Outline for today

- Inverses of Transforms
- Curves overview
- Bézier curves
Graphics pipeline transformations

- Remember the series of transforms in the graphics pipe:
  - **M** - model: places object in world space
  - **C** - camera: places camera in world space
  - **P** - projection: from camera space to normalized view space
  - **D** - viewport: remaps to image coordinates

- And remember about **C**:
  - handy for positioning the camera as a model
  - *backwards* for the pipeline:
    - we need to get from world space to camera space

- So we need to use **C⁻¹**
  - You’ll need it for project 4: OpenGL wants you to load **C⁻¹** as the base of the MODELVIEW stack
How do we get $\mathbf{C}^{-1}$?

- Could construct $\mathbf{C}$, and use a matrix-inverse routine
  - Would work.
  - But relatively slow.
  - And we didn’t give you one :)
- Instead, let’s construct $\mathbf{C}^{-1}$ directly
  - based on how we constructed $\mathbf{C}$
  - based on shortcuts and rules for affine transforms
Inverse of a translation

- Translate back, i.e., negate the translation vector

\[
T(\bar{v}) = \begin{bmatrix}
1 & 0 & 0 & v_x \\
0 & 1 & 0 & v_y \\
0 & 0 & 1 & v_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
T^{-1}(\bar{v}) = T(-\bar{v}) = \begin{bmatrix}
1 & 0 & 0 & -v_x \\
0 & 1 & 0 & -v_y \\
0 & 0 & 1 & -v_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Easy to verify:

\[
T(-\bar{v}) T(\bar{v}) = \begin{bmatrix}
1 & 0 & 0 & -v_x \\
0 & 1 & 0 & -v_y \\
0 & 0 & 1 & -v_z \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 & v_x \\
0 & 1 & 0 & v_y \\
0 & 0 & 1 & v_z \\
0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & v_x - v_x \\
0 & 1 & 0 & v_y - v_y \\
0 & 0 & 1 & v_z - v_z \\
0 & 0 & 0 & 1
\end{bmatrix} = I
\]
Inverse of a scale

- Scale by the inverses

\[
S(s_x, s_y, s_z) = \begin{bmatrix}
  s_x & 0 & 0 & 0 \\
  0 & s_y & 0 & 0 \\
  0 & 0 & s_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
S^{-1}(s_x, s_y, s_z) = S\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right) = \begin{bmatrix}
  1/s_x & 0 & 0 & 0 \\
  0 & 1/s_y & 0 & 0 \\
  0 & 0 & 1/s_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

- Easy to verify:

\[
S\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right) S(s_x, s_y, s_z) = \begin{bmatrix}
  1/s_x & 0 & 0 & 0 \\
  0 & 1/s_y & 0 & 0 \\
  0 & 0 & 1/s_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  s_x & 0 & 0 & 0 \\
  0 & s_y & 0 & 0 \\
  0 & 0 & s_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} = I
\]
Inverse of a rotation

- Rotate about the same axis, with the oppose angle:
  \[ \mathbf{R}^{-1}(\vec{a}, \theta) = \mathbf{R}(\vec{a}, -\theta) \]

For example:
\[
\mathbf{R}_z(\theta) = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 & 0 \\
\sin(\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\mathbf{R}_z^{-1}(\theta) = \mathbf{R}_z(-\theta) = \begin{bmatrix}
\cos(\theta) & -\sin(-\theta) & 0 & 0 \\
\sin(-\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Inverse of a rotation is the transpose:
  \[ \mathbf{R}^{-1}(\vec{a}, \theta) = \mathbf{R}^T(\vec{a}, \theta) \]
  - Columns of a rotation matrix are orthonormal
  - \( \mathbf{A}^T \mathbf{A} \) produces all columns’ dot-product combinations as matrix
  - Dot product of a column with itself = 1 (on the diagonal)
  - Dot product of a column with any other column = 0 (off the diagonal)
Inverses of composition

- If you have a series of transforms composed together

\[ M = A \ B \ C \ D \]

To invert, compose inverses in the reverse order

\[ M^{-1} = D^{-1} \ C^{-1} \ B^{-1} \ A^{-1} \]

