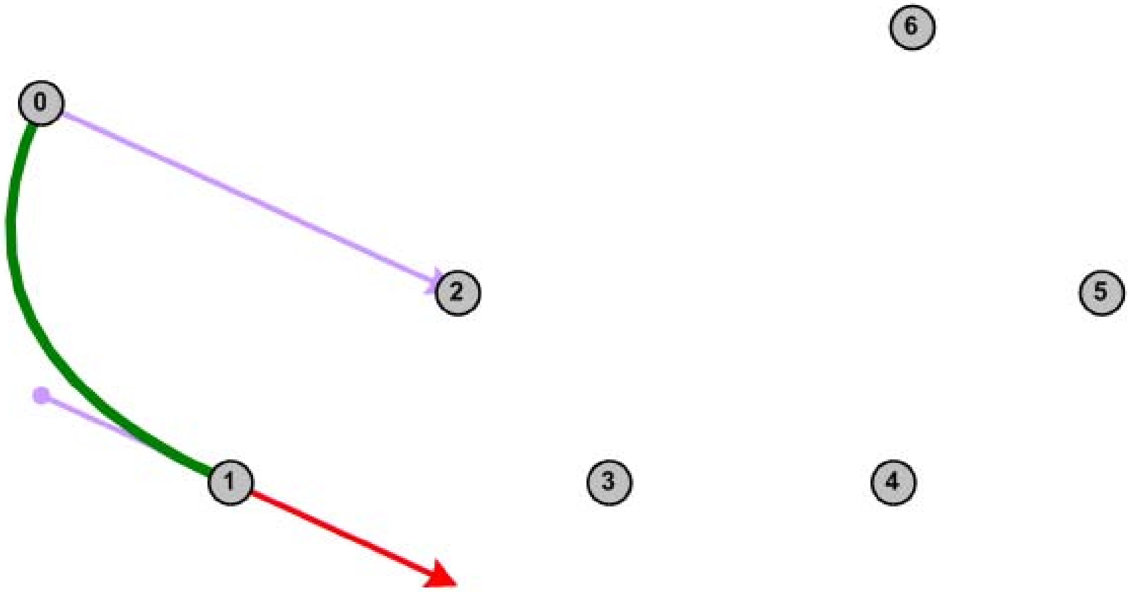
Catmull-Rom Splines

- 3. Set the velocity for point 1 to be ½ of that.



Catmull-Rom Splines

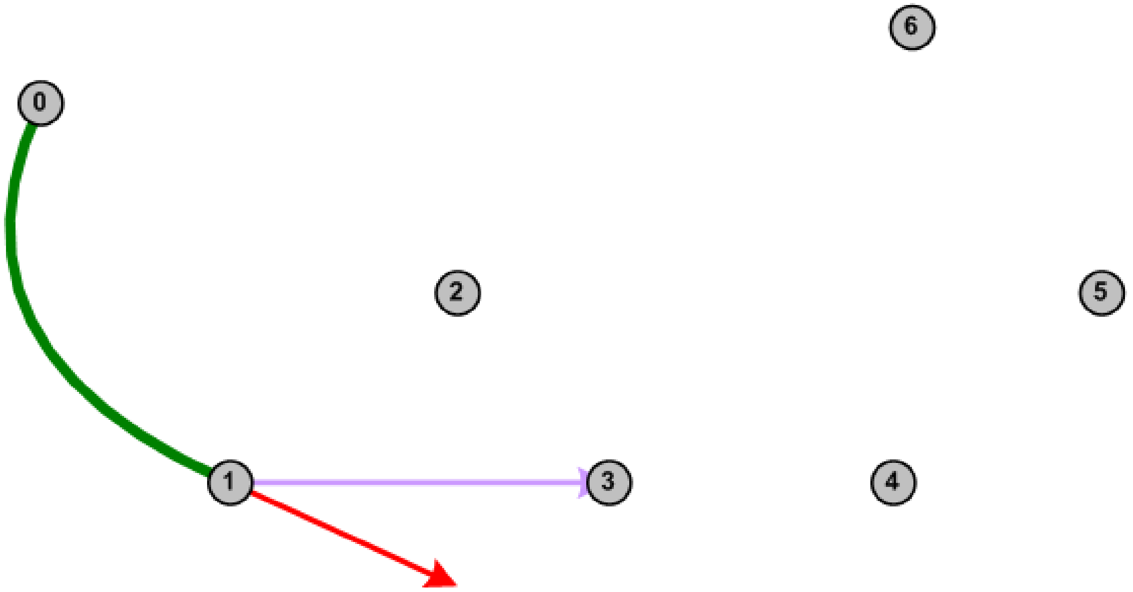
- Now we have set positions 0 and 1, and velocities at points 0 and 1. Hermite curve!



