Interpolation

Computer Graphics

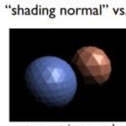
Overview

- Interpolation (Why we need interpolation)
- Averaging & Blending
- Linear (1D and 2D)
- Angular (Small & Large Angles)

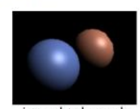
Why do we need interpolation?

- Where do we see interpolation in computer graphics?
- Is interpolation only for animation?

- Texture coordinates
  - without interpolating there can't really be textures
- Surface normals
  - for smooth surfaces approximated with meshes
  - use interpolated normal for shading in place of actual normal
  - "shading normal" vs. "geometric normal"

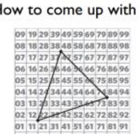


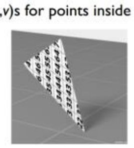
geometric normals



interpolated normals

- Texture coordinates are per-vertex data like vertex positions
  - can think of them as a second position: each vertex has a position in 3D space and in 2D texture space
- How to come up with (u,v)s for points inside triangles?

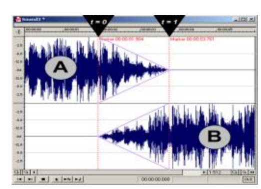




Mixing (Interpolation)

We use "lerping" all the time, under different names.

For example: an Audio crossfade



Blending Concepts

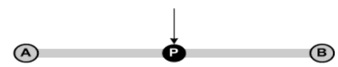
- Refresh

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

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".

We can use any blend weights we want, as long as they add up to 1.0 (100%).

Averaging and Blending

So if we generalized it, we would 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**

Blending Compound Data (e.g., Vectors)

We can blend more than just simple numbers!

Blending 2D or 3D vectors, for example, is a cinch:

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

Just blend each component (x,y,z) separately, at the same time.

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

Blending Compound Data

Need to be careful, though!

Not all compound data types will blend correctly with this approach.

Examples: Color RGBs, Euler angles (yaw/pitch/roll), Matrices, Quaternions...

- ...in fact, there are a bunch that won't.

Blending Compound Data (e.g., Colour)

Here's an RGB colour example:

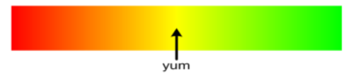
If A is **RGB(255, 0, 0)** – **bright red**
...and B is **RGB(0, 255, 0)** – **bright green**

Blending the two (with t = 0.5) gives:
RGB(127, 127, 0)

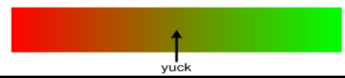
- ...which is a **dull, swampy color**. Yuck.

Blending Compound Data

What we **wanted** was this:



...and what we got instead was this:



Interpolation

Basically:

whenever we do any sort of **blend over time**

we're **lerping** (interpolating)

Interpolation

Interpolation (also called "Lerping") is just changing blend weights to do blending over time.

i.e. Turning the knob (t) progressively, not just setting it to some position.

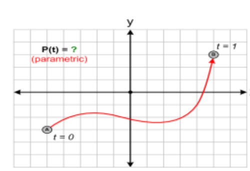
Often we crank slowly from t=0 to t=1.

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
  - ....

Parametric Curves



Parametric curves are curves that are defined using parametric equations.

Bézier Curves

(pronounced "bay-zeeyay")

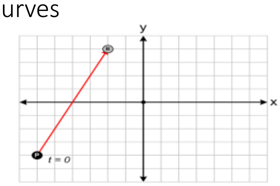
Linear Bézier Curves

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 Bézier Curves



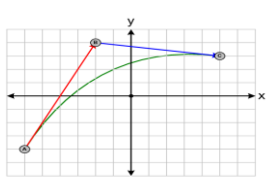
P = ((1-t) * A) + (t * B)   *// weighted average*

or, as I prefer to write it:

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

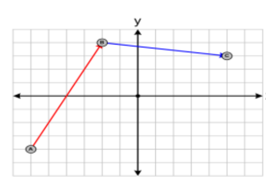
Quadratic Bézier Curves

Quadratic Bézier Curves



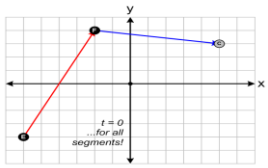
Three **control points**: A, B, and C

Quadratic Bézier Curves



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

Quadratic Bézier Curves

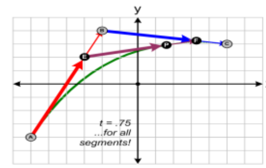


Three **control points**: A, B, and C

Two different Linear Beziers: AB and BC

Instead of "P", using "E" for AB and "F" for BC

Quadratic Bézier Curves



$E(t) = sA + tB$ ← *where* $s = 1-t$

$F(t) = sB + tC$

$P(t) = sE + tF$ ← *technically E(t) and F(t) here*

Quadratic Bézier Curves

One equation to rule them all:

