Curves

- So far we've seen polylines
  - GL_LINE_STRIP, etc.
- Smooth curves would be better for
  - building models
  - animation
- Possible representations
  - Explicit
    \[ y = f(x) \]
    - can use only if the curve is a function
  - Implicit
    \[ f(x, y, z) = 0 \]
    - difficult to work with
  - Parametric
    \[ (f(u), g(u)) \]
- Our choice: parametric curves where the functions are all polynomials in the parameter
  - easy (and efficient) to compute
  - infinitely differentiable
  - we'll look at ones defined by control points and an algorithm operating on them

General formula

\[
V'_0 = (1-u)V_0 + uV_1 \\
V'_1 = (1-u)V_1 + uV_2 \\
\vdots \\
V'_n = (1-u)V'_0 + uV'_1 \\
= (1-u)((1-u)V_0 + uV_1) + u((1-u)V_1 + uV_2) \\
= (1-u)^2V_0 + 2(1-u)uV_1 + u^2V_2 \\
\vdots \\
B(u) = \sum_{k=0}^{n} \binom{n}{k}(1-u)^{(n-k)}u^kV_k
\]

Bernstein polynomials

- The coefficients of the control points are functions called the Bernstein polynomials
  \[
  B_0(u) = (1-u)^3 \\
  B_1(u) = 3u(1-u)^2 \\
  B_2(u) = 3u^2(1-u) \\
  B_3(u) = u^3
  \]
- Useful properties (when 0 <= u <= 1)
  - each is between 0 and 1
  - they add to exactly 1
  \[ 1 = 1^3 = ((1-u)+u)^3 = B_0(u)+B_1(u)+B_2(u)+B_3(u) \]
- Convex combination of control points implies that the curve lies within their convex hull
Beziers Java applet

- Try this online at http://www.gris.uni-tuebingen.de/projects/ilo/repository/bezier.jar
- Move the
  - interpolation point, see how the others (and the point on curve) move
  - control points (can even make loops)

Displaying Bézier curves

- How could we draw one of these things?
- It would be nice if we had an adaptive algorithm, that would take into account curvature / flatness

Subdivide and conquer

```python
def DisplayBezier(V):
    assert(len(V) == 4)
    if FlatEnough(V):
        Line(V[0], V[3])
    else:
        L, R = Subdivide(V)
        DisplayBezier(L)
        DisplayBezier(R)
```

Testing for flatness

- Compare total length of control polygon to length of line connecting endpoints:

\[
\left| V_0 - V_1 \right| + \left| V_1 - V_2 \right| + \left| V_2 - V_3 \right| < 1 + \varepsilon
\]
More complex curves

- Suppose we want to draw a more complex curve
  Why not use a high-order Bézier?
  High order polynomials are difficult to control

- Instead, we’ll splice together a curve from individual segments that are cubic Béziers
  Why cubic?
  Lowest dimension with control for the second derivative
  Lowest dimension for non-planar polynomial curves

- There are three properties we’d like to have in our newly constructed splines...

Local control

- Every control point affects every point on the curve (except the endpoints)

- Moving a single control point affects the whole curve!

- We’d like our spline to have **local control**
  - that is, have each control point affect some well-defined neighborhood around that point

Interpolation

- Bézier curves are **approximating**
  - The curve does not (necessarily) pass through all the control points
  - Each point pulls the curve toward it, but other points are pulling as well

- Instead, we may prefer a spline that is **interpolating**
  - That is, that always passes through every control point

Continuity

- We want our curve to have **continuity**
  - There shouldn’t be an abrupt change when we move from one segment to the next.

- There are nested degrees of continuity:
  - Not C⁰:
  - C¹, C²:
  - C³, C⁴, ...:

  Superscript tells how many derivatives are continuous
Ensuring continuity

- C² continuous curves would be nice
- Since the functions defining a Bézier curve are polynomial
  - all their derivatives exist and are continuous
  - therefore, we only need to worry about the derivatives at the endpoints of the curve

$$B(u) = \sum_{k} \binom{3}{k} (1-u)^{3-k} u^k V_k$$

$$= (1-u)^3 V_0 + 3(1-u)^2 u V_1 + 3(1-u) u^2 V_2 + u^3 V_3$$

$$= (1-3u+3u^2-u^3) V_0 + 3(u-2u^2+u^3) V_1 + 3(u^2-u^3) V_2 + u^3 V_3$$

$$= \left[ \begin{array}{c} -u^3 \\ 3u^2 - 3u^3 + u^3 \end{array} \right] V_0 \\
= \left[ \begin{array}{c} 3u^2 - 6u^3 + 3u^2 - 2u^3 + u^3 \end{array} \right] V_1 \\
= \left[ \begin{array}{c} -3u^3 + 3u^2 - 2u^3 + u^3 \end{array} \right] V_2 \\
= \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] V_3$$

Evaluating derivatives

- How do we get derivatives w.r.t. u?