Catmull-Rom Splines

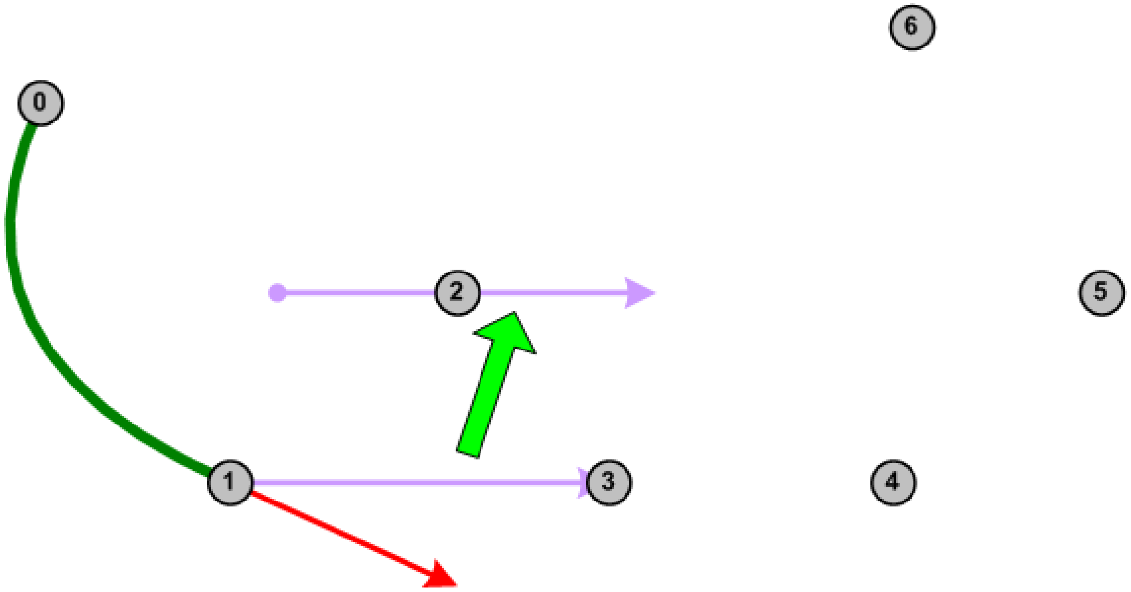
- 4. Compute a vector from point 1 to point 3.

  $(Vec_{1\_to\_3} = P_3 - P_1)$



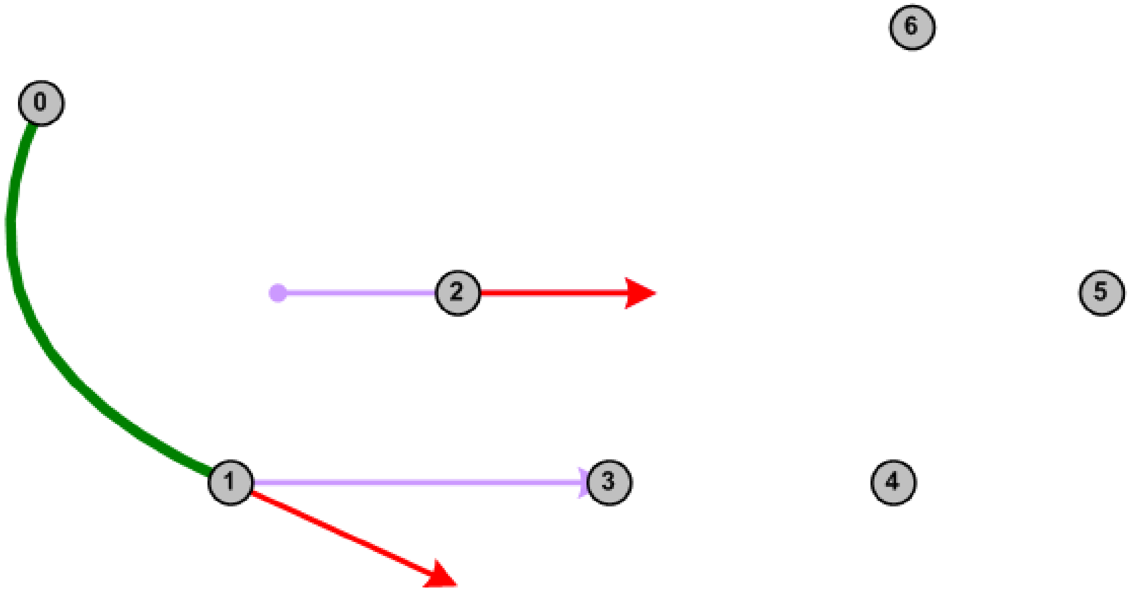
Catmull-Rom Splines

» That will be our tangent for point 2.



