Curves and Surfaces

Outline

• Parametric curves and surfaces
• Subdivision
• Catmull-Rom Splines
Parametric Lines

- Parametric definition of a line segment:
  \[ p(t) = (1 - t)p_0 + tp_1 \]

Linear Interpolation as Blending

- Consider each point on the line segment as a sum of control points \( p_i \) weighted by blending functions \( B_i \):
  \[ p(t) = \sum_{i=0}^{n} B_i^n(t)p_i \]
  - Here we have \( B_0 = 1 - t \) and \( B_1 = t \)
Improving Interpolation

- $C^n$ continuity: $n^{th}$ derivative is continuous everywhere on the curve
- Linear interpolation over multiple connected line segments has $C^0$ continuity, but not $C^1$ or higher continuity, which would make for a smoother curve

![Image of C0 vs C1 continuity]

Interpolating Interpolants

- For 3 points $a$, $b$, and $c$, we can define a smoother curve by linearly interpolating along the line between points $d$ and $e$ linearly interpolated between $a$, $b$ and $b$, $c$, respectively
- This curve approximates $a$, $b$, and $c$, because it doesn’t go through all of them
- True interpolating curves include all of the original points

![Image of interpolating points and curve]
Interpolating Interpolants

\[ p(t) = (1 - t)d + te \]
\[ = (1 - t)[(1 - t)a + tb] + t[(1 - t)b + tc] \]

Changing notation...

\[ p(t) = (1 - t)[(1 - t)p_0 + tp_1] + t[(1 - t)p_1 + tp_2] \]
\[ = (1 - t)^2 p_0 + 2t (1 - t)p_1 + t^2 p_2 \]
\[ = \sum_{i=0}^{n} B_i^n(t)p_i \quad \text{now } n = 2, \text{ quadratic blending} \]
Quadratic Blending Functions

- Blending functions are also called **Bernstein** polynomials

\[
\begin{align*}
(1 - t)^2 & \quad t^2 \\
2t(1 - t) & \\
\end{align*}
\]

Bézier Curves

- Curve approximation through **recursive** application of linear interpolations
  - Linear: 2 control points, 2 linear Bernstein polynomials
  - Quadratic: 3 control points, 3 quadratic Bernstein polynomials
    - \( N \) control points = \( N - 1 \) degree curve

- Notes
  - Only endpoints are interpolated (i.e., on the curve)
  - Every control point affects every point on curve
    - Makes modeling harder
Extension to Surfaces

• Multiply blending functions for each dimension together
  – Bilinear patch (same as bilinear interpolation for texture mapping)
    • Need 2 x 2 control points
  – Biquadratic Bézier patch
    • Need 3 x 3 control points

Piecewise Bézier Curves

• Near equality of degree of curve to number of control points makes long curves expensive to evaluate
• Idea:
  – Compute low-degree curves Bézier over short subsequences of control points
  – Join together so as to maintain continuity
• Just setting last control point of first curve equal to first control point of last curve does not ensure C^1 continuity
  – Must make tangents coincide, etc.
Piecewise Bézier Curves

- Near equality of degree of curve to number of control points makes long curves expensive to evaluate
- Idea:
  - Compute low-degree curves Bézier over short subsequences of control points
  - Join together so as to maintain continuity
- Just setting last control point of first curve equal to first control point of last curve does not ensure $C^1$ continuity
  - Must make tangents coincide, etc.

B-splines

- “Piecewise Bézier curves done right”
  - Only use low-degree polynomials for many control points
  - Local modeling control
- Idea: Smoothly reduce each control point’s influence on curve shape to zero outside of small local window
- Cubic B-splines
  - 4 adjacent control points have non-zero values of blending function
  - Guarantees $C^1$ continuity
B-spline blending functions of other degrees

Degree 0

Degree 1: Linear

Degree 2: Quadratic

Degree 3: Cubic

Curve Subdivision

• Goal: Obtain smooth curves starting from small number of line segments
• One approach: **Corner-cutting** subdivision
  - Repeatedly chop off corners of polygon
  - **Limit curve** is shape that would be reached after an infinite series of such subdivisions
Midpoint Corner-cutting Algorithm

```c
function midpoint_subdivide(p0, p1, p2, depth)
{
    if (depth > 0) {
        p01 = (p0 + p1)/2;
        p12 = (p1 + p2)/2;
        pm = (p01 + p12)/2;
        midpoint_subdivide(p0, p01, pm, depth - 1);
        midpoint_subdivide(pm, p12, p2, depth - 1);
    }  // else draw
}
```