- What is $B'(0)$?

- What is $B''(0)$?

- What is $B'(1)$?

- What is $B''(1)$?

$$B'(u) = \left[ \begin{array}{ccc} 3 & -6 & 3 \\ -3 & 3 & 0 \end{array} \right] V_0$$

$$= \left[ \begin{array}{ccc} 3 & -6 & 3 \\ -3 & 3 & 0 \end{array} \right] V_1$$

$$= \left[ \begin{array}{ccc} -3 & 3 & 0 \\ 1 & 0 & 0 \end{array} \right] V_2$$

$$= \left[ \begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \end{array} \right] V_3$$

$$B'(u) = \left[ \begin{array}{ccc} 3 & -6 & 3 \\ -3 & 3 & 0 \end{array} \right] V_0$$

$$= \left[ \begin{array}{ccc} 3 & -6 & 3 \\ -3 & 3 & 0 \end{array} \right] V_1$$

$$= \left[ \begin{array}{ccc} -3 & 3 & 0 \\ 1 & 0 & 0 \end{array} \right] V_2$$

$$= \left[ \begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \end{array} \right] V_3$$

$$B''(u) = \left[ \begin{array}{ccc} -1 & 3 & -3 \\ 3 & -6 & 3 \end{array} \right] V_0$$

$$= \left[ \begin{array}{ccc} -1 & 3 & -3 \\ 3 & -6 & 3 \end{array} \right] V_1$$

$$= \left[ \begin{array}{ccc} 3 & -6 & 3 \\ -3 & 3 & 0 \end{array} \right] V_2$$

$$= \left[ \begin{array}{ccc} -3 & 3 & 0 \\ 1 & 0 & 0 \end{array} \right] V_3$$

Ensuring C² continuity

- Given a cubic Bézier segment $(V_0, V_1, V_2, V_3)$
  - add another curve $(W_0, W_1, W_2, W_3)$ to it
  - in such a way that the joint is C²

- But first, if $a$ and $b$ are points, what is $(2a-b)$?

$$2a-b = a + (a-b)$$

"mirror b w.r.t. a"

Derivatives at the endpoints

- In general, the $n^{th}$ derivative at an endpoint depends only on the $n+1$ points nearest that endpoint
- Geometrical interpretation of derivatives?
  1st derivative at start 3 times the vector $V_1 - V_0$ ($V_3 - V_2$ at the end)
  2nd derivative 6 times the vector sum of $V_0 - V_1$ and $V_2 - V_1$
Ensuring $C^2$ continuity

- **$C^0$ constraint:**
  \[ B(0) = B(1) \]
  \[ \Rightarrow \ W_0 = V_3 \]
  "$W_0 = \text{Attach end points}"

- **$C^1$ constraint:**
  \[ B'(0) = B'(1) \]
  \[ \Rightarrow \ W_1 - W_0 = V_3 - V_2 \]
  \[ \Rightarrow \ W_1 = 2V_3 - V_2 \]

- **$C^2$ constraint:**
  \[ B''(0) = B''(1) \]
  \[ \Rightarrow \ W_0 - 2W_1 + W_2 = V_1 - 2V_2 + V_3 \]
  \[ \Rightarrow \ W_2 = 2W_1 + V_1 - 2V_2 \]

  "$W_2 = \text{First mirror } V_1 \text{ w.r.t } V_2$, then that w.r.t $W_1$"

Only $W_3$ remains free!