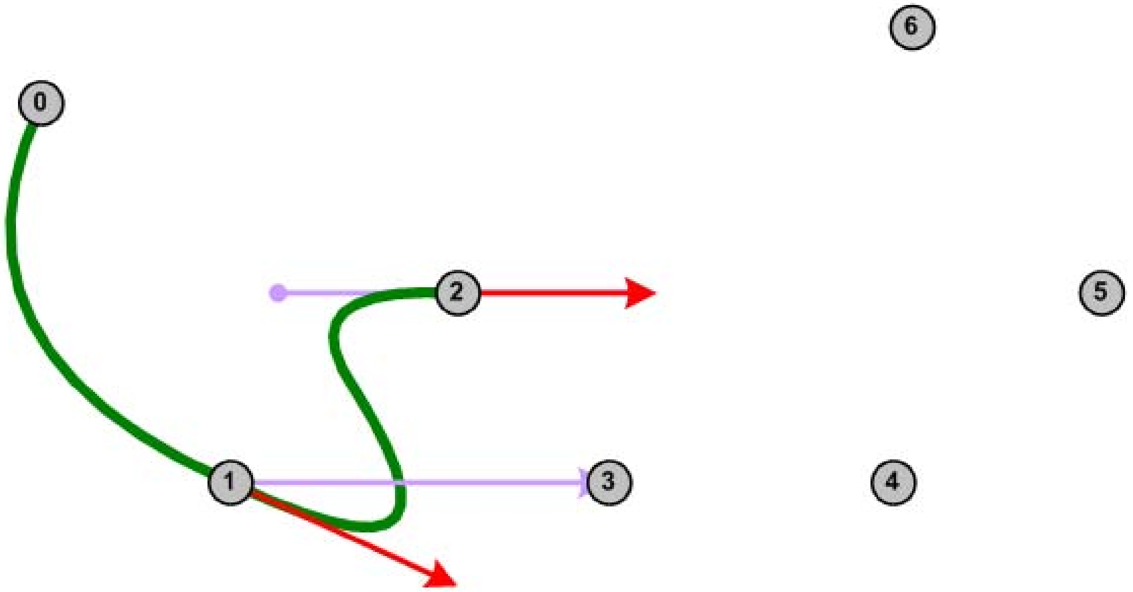
Catmull-Rom Splines

» 5. Set the velocity for point 2 to be ½ of that.



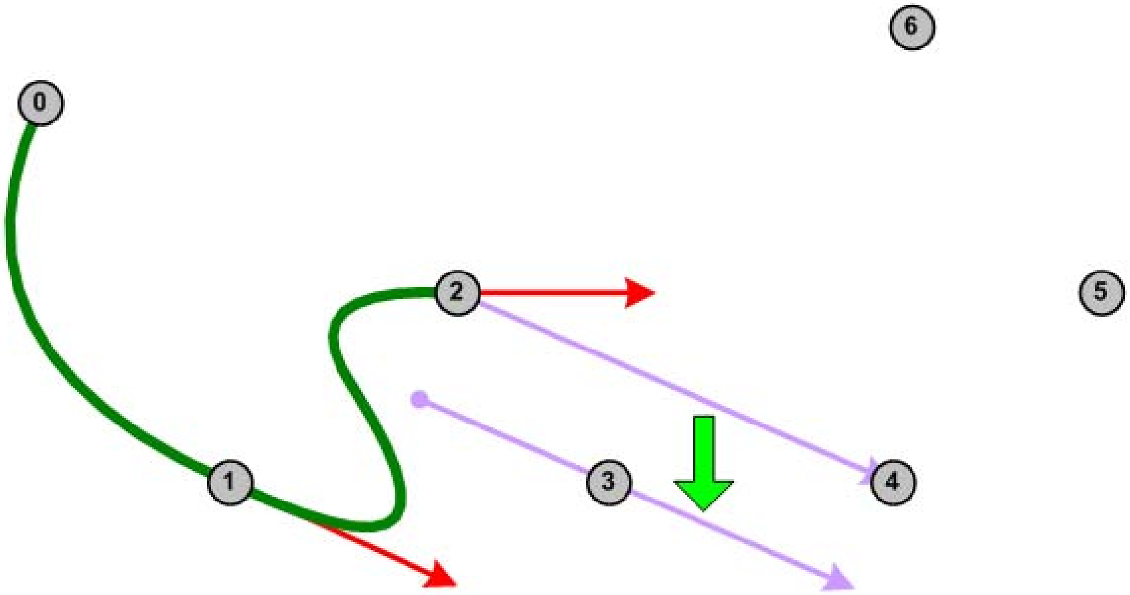
Catmull-Rom Splines

- Now we have set positions and velocities for points 0, 1, and 2. We have a Hermite spline!



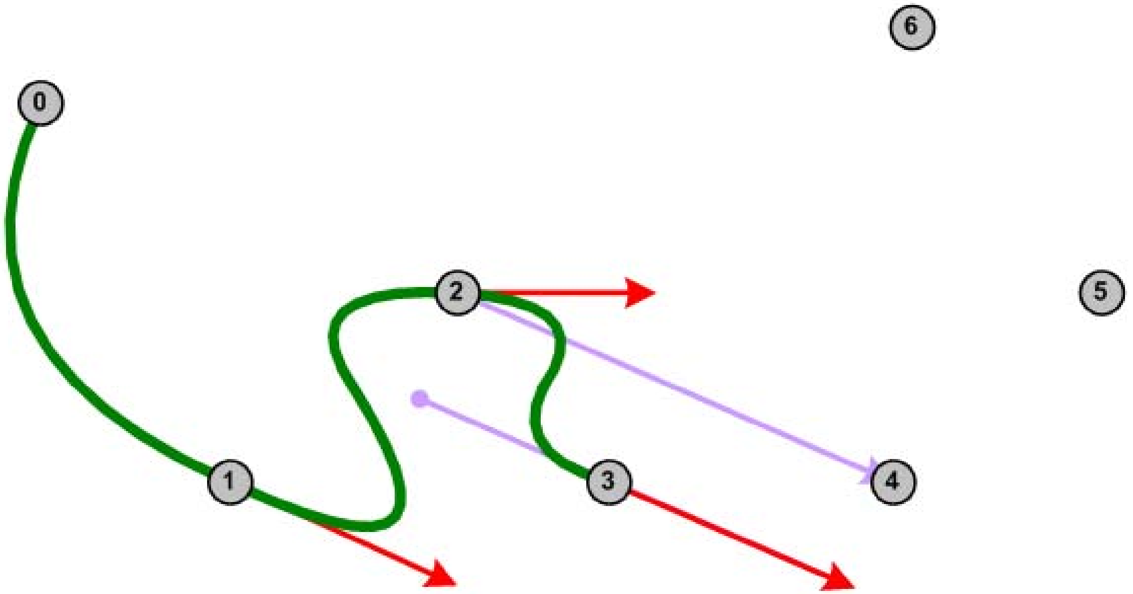
Catmull-Rom Splines

- Repeat the process to compute velocity at point 3.