Easy to verify:

\[ M^{-1} \ M = (D^{-1} \ C^{-1} \ B^{-1} \ A^{-1})(A \ B \ C \ D) \]

\[ = D^{-1} \ C^{-1} \ B^{-1} \ A^{-1} \underbrace{A}_{I} \ B \ C \ D \]

\[ = D^{-1} \ C^{-1} \ B^{-1} \underbrace{B}_{I} \ C \ D \]

\[ = D^{-1} \underbrace{C}_{I} \ C \ D \]

\[ = D^{-1} \ D \]

\[ = I \]
Composing with inverses, pictorially

- To go from one space to another, compose along arrows
  - Backwards along arrow: use inverse transform

\[
\text{Lamp in world coords} = M_{\text{table1}} \cdot M_{\text{top1}} \cdot M_{\text{lamp}}
\]

\[
\text{Plant in Tabletop1 coords} = M_{\text{top1}}^{-1} \cdot M_{\text{table1}}^{-1} \cdot M_{\text{table2}} \cdot M_{\text{top2}} \cdot M_{\text{plant}}
\]
Model-to-Camera transform

Model-to-Camera transform is given by

$$\text{Model-to-Camera} = \text{C}^{-1}\text{M}$$
The look-at transformation

- Remember, we constructed $C$ using the look-at idiom:

  Given: eye point $e$, target point $t$, and up vector $\mathbf{u}$
  Construct: columns of camera matrix $C$

  $\mathbf{d} = e$
  $\hat{\mathbf{c}} = \frac{e - t}{|e - t|}$
  $\hat{\mathbf{a}} = \frac{\mathbf{u} \times \hat{\mathbf{c}}}{|\mathbf{u} \times \hat{\mathbf{c}}|}$
  $\mathbf{b} = \hat{\mathbf{c}} \times \hat{\mathbf{a}}$

  Important: $\hat{\mathbf{a}}, \mathbf{b}, \hat{\mathbf{c}}$ are orthonormal
\( C^{-1} \) from \( a, b, c, d \) columns

- If we construct a transform using \( \vec{a}, \vec{b}, \vec{c}, \vec{d} \) columns, it's the same as a composition.
  - First rotate/scale using \( \vec{a}, \vec{b}, \vec{c} \), then translate by \( \vec{d} \):
    \[
    C = \begin{bmatrix}
    a_x & b_x & c_x & d_x \\
    a_y & b_y & c_y & d_y \\
    a_z & b_z & c_z & d_z \\
    0 & 0 & 0 & 1
    \end{bmatrix}
    = \begin{bmatrix}
    1 & 0 & 0 & d_x \\
    0 & 1 & 0 & d_y \\
    0 & 0 & 1 & d_z \\
    0 & 0 & 0 & 1
    \end{bmatrix}
    = T(\vec{d}) \ M
    \]
  - If \( \vec{a}, \vec{b}, \vec{c} \) are orthonormal, they define a pure rotation:
    \[
    C = T(\vec{d}) \ R
    \]
- To take the inverse:
  \[
  C^{-1} = \left( T(\vec{d}) \ R \right)^{-1} = R^{-1} \ T^{-1}(\vec{d})
  \]
  \[
  C^{-1} = R^T \ T(-\vec{d})
  \]
  \[
  C^{-1} = \begin{bmatrix}
    a_x & a_y & a_z & 0 \\
    b_x & b_y & b_z & 0 \\
    c_x & c_y & c_z & 0 \\
    0 & 0 & 0 & 1
    \end{bmatrix}
    \begin{bmatrix}
    1 & 0 & 0 & -d_x \\
    0 & 1 & 0 & -d_y \\
    0 & 0 & 1 & -d_z \\
    0 & 0 & 0 & 1
    \end{bmatrix}
    \]
- Build \( R^T \) using \( \vec{a}, \vec{b}, \vec{c} \) as rows, build \( T(-\vec{d}) \), compose them
- Notice, this does the translation first, then the rotation.