Beziers in Blender

- **Add 2 curves**
  - SPACE -> Add -> Curve -> Bezier Curve
  - $C^0$ continuous

- **Connect the segments**
  - select ends with Rclick and SHIFT+Rclick
  - hit f ("make a face")

- **The Bezier controls for a segment are**
  - two consecutive vertices on curve ($V_0$, $V_3$)
  - the handles of $V_0$ and $V_3$ on segment's side

- **Add vertices**
  - select one vertex with right button
  - CTRL + left click adds them

Beziers in Blender

- **There are four handle types**
  - SHIFT-h "auto" (yellow)
    - tries to keep curve smooth
    - not really $C^2$ though
  - $h$ toggles "aligned" (pink) and "free" (black)
    - aligned is tangent continuous but not really $C^1$
    - free is only $C^0$
  - $v$ "vertex" (green)
    - like free except tangents aim directly at the other ends of segments

Building a complex spline

- **Constraining a Bezier curve made of many segments to be $C^2$ continuous is a lot of work**
  - for each new segment we have to add 3 new control points
  - only one of the control points is really free

- **B-splines are easier (and $C^2$)**
  - First specify 4 vertices (de Boor points), then one per segment

How many segments here?

3, four de Boor points for the first segment, then one per segment
Building a complex spline

• Where are the Bézier control points?
  - $V_1 = (2B_1 + B_2) / 3$
  - $V_0 = ((B_0 + 2B_1)/3 + (2B_1 + B_2)/3) / 2 = (B_0 + 4B_1 + B_2) / 6$

• Express them in terms of the de Boor points
  - $B_1 = 2V_1 - V_2$
  - $B_0 = 3[(2V_0 - V_1) - B_1] + B_1 = 6V_0 - 3V_1 - 2B_1$
    $= 6V_0 - 3V_1 - 2(V_1 - V_2) = 2V_2 - V_1 + 6(V_0 - V_1)$
    $= B_2 + 6(V_0 - V_1)$

Example

• How to get Bézier controls from B-spline controls?
  - split $B_1 – B_2$ segment into three parts, put $V_1$ and $V_2$ there
  - split $B_0 – B_1$ segment into three parts, put $V_0$ in the middle between $V_1$ and the segment closest to $B_1$
  - repeat for $V_3$

• How to get Bézier controls from B-spline controls?
  - extend $V_1 – V_2$ so it’s three times longer, get $B_1$ and $B_2$
  - reflect $V_1$ w.r.t. $V_0$, get a helper point
  - extend the helper point three-fold, get $B_0$
  - repeat for $B_3$
**Endpoints of B-splines**

- We can see that B-splines don’t interpolate the de Boor points.
- It would be nice if we could at least control the endpoints of the splines explicitly.
- There’s a hack to make the spline begin and end at control points by repeating them.
  - How many times? $3$
  - $V_0 = (B_0 + 4B_1 + B_2) / 6$
  - three control points affect a beginning or ending Bezier point.

---

**Finding the derivatives**

- Now what we need to do is solve for the derivatives.
- To do this we’ll use the $C^2$ continuity requirement.
  - end points match
  - tangents match
  - second derivatives match

$$
\begin{align*}
V_0 &= C_0 & W_0 &= C_1 \\
V_1 &= C_0 + \frac{1}{3}D_0 & W_1 &= C_1 + \frac{1}{3}D_1 \\
V_2 &= C_1 - \frac{1}{3}D_1 & W_2 &= C_2 - \frac{1}{3}D_2 \\
V_3 &= C_1 & W_3 &= C_2 \\
6(V_1 - 2V_2 + V_3) &= 6(W_0 - 2W_1 + W_2) \\
=> 3V_1 - 6V_2 &= -6W_1 + 3W_2 \\
=> 3C_0 + D_0 - 6C_1 + 2D_1 &= -6C_1 - 2D_1 + 3C_2 - 2D_2 \\
=> D_0 + 4D_1 + D_2 &= 3(C_2 - C_0)
\end{align*}
$$

---

**$C^2$ interpolating splines**

- Of the 3 nice things (continuity, local control, interpolation) we don’t have the last one.
- Here’s the idea behind $C^2$ interpolating splines.
  - suppose we had cubic Béziers connecting control points $C_0$, $C_1$, $C_2$, …
  - and that we somehow knew the first derivative at each point (which we don’t…)

$$
D_0 + 4D_1 + D_2 = 3(C_2 - C_0) \\
D_1 + 4D_2 + D_3 = 3(C_3 - C_1) \\
\vdots \\
D_{m-2} + 4D_{m-1} + D_m = 3(C_m - C_{m-2})
$$

- How many equations is this? $m-1$
- How many unknowns are we solving for? $m+1$
Not quite done yet