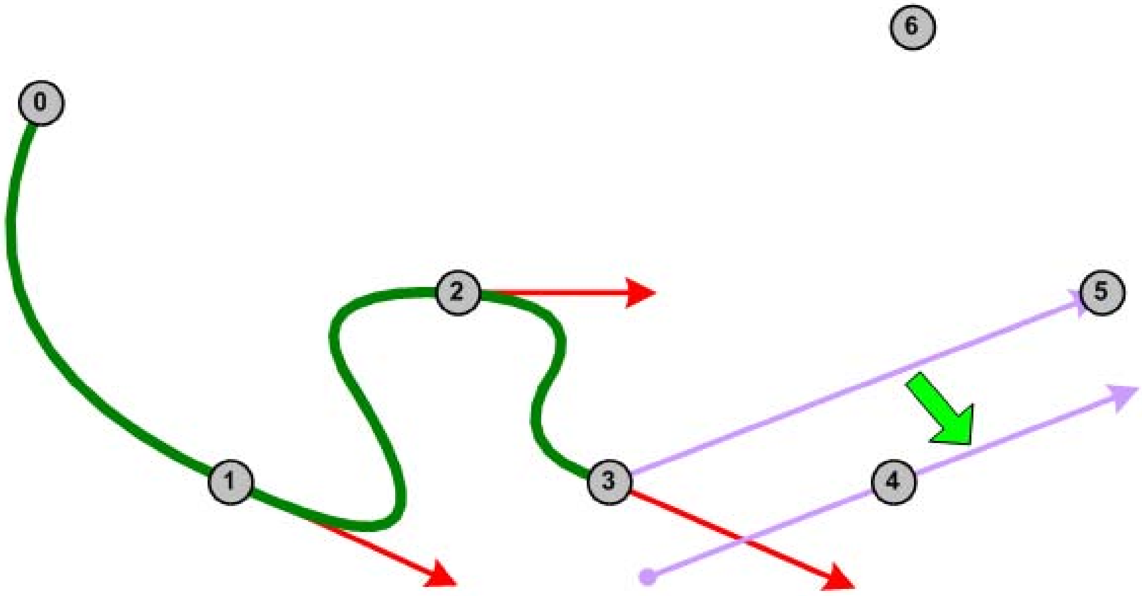
Catmull-Rom Splines

- Repeat the process to compute velocity at point 3.



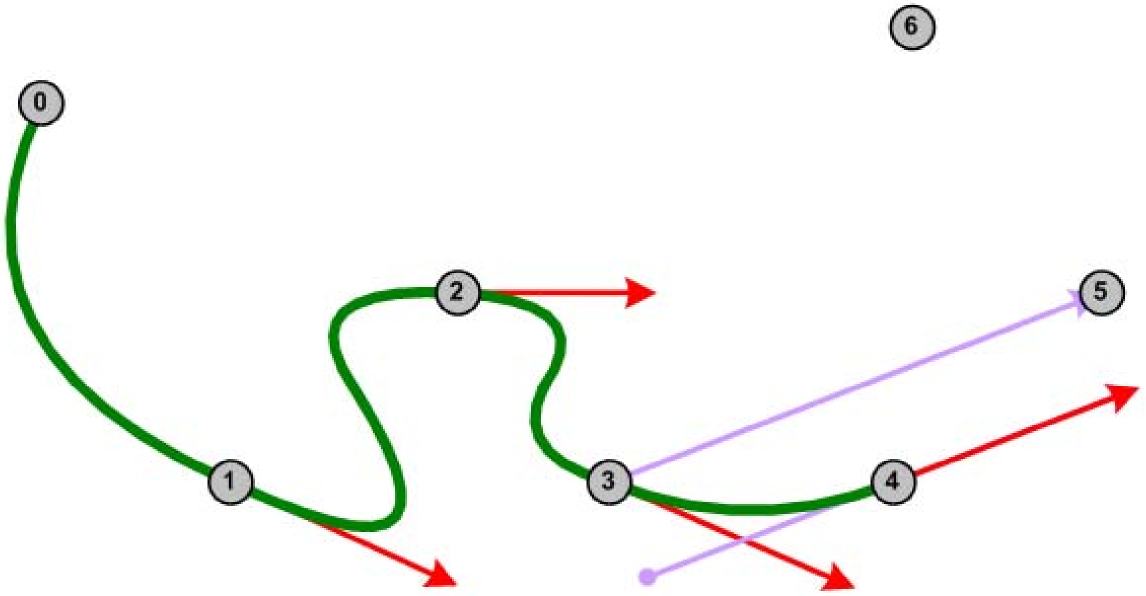
Catmull-Rom Splines

- And at point 4.