Exercise: what does the final matrix look like?
Outline for today

- Inverses of Transforms
- *Curves overview*
- Bézier curves
Usefulness of curves in modeling

- Surface of revolution
Usefulness of curves in modeling

- Extruded/swept surfaces
Usefulness of curves in modeling

- Surface patches
Usefulness of curves in animation

- Provide a “track” for objects

http://www.f-lohmueller.de/
Usefulness of curves in animation

- Specify parameter values over time: 2D curve editor
How to represent curves

- Specify every point along a curve?
  - Used sometimes as “freehand drawing mode” in 2D applications
  - Hard to get precise results
  - Too much data, too hard to work with generally

- Specify a curve using a small number of “control points”
  - Known as a spline curve or just spline

![Diagram of a vase drawn using control points](image)
Interpolating Splines

- Specify points, the curve goes through all the points
- Seems most intuitive
- Surprisingly, not usually the best choice.
  - Hard to predict behavior
    - Overshoots
    - Wiggles
  - Hard to get “nice-looking” curves
Approximating Splines

- "Influenced" by control points but not necessarily go through them.

- Various types & techniques
  - Most common: (Piecewise) Polynomial Functions
  - Most common of those:
    - Bézier
    - B-spline
  - Each has good properties
  - We’ll focus on Bézier splines
What is a curve, anyway?

- We draw it, think of it as a thing existing in space
- But mathematically we treat it as a function, \( x(t) \)
  - Given a value of \( t \), computes a point \( x \)
  - Can think of the function as moving a point along the curve
The tangent to the curve

- Vector points in the direction of movement
  - (Length is the speed in the direction of movement)
  - Also a function of $t$, written $x'(t)$ or $\frac{dx}{dt}(t)$
Polynomial Functions

- **Linear:** (1st order)  
  \[ f(t) = at + b \]

- **Quadratic:** (2nd order)  
  \[ f(t) = at^2 + bt + c \]

- **Cubic:** (3rd order)  
  \[ f(t) = at^3 + bt^2 + ct + d \]
Point-valued Polynomials (Curves)

- **Linear:** (1\textsuperscript{st} order) \( x(t) = at + b \)
- **Quadratic:** (2\textsuperscript{nd} order) \( x(t) = at^2 + bt + c \)
- **Cubic:** (3\textsuperscript{rd} order) \( x(t) = at^3 + bt^2 + ct + d \)

Each is 3 polynomials “in parallel”:

\[
x_x(t) = a_xt + b_x \quad x_y(t) = a_yt + b_y \quad x_z(t) = a_zt + b_z
\]

- We usually define the curve for \( 0 \leq t \leq 1 \)
How much do you need to specify?

- Two points define a line (1\textsuperscript{st} order)
- Three points define a quadratic curve (2\textsuperscript{nd} order)
- Four points define a cubic curve (3\textsuperscript{rd} order)
- \(k+1\) points define a \(k\)-order curve

- Let’s start with a line…
Linear Interpolation

- **Linear interpolation**, AKA **Lerp**
  - Generates a value that is somewhere in between two other values
  - A ‘value’ could be a number, vector, color, …

- Consider interpolating between two points \( \mathbf{p}_0 \) and \( \mathbf{p}_1 \) by some parameter \( t \)
  - This defines a “curve” that is straight. AKA a first-order spline
  - When \( t=0 \), we get \( \mathbf{p}_0 \)
  - When \( t=1 \) we get \( \mathbf{p}_1 \)
  - When \( t=0.5 \) we get the midpoint

\[
x(t) = Lerp(t, \mathbf{p}_0, \mathbf{p}_1) = (1-t)\mathbf{p}_0 + t \mathbf{p}_1
\]
Linear interpolation

- We can write this in three ways
  - All exactly the same equation
  - Just different ways of looking at it
  - Different properties become apparent

- As a weighted average of the control points:
  \[ x(t) = (1 - t)p_0 + (t)p_1 \]

- As a polynomial in \( t \):
  \[ x(t) = (p_1 - p_0)t + p_0 \]

- In a matrix form:
  \[ x(t) = \begin{bmatrix} p_0 & p_1 \end{bmatrix} \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} t \end{bmatrix} \]}
Linear interpolation as weighted average

\[ x(t) = (1 - t)p_0 + (t)p_1 \]

\[ = B_0(t) p_0 + B_1(t)p_1, \text{ where } B_0(t) = 1 - t \text{ and } B_1(t) = t \]