Bézier curves and B-splines via subdivision

- The midpoint corner-cutting algorithm is a subdivision definition of quadratic Bézier curves
- **Chaikin**'s subdivision scheme defines quadratic B-splines
  - Chop off “1/4” corner
Surface Subdivision

- Analogous to curve subdivision:
  1. **Refine mesh**: Choose new vertices, update connectivity
  2. **Smooth mesh**: Move vertices

Loop subdivision

- Smooths **triangle** mesh
- Subdivision replaces 1 triangle with 4

- Approximating scheme
  - Original vertices not guaranteed to be in subdivided mesh
Loop subdivision: Example

Catmull-Rom spline

- Similar to cubic B-splines in that only 4 control points have non-zero influence over each section of curve
  - But it’s **interpolating**
- Four points define curve between 2\textsuperscript{nd} and 3\textsuperscript{rd}
Catmull-Rom spline: Example

Catmull-Rom spline

• We want a cubic polynomial curve defined parametrically over the interval \( t \) in \([0, 1]\) which starts at \( P_0 \) and ends at \( P_1 \), with starting and ending slopes of \( P_0^0 \) and \( P_1^0 \), respectively.
• It has this general form:

\[
P(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3
\]
Inferring the Coefficients

\[ P(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \]

- The control points are located at \( t = 0 \) and \( t = 1 \) on the curve segment, so we can relate them to the polynomial coefficients as follows:

\[
\begin{align*}
P(0) &= a_0 \\
P(1) &= a_0 + a_1 + a_2 + a_3 \\
P'(0) &= a_1 \\
P'(1) &= a_1 + 2a_2 + 3a_3
\end{align*}
\]

Catmull-Rom spline

Solving for the coefficients and grouping terms yields:

\[ P(t) = (1 - 3t^2 + 2t^3)P(0) + (3t^2 - 2t^3)P(1) + (t - 2t^2 + t^3)P'(0) + (-t^2 + t^3)P'(1) \]

which can be written in matrix form as:

\[
P(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 & 3 & 2 & 1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} P(0) \\ P(1) \\ P'(0) \\ P'(1) \end{bmatrix}
\]
Catmull-Rom: Tangents

- To compute the tangent $p_k^0$ at a control point $p_k$ along the curve segment, use chord approximation
  - Average of vector from point before $p_{k-1}$ to $p_k$ and vector from $p_k$ to point after $p_{k+1}$
  - This is half of vector between $p_{k-1}$ and $p_{k+1}$

\[ p_k' = \frac{p_{k+1} - p_{k-1}}{2} \]
Incorporating Tangent Approximations

• This changes our previous expression from:

\[ \mathbf{P}(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}(0) \\ \mathbf{P}(1) \\ \mathbf{P}'(0) \\ \mathbf{P}'(1) \end{bmatrix} \]


to:

\[ \mathbf{P}(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}_{i+1}-\mathbf{P}_{i-1} \\ \frac{1}{2} \mathbf{P}_{i+2}-\mathbf{P}_i \end{bmatrix} \]

courtesy of K. Joy

Catmull-Rom: Removing derivative terms

• Now factor out arithmetic to turn the tangents into the underlying control points. Go from:

\[ \mathbf{P}(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}_i \\ \mathbf{P}_{i+1} \\ \frac{1}{2} \mathbf{P}_{i+2}-\mathbf{P}_i \end{bmatrix} \]

to:

\[ \mathbf{P}(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \mathbf{P}_{i-1} \\ \mathbf{P}_i \\ \mathbf{P}_{i+1} \\ \mathbf{P}_{i+2} \end{bmatrix} \]

courtesy of K. Joy
Catmull-Rom: Blending matrix*

- Combine to get final blending matrix:

\[
\begin{bmatrix}
0 & 2 & 0 & 0 \\
-1 & 0 & 1 & 0 \\
2 & -5 & 4 & -1 \\
-1 & 3 & -3 & 1
\end{bmatrix}
\]

- To trace curve, iterate through subsets of 4 control points \((p_{i-1}, p_i, p_{i+1}, p_{i+2})\)
  - Inner loop iterates over \(t\) in \([0, 1]\)

*Valid for all but first and last segments of open curve (use different method there for tangents)

---

Catmull-Rom spline blending functions

\(B_0\) \hspace{2cm} \(B_1\) \hspace{2cm} \(B_2\) \hspace{2cm} \(B_3\)

(from Hearn & Baker)
Splines for camera motion: Example

- Use Catmull-Rom spline to define smooth camera path—e.g., a roller coaster
- Then just keep calling `gluLookat()` while tracing the curve

Video demonstration:
http://www.youtube.com/watch?v=WwLWwaMr1YM&feature=related