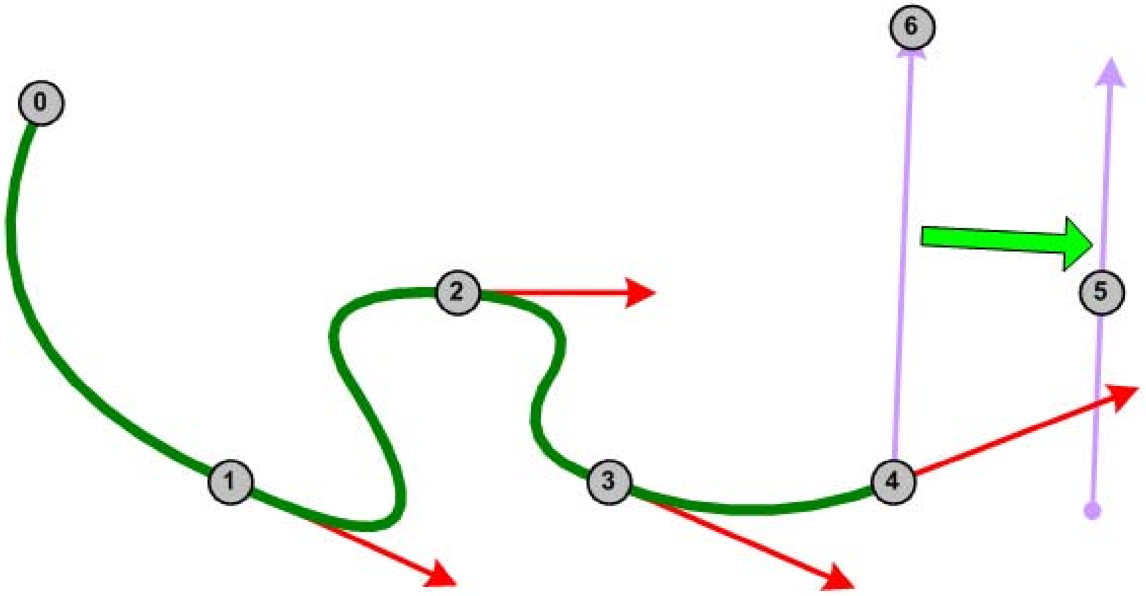
Catmull-Rom Splines

- And at point 4.



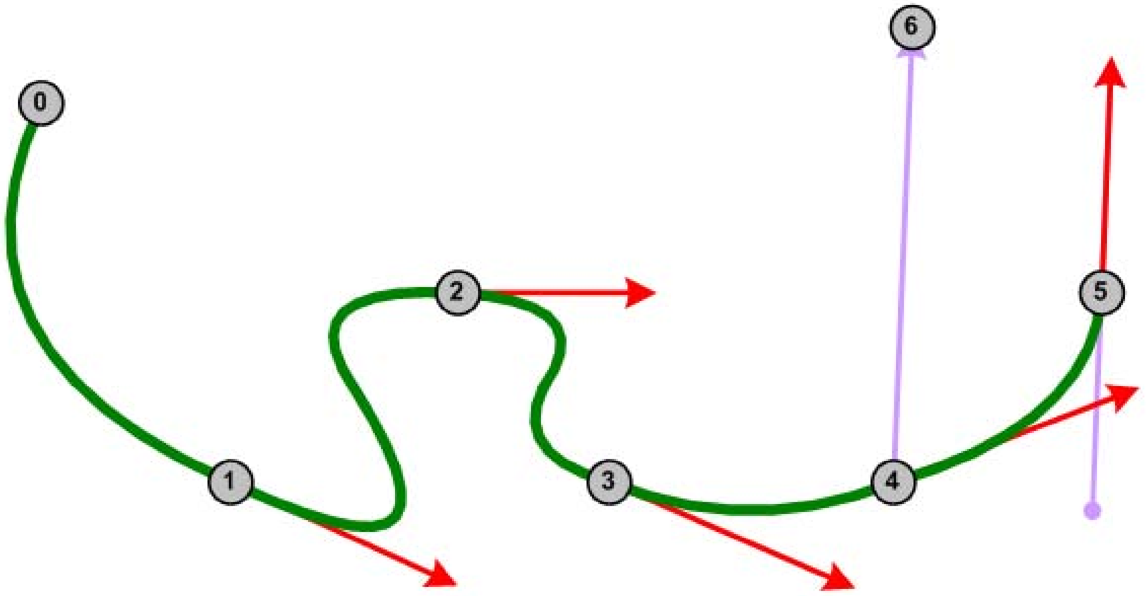
Catmull-Rom Splines

- Compute velocity for point 5.