- Each weight is a function of \( t \)
  - The sum of the weights is always 1, for any value of \( t \)
  - Also known as *blending functions*
Linear interpolation as polynomial

\[ x(t) = \left( \begin{array}{c} p_1 - p_0 \end{array} \right) \cdot t + \begin{array}{c} p_0 \end{array} \]

- Curve is based at point \( p_0 \)
- Add the vector, scaled by \( t \)
Linear interpolation in matrix form

\[
x(t) = \begin{bmatrix} p_0 & p_1 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t \\ 1 \end{bmatrix} = G \cdot B \cdot T
\]

where:

- \( G = \begin{bmatrix} p_0 & p_1 \end{bmatrix} \) is the "Geometry matrix"
- \( B = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \) is the "Geometric Basis"
- \( T = \begin{bmatrix} t \\ 1 \end{bmatrix} \) is the "Polynomial basis"

- Actually, this is shorthand for separate equations for \( x, y, z \)

\[
x_x(t) = \begin{bmatrix} p_{0x} & p_{1x} \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t \\ 1 \end{bmatrix}
\]

\[
x_y(t) = \begin{bmatrix} p_{0y} & p_{1y} \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t \\ 1 \end{bmatrix}
\]

\[
x_z(t) = \begin{bmatrix} p_{0z} & p_{1z} \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t \\ 1 \end{bmatrix}
\]

- Or it can really be put into one matrix

\[
x(t) = \begin{bmatrix} p_{0x} & p_{1x} \\ p_{0y} & p_{1y} \\ p_{0z} & p_{1z} \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t \\ 1 \end{bmatrix}
\]
Linear Interpolation: tangent

- For a straight line, the tangent is constant

\[ x'(t) = p_1 - p_0 \]

- As a weighted average of the control points:

\[ x'(t) = (-1)p_0 + (+1)p_1 \]

- As a (trivial, zero-order) polynomial in \( t \):

\[ x'(t) = 0t + (p_1 - p_0) \]

- In a matrix form:

\[ x'(t) = \begin{bmatrix} p_0 & p_1 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \]
Outline for today

- Inverses of Transforms
- Curves overview
- Bézier curves
Bézier Curves

- Can be thought of as a higher order extension of linear interpolation

Linear

Quadratic

Cubic
Cubic Bézier Curve

- Most common case
  - 4 points for a cubic Bézier
  - Interpolates the endpoints
  - Midpoints are “handles” that control the tangent at the endpoints
  - Easy and intuitive to use

- Many demo applets online
  - [http://www.cs.unc.edu/~mantler/research/bezier/](http://www.cs.unc.edu/~mantler/research/bezier/)
  - [http://www.sunsite.ubc.ca/LivingMathematics/V001N01/UBCExamples/Bezier/bezier.html](http://www.sunsite.ubc.ca/LivingMathematics/V001N01/UBCExamples/Bezier/bezier.html)

- Convex Hull property
- Variation-diminishing property
Bézier Curve Formulation

- Ways to formulate Bézier curves, analogous to linear:
  - Weighted average of control points -- weights are *Bernstein polynomials*
  - Cubic polynomial function of $t$
  - Matrix form
- Also, the *de Casteljau* algorithm: recursive linear interpolations

- Aside: Many of the original CG techniques were developed for Computer Aided Design and manufacturing.
  - Before games, before movies, CAD/CAM was the big application for CG.
  - Pierre Bézier worked for Peugeot, developed curves in 1962
  - Paul de Casteljau worked for Citroen, developed the curves in 1959
Find the point $x$ on the curve as a function of parameter $t$: 

$\mathbf{x}(t)$
de Casteljau Algorithm

- A recursive series of linear interpolations
  - Works for any order. We’ll do cubic
- Not terribly efficient to evaluate this way
  - Other forms more commonly used
- So why study it?
  - Kinda neat
  - Intuition about the geometry
  - Useful for subdivision (later today)
de Casteljau Algorithm