$E(t) = sA + tB$

$F(t) = sB + tC$

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

or

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

or

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

or

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

(BTW, there's our "quadratic" t^2)

Quadratic Bézier Curves

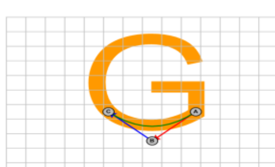
» Remember:

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 Bézier Curves



BONUS: This is also how they make

True Type Fonts look nice and curvy.

Cubic Bézier Curves

» Remember:

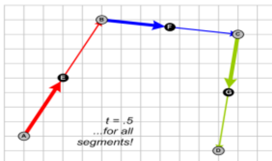
A Cubic Bezier curve is just
a **blend of two Quadratic** Bezier curves.

...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 Bézier Curves



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

Cubic Bézier Curves

Do some hand-waving mathemagic here...
...and we get **one equation to rule them all**:

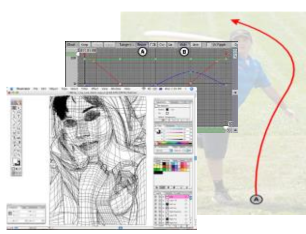
$\boxed{\mathbf{P(t)} = (s^3)A + 3(s^2t)B + 3(st^2)C + (t^3)D}$

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

Cubic Bézier Curves

Seen in lots of places:

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



Other Splines

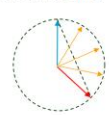
- Catmull-Rom
- Hermite
- Cardinal
- Kochanek-Bartel (KB)
- B-Splines

Angular

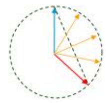
- What is special about angular situations (e.g., π & 2π)?

Simple example

- Subdivide arc, not line



Small Angles



- Apply to lerp

$$(\mathbf{x}_1 - \mathbf{x}_0)t + \mathbf{x}_0$$

- Lerp similar, but can normalize (nlerp)

Large Angles

- How do we represent orientation/transforms?
- What problems do we have with interpolating this?
- How to identify the shortest path and axis in 3D
- Why do we use quaternions?

Quaternion Refresh

Multiplication

- More complex (har) than complex
- Take $\mathbf{q}_0 = (w_0, \mathbf{v}_0)$ $\mathbf{q}_1 = (w_1, \mathbf{v}_1)$

$$\mathbf{q}_1\mathbf{q}_0 = (w_1 w_0 - \mathbf{v}_1 \bullet \mathbf{v}_0, w_1\mathbf{v}_0 + w_0\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_0)$$

- Non-commutative:

$$\mathbf{q}_1\mathbf{q}_0 \neq \mathbf{q}_0\mathbf{q}_1$$

Quaternion Refresh

Identity and Inverse

- Identity quaternion is $(1, 0, 0, 0)$
  - applies no rotation
  - remains at reference orientation
- $\mathbf{q}^{-1}$ is inverse
  - $\mathbf{q} \cdot \mathbf{q}^{-1}$ gives identity quaternion
- Inverse is same axis but opposite angle

Quaternion Interpolation

- As with complex numbers
  - Lerp

$$\mathbf{q}_t = (1-t)\mathbf{q}_0 + t\mathbf{q}_1$$

  - Slerp

$$\mathbf{q}_t = \frac{\sin(1-t)\alpha}{\sin\alpha}\mathbf{q}_0 + \frac{\sin t\alpha}{\sin\alpha}\mathbf{q}_1$$

Quaternion Interpolation

- Technique depends upon the data
- Lerp is generally good enough for most situations
- SLerp generally not used much in graphics (i.e., other than camera/motion control)
  - Also need to normalize quaternion

Conclusion

- Should be able to apply basic interpolation concepts
- Read around on interpolation principles/methods
  - Combine with geometry shader (i.e., smooth surfaces)
- Multi-discipline subject (i.e., not just used in Computer Graphics)