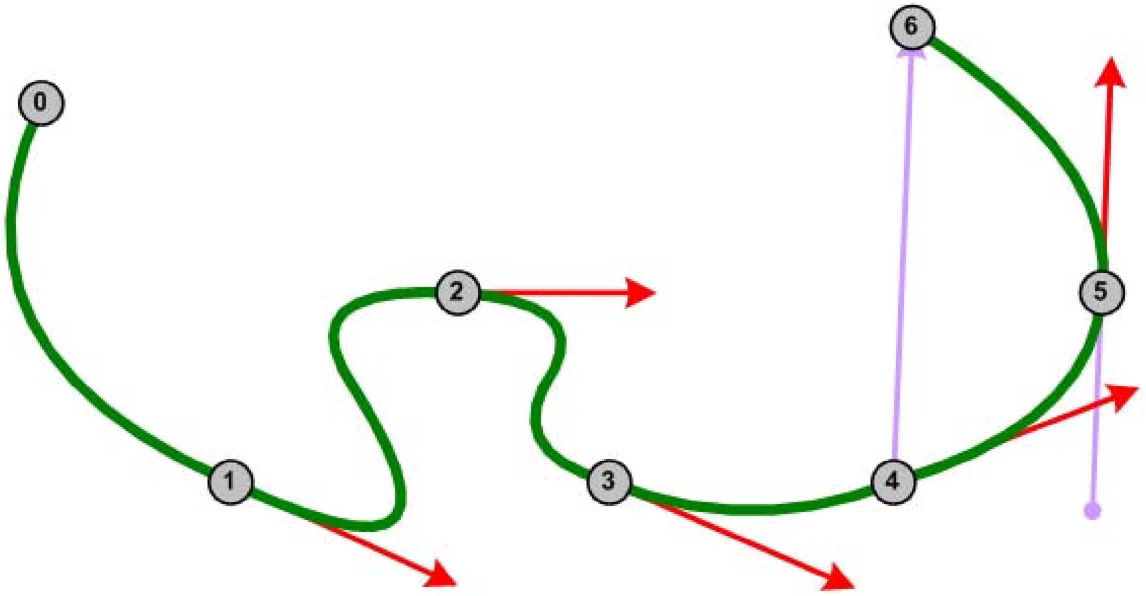
Catmull-Rom Splines

- Compute velocity for point 5.



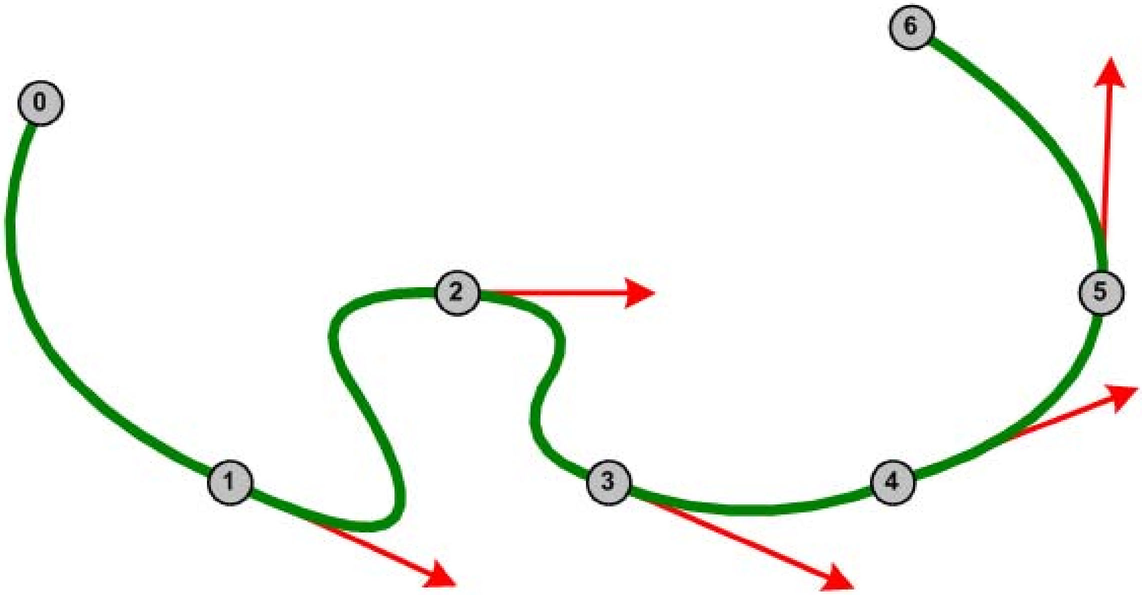
Catmull-Rom Splines

» We already set the velocity for point 6 to be zero, so we can close out the spline.



Catmull-Rom Splines

- And voila! A Catmull-Rom (Hermite) spline.



Catmull-Rom Splines

Here’s the math for a Catmull-Rom Spline:

- Place knots where you want them (**A**, **D**, etc.)
- Position at the Nth point is P_N
- Velocity at the Nth point is V_N
- $V_N = (P_{N+1} – P_{N-1}) / 2$

- i.e. Velocity at point P is half of [the vector pointing from the previous point to the next point].

Cardinal Splines

Cardinal Splines

- Same as a Catmull-Rom spline, but with an extra parameter: **Tension**.

- Tension can be set from 0 to 1.