- Start with the control points
- And given a value of $t$
  - In the drawings, $t \approx 0.25$
de Casteljau Algorithm

\[ q_0(t) = \text{Lerp}(t, p_0, p_1) \]
\[ q_1(t) = \text{Lerp}(t, p_1, p_2) \]
\[ q_2(t) = \text{Lerp}(t, p_2, p_3) \]
de Casteljau Algorithm

\[ r_0(t) = Lerp\left(t, q_0(t), q_1(t)\right) \]

\[ r_1(t) = Lerp\left(t, q_1(t), q_2(t)\right) \]
de Casteljau Algorithm

\[ \mathbf{x}(t) = \text{Lerp}(t, \mathbf{r}_0(t), \mathbf{r}_1(t)) \]
de Casteljau algorithm

- Applets
  - http://www2.mat.dtu.dk/people/J.Gravesen/cagd/decast.html
Recursive Linear Interpolation

\[ x = \text{Lerp}(t, r_0, r_1) \]
\[ r_0 = \text{Lerp}(t, q_0, q_1) \]
\[ r_1 = \text{Lerp}(t, q_1, q_2) \]
\[ q_0 = \text{Lerp}(t, p_0, p_1) \]
\[ q_1 = \text{Lerp}(t, p_1, p_2) \]
\[ q_2 = \text{Lerp}(t, p_2, p_3) \]

Diagram:

- \( p_0 \)
- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
Expand the Lerps

\[ q_0(t) = Lerp(t, p_0, p_1) = (1 - t)p_0 + tp_1 \]
\[ q_1(t) = Lerp(t, p_1, p_2) = (1 - t)p_1 + tp_2 \]
\[ q_2(t) = Lerp(t, p_2, p_3) = (1 - t)p_2 + tp_3 \]

\[ r_0(t) = Lerp(t, q_0(t), q_1(t)) = (1 - t)((1 - t)p_0 + tp_1) + t((1 - t)p_1 + tp_2) \]
\[ r_1(t) = Lerp(t, q_1(t), q_2(t)) = (1 - t)((1 - t)p_1 + tp_2) + t((1 - t)p_2 + tp_3) \]

\[ x(t) = Lerp(t, r_0(t), r_1(t)) \]
\[ = (1 - t)((1 - t)((1 - t)p_0 + tp_1) + t((1 - t)p_1 + tp_2)) \]
\[ + t((1 - t)((1 - t)p_1 + tp_2) + t((1 - t)p_2 + tp_3)) \]
Weighted average of control points

- Group this as a weighted average of the points:
  \[ x(t) = (1 - t)(1 - t)(1 - t)p_0 + t(p_1) + t((1 - t)p_1 + t(p_2)) \]
  \[ + t((1 - t)(1 - t)p_1 + t(p_2) + t((1 - t)p_2 + t(p_3))) \]

  \[ x(t) = (1 - t)^3 p_0 + 3(1 - t)^2 t p_1 + 3(1 - t)t^2 p_2 + t^3 p_3 \]

  \[ x(t) = \begin{cases} 
  B_0(t) & (-t^3 + 3t^2 - 3t + 1)p_0 \\
  B_1(t) & (3t^3 - 6t^2 + 3t)p_1 \\
  B_2(t) & (-3t^3 + 3t^2)p_2 \\
  B_3(t) & (t^3)p_3 
  \end{cases} \]
Bézier using Bernstein Polynomials

\[ x(t) = B_0(t)p_0 + B_1(t)p_1 + B_2(t)p_2 + B_3(t)p_3 \]

The cubic Bernstein polynomials:

\[ B_0(t) = -t^3 + 3t^2 - 3t + 1 \]
\[ B_1(t) = 3t^3 - 6t^2 + 3t \]
\[ B_2(t) = -3t^3 + 3t^2 \]
\[ B_3(t) = t^3 \]

\[ \sum B_i(t) = 1 \]

- **Notice:**
  - Weights always add to 1
  - \( B_0 \) and \( B_3 \) go to 1 -- interpolating the endpoints
General Bernstein Polynomials

\[ B_0^1(t) = -t + 1 \]
\[ B_1^1(t) = t \]
\[ B_2^1(t) = t^2 \]

\[ B_0^2(t) = t^2 - 2t + 1 \]
\[ B_1^2(t) = -2t^2 + 2t \]
\[ B_2^2(t) = t^2 \]

\[ B_0^3(t) = -t^3 + 3t^2 - 3t + 1 \]
\[ B_1^3(t) = 3t^3 - 6t^2 + 3t \]
\[ B_2^3(t) = -3t^3 + 3t^2 \]
\[ B_3^3(t) = t^3 \]

\[ B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i \]
\[ \binom{n}{i} = \frac{n!}{i!(n-i)!} \]

\[ \sum B_i^n(t) = 1 \]
General Bézier using Bernstein Polynomials

Bernstein polynomial form of an $n$th-order Bézier curve:

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

$$x(t) = \sum_{i=0}^{n} B_i^n(t) p_i$$
Convex Hull Property

- Construct a convex polygon around a set of points
  - The *convex hull* of the control points
- Any weighted average of the points, with the weights all between 0 and 1:
  - Known as a *convex combination* of the points
  - Result always lies within the convex hull (including on the border)

- Bézier curve is a convex combination of the control points
  - Curve is always inside the convex hull
  - Very important property!
    - Makes curve predictable
    - Allows culling
    - Allows intersection testing
    - Allows adaptive tessellation
Cubic Equation Form

Start with Bernstein form:
\[ x(t) = (-t^3 + 3t^2 - 3t + 1)p_0 + (3t^3 - 6t^2 + 3t)p_1 + (-3t^3 + 3t^2)p_2 + (t^3)p_3 \]

Regroup into coefficients of \( t \):
\[ x(t) = (-p_0 + 3p_1 - 3p_2 + p_3)t^3 + (3p_0 - 6p_1 + 3p_2)t^2 + (-3p_0 + 3p_1)t + (p_0)1 \]

\[
\begin{align*}
x(t) &= at^3 + bt^2 + ct + d \\
a &= (-p_0 + 3p_1 - 3p_2 + p_3) \\
b &= (3p_0 - 6p_1 + 3p_2) \\
c &= (-3p_0 + 3p_1) \\
d &= (p_0)
\end{align*}
\]

- Good for fast evaluation: precompute constant coefficients \((a,b,c,d)\)
- Doesn’t give much geometric intuition
  - But the geometry can be extracted from the coefficients
Aside: linear combinations of points

- Reminder: we can’t scale a point or add two points
  - Can subtract two points
  - Can take weighted average of points if the weights add up to one
  - Act on homogeneous points: w component of result must be 1

\[ \frac{1}{4} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \frac{3}{4} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ OK point} \]

- Can also take weighted average of points if the weights add up to 0
  - The result gives w=0, i.e. a vector
  - E.g. \( p - q \) is the same as \( (+1)p + (-1)q \)
  - Can also do \((-1)p_0 + (3)p_1 + (-3)p_2 + (1)p_3\)

\[ -1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 3 \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 3 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ OK vector} \]
Cubic Equation, vector notation

\[ x(t) = \vec{a}t^3 + \vec{b}t^2 + \vec{c}t + d \]

\[ \vec{a} = (-p_0 + 3p_1 - 3p_2 + p_3) \]
\[ \vec{b} = (3p_0 - 6p_1 + 3p_2) \]
\[ \vec{c} = (-3p_0 + 3p_1) \]
\[ d = (p_0) \]

- Curve is based at \( p_0 \) AKA \( d \)
- Increasing \( t \) introduces the other vectors:
  - first order: \( \vec{c} \) -- moves towards \( p_1 \)
  - second order: \( \vec{b} \) -- subtracts off \( \vec{c} \), pulls towards \( p_2 \)
  - third order: \( \vec{a} \) -- subtracts off everything, moves towards \( p_3 \)
Cubic Matrix Form

\[ x(t) = \begin{bmatrix} \vec{a} & \vec{b} & \vec{c} & \vec{d} \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \]

\[ \vec{a} = (-p_0 + 3p_1 - 3p_2 + p_3) \]
\[ \vec{b} = (3p_0 - 6p_1 + 3p_2) \]
\[ \vec{c} = (-3p_0 + 3p_1) \]
\[ \vec{d} = (p_0) \]

\[ x(t) = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \]

- Other cubic splines use different basis matrix \( B \)
  - Hermite, Catmull-Rom, B-Spline, …
Cubic Matrix Form