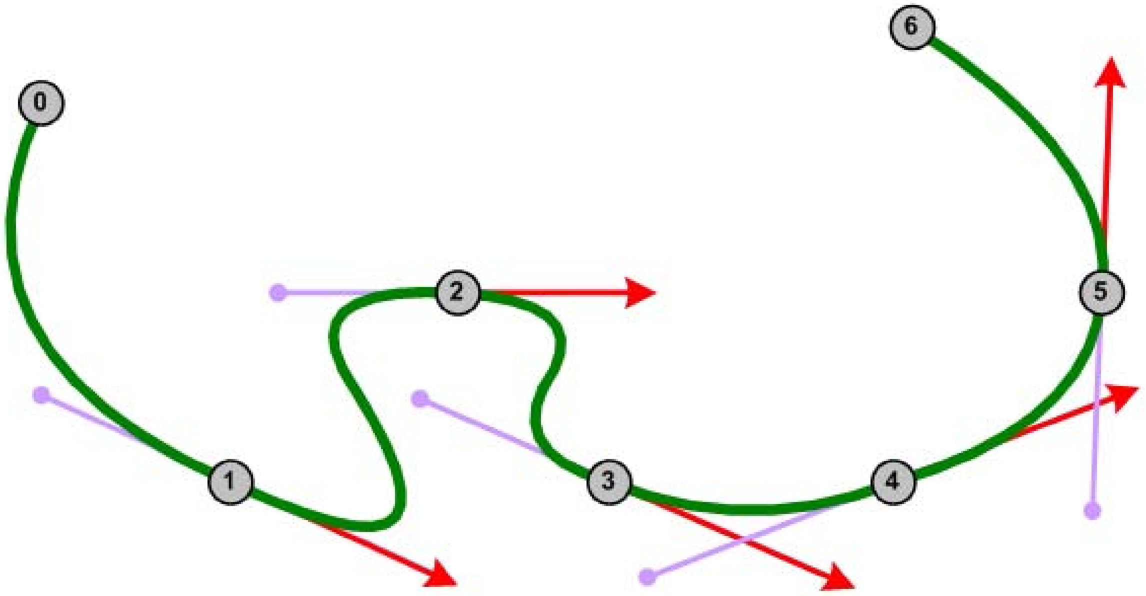
- A tension of 0 is just a Catmull-Rom spline.

- Increasing tension causes the velocities at all points in the spline to be scaled down.

Cardinal Splines

» So here is a Cardinal spline with tension=0

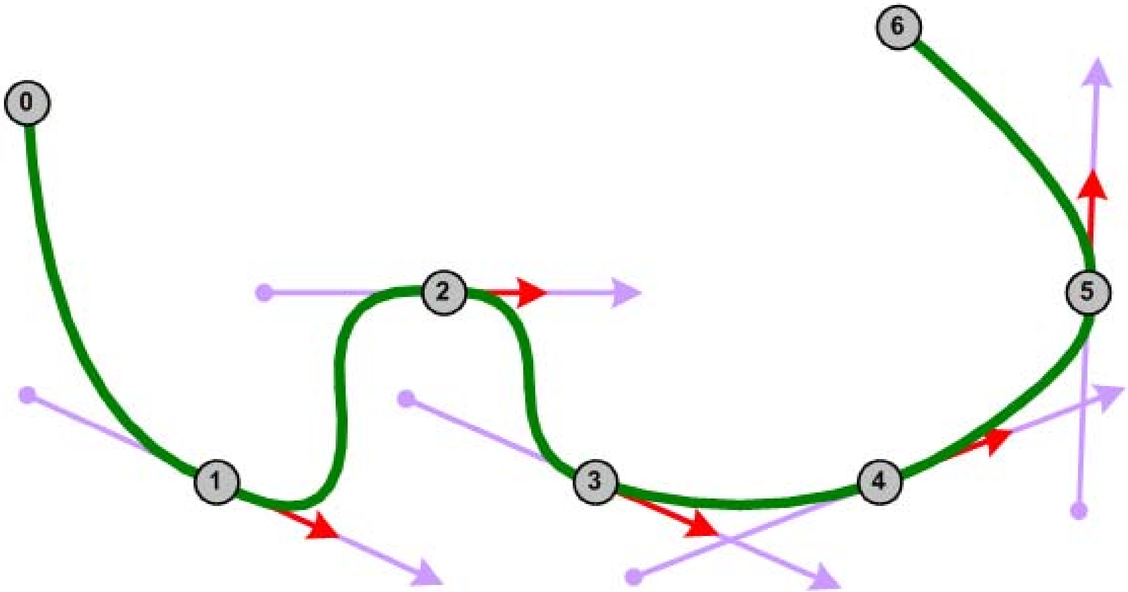
(same as a Catmull-Rom spline)



Cardinal Splines

- So here is a Cardinal spline with tension=.5

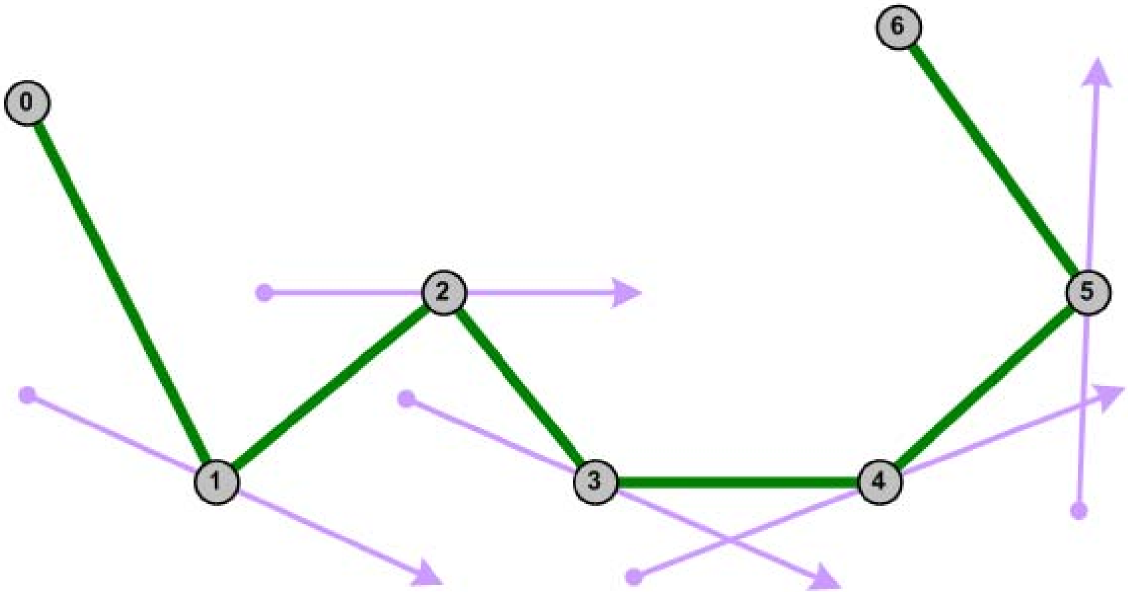
  (velocities at points are ½ of the Catmull-Rom)



Cardinal Splines

- And here is a Cardinal spline with tension=1

  (velocities at all points are zero)



Cardinal Splines

Here’s the math for a Cardinal Spline:

- Place knots where you want them (**A**, **D**, etc.)
- Position at the Nth point is P_N
- Velocity at the Nth point is V_N
- $V_N = (1 - \text{tension})(P_{N+1} - P_{N-1}) / 2$

- i.e. Velocity at point P is **some fraction of** half of [the vector pointing from the previous point to the next point].
- i.e. Same as Catmull-Rom, but V_N gets scaled down because of the (1 – tension) multiply.

Other Spline Types

Kochanek–Bartels Splines

- Same as a Cardinal spline (includes **Tension**), but with two extra tweaks (usually set on the entire spline):

Bias (from -1 to +1):

- A zero bias leaves the velocity vector alone
- A positive bias rotates the velocity vector to be more aligned with the point BEFORE this point
- A negative bias rotates the velocity vector to be more aligned with the point AFTER this point

Continuity (from -1 to +1):

- A zero continuity leaves the velocity vector alone
- A positive continuity “poofs out” the corners
- A negative continuity “sucks in / squares off” corners

B-Splines

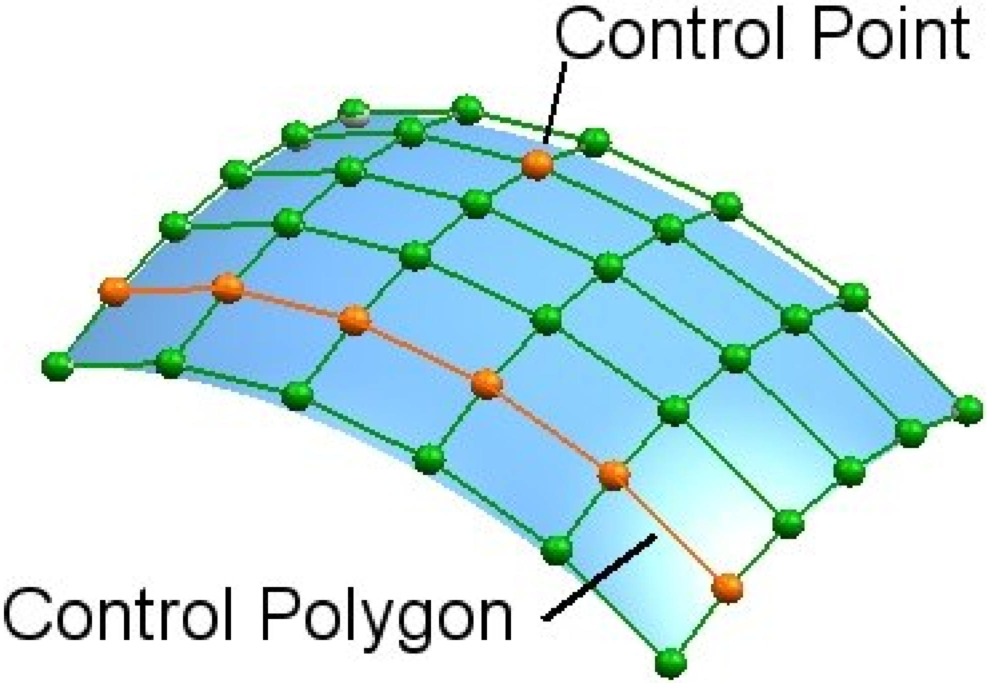
- Stands for “basis spline”.
- Just a generalization of Bezier splines.
- The basic idea:

  At any given time, P(t) is a weighted-average blend of 2, 3, 4, or more points in its neighborhood.

- Equations are usually given in terms of the blend weights for each of the nearby points based on where **t** is at.

Curved Surfaces

» Way beyond the scope of this talk, but basically you can criss-cross splines and form 2d curved surfaces.



Thanks!

Feel free to contact me:

Squirrel Eiserloh

Director
TrueThought LLC

Squirrel@Eiserloh.net
Squirrel@TrueThought.com