- 3 parallel equations, in x, y and z:

\[
x_x(t) = \begin{bmatrix} p_{0x} & p_{1x} & p_{2x} & p_{3x} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}
\]

\[
x_y(t) = \begin{bmatrix} p_{0y} & p_{1y} & p_{2y} & p_{3y} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}
\]

\[
x_z(t) = \begin{bmatrix} p_{0z} & p_{1z} & p_{2z} & p_{3z} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}
\]
Matrix Form

- Bundle into a single matrix

\[
x(t) = \begin{bmatrix}
p_{0x} & p_{1x} & p_{2x} & p_{3x} \\
p_{0y} & p_{1y} & p_{2y} & p_{3y} \\
p_{0z} & p_{1z} & p_{2z} & p_{3z}
\end{bmatrix} \begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 0 \\
-3 & 3 & 0 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix} \begin{bmatrix}
t^3 \\
t^2 \\
t \\
1
\end{bmatrix}
\]

- Evaluate quickly:
  - Precompute \( C \)
  - Take advantage of existing 4x4 matrix hardware support
Tangent

- The derivative of a curve represents the tangent vector to the curve at some point.

\[ \frac{dx}{dt}(t) \]

\( x(t) \)
Tangent

- Computing the tangent of a polynomial curve is easy:

\[ x(t) = \bar{a}t^3 + \bar{b}t^2 + \bar{c}t + \bar{d} \quad x'(t) = \frac{dx}{dt}(t) = 3\bar{a}t^2 + 2\bar{b}t + \bar{c} \]

\[ x(t) = \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \quad x'(t) = \frac{dx}{dt}(t) = \begin{bmatrix} \bar{a} \\ \bar{b} \\ \bar{c} \\ \bar{d} \end{bmatrix} \begin{bmatrix} 3t^2 \\ 2t \\ 1 \\ 0 \end{bmatrix} \]

- Notice \( x'(t) \) is a vector
  - Doesn’t depend on \( \bar{d} \)
  - Doesn’t depend on position of curve
Transforming Bézier curves

Two ways to transform a Bézier curve

- Transform the control points, then compute resulting spline point
- Compute spline point, then transform it

Either way, get the same point!

- Curve is defined via affine combination of points
- Invariant under affine transformations
- Convex hull property always remains
Drawing Bézier Curves

- How can you draw a curve?
  - Generally no low-level support for drawing curves
  - Can only draw line segments or individual pixels

- Approximate the curve as a series of line segments
  - Analogous to tessellation of a surface
  - Methods:
    - Sample uniformly
    - Sample adaptively
    - Recursive Subdivision
Uniform Sampling

- Approximate curve with \( N \) straight segments
  - \( N \) chosen in advance
  - Evaluate \( x_i = x(t_i) \) where \( t_i = \frac{i}{N} \) for \( i = 0, 1, \ldots, N \)
    \[
x_i = \bar{a} \frac{i^3}{N^3} + \bar{b} \frac{i^2}{N^2} + \bar{c} \frac{i}{N} + d
    \]
  - Connect the points with lines

- Too few points?
  - Bad approximation
  - “Curve” is faceted

- Too many points?
  - Slow to draw too many line segments
  - Segments may draw on top of each other
Adaptive Sampling

- Use only as many line segments as you need
  - Fewer segments needed where curve is mostly flat
  - More segments needed where curve bends
  - No need to track bends that are smaller than a pixel

- Various schemes for sampling, checking results, deciding whether to sample more

- Or, use knowledge of curve structure:
  - Adapt by recursive subdivision
Recursive Subdivision

- Any cubic curve segment can be expressed as a Bézier curve
- Any piece of a cubic curve is itself a cubic curve
- Therefore:
  - Any Bézier curve can be broken up into smaller Bézier curves
  - But how…?
de Casteljau subdivision

- de Casteljau construction points are the control points of two Bézier sub-segments
Adaptive subdivision algorithm:

- Use de Casteljau construction to split Bézier segment
- Examine each half:
  - If flat enough: draw line segment
  - Else: recurse

To test if curve is flat enough
- Only need to test if hull is flat enough
  - Curve is guaranteed to lie within the hull
- e.g., test how far the handles are from a straight segment
  - If it’s about a pixel, the hull is flat
Done

- Next class:
  - Extending to longer curves
  - Extending to curved surfaces