- We have two additional degrees of freedom, which we can nail down by imposing more conditions on the curve.
- There are various ways to do this. We'll use the variant called natural $C^2$ interpolating splines, which requires the second derivative to be zero at the endpoints.
- This condition gives us the two additional equations we need. At the starting point, it is:

$$6(V_0 - 2V_1 + V_2) = 0$$

$$=> C_0 - 2(C_0 + D_0/3) + C_1 - D_1/3 = 0$$

$$=> -3C_0 -2D_0 + 3C_1 - D_1 = 0$$

$$=> 2D_0 + D_1 = 3(C_1 - C_0)$$

Solving for the derivatives

- Let’s collect our $m+1$ equations into a single linear system:

$$\begin{bmatrix} 2 & 1 & 0 & \cdots & 0 \\ 1 & 4 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 4 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 1 & \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{m-1} \\ D_m \end{bmatrix} = \begin{bmatrix} 3(C_1 - C_0) \\ 3(C_2 - C_0) \\ 3(C_3 - C_1) \\ \vdots \\ 3(C_m - C_{m-2}) \\ 3(C_m - C_{m-1}) \end{bmatrix}$$

- It’s easier to solve than it looks.
- We can use forward elimination to zero out everything below the diagonal, then back substitution to compute each $D$ value.

$C^2$ interpolating spline

- Once we’ve solved for the real $D_i$s, we can plug them in to find our Bézier control points and draw the final spline:

A third option

- If we’re willing to sacrifice $C^2$ continuity, we can get interpolation and local control.
- Instead of finding the derivatives by solving a system of continuity equations, just pick something arbitrary but local.
- If we set each derivative to be a constant multiple of the vector between the previous and next controls, we get a Catmull-Rom spline.
**Catmull-Rom splines**

- The math for Catmull-Rom splines is pretty simple:

\[
V_0 = C_i \\
V_1 = C_i + \frac{1}{2}(C_2 - C_0) \\
V_2 = C_2 + \frac{1}{2}(C_3 - C_1) \\
V_3 = C_2
\]

**Surfaces of revolution**

- Idea: rotate a 2D profile curve around an axis

**Constructing surfaces of revolution**

- Given: A curve \( C(u) \) in the yz-plane:

\[
C(u) = \begin{bmatrix}
0 \\
c_y(u) \\
c_z(u) \\
1
\end{bmatrix}
\]

- Let \( R_z(v) \) be a rotation about the z-axis

- Find: A surface \( S(u,v) \) which is \( C(u) \) rotated about the z-axis

\[
S(u,v) = \begin{bmatrix}
\sin(2\pi v)c_y(u) \\
\cos(2\pi v)c_z(u) \\
c_z(u) \\
1
\end{bmatrix}
\]

**General sweep surfaces**

- The surface of revolution is a special case of a swept surface

- Idea: Trace out surface \( S(u,v) \) by moving a profile curve \( C(u) \) along a trajectory curve \( T(v) \)

- More specifically:
  - suppose that \( C(u) \) lies in an \( (x_c,y_c) \) coordinate system with origin \( O_c \)
  - for every point along \( T(v) \), lay \( C(u) \) so that \( O_c \) coincides with \( T(v) \)
**Orientation**

- The big issue:
  - How to orient $C(u)$ as it moves along $T(v)$?

- Here are two options:
  1. **Fixed (or static)**
     - just translate $O_c$ along $T(v)$
  2. **Moving**
     - use the **Frenet frame** of $T(v)$
     - allows smoothly varying orientation
     - permits surfaces of revolution, for example

**Frenet frames**

- Motivation: Given a curve $T(v)$, we want to attach a smoothly varying coordinate system

$$
\hat{t}(v) = \text{normalize}(T'(v)) \\
\hat{b}(v) = \text{normalize}(T'(v) \times T''(v)) \\
\hat{n}(v) = \hat{b}(v) \times \hat{t}(v)
$$

- As we move along $T(v)$, the Frenet frame $(t,b,n)$ varies smoothly

**Frenet swept surfaces**

- Orient the profile curve $C(u)$ using the Frenet frame of the trajectory $T(v)$:
  - put $C(u)$ in the **normal plane** $nb$
  - place $O_c$ on $T(v)$
  - align $x_c$ for $C(u)$ with $-n$
  - align $y_c$ for $C(u)$ with $b$

- If $T(v)$ is a circle, you get a surface of revolution exactly!

- Several variations are possible:
  - scale $C(u)$ as it moves, possibly using length of $T(v)$ as a scale factor
  - morph $C(u)$ into some other curve as it moves along $T(v)$
  - …

**Tensor product Bézier surfaces**

- Given a grid of control points $V_{ij}$ forming a **control net**, construct a surface $S(u,v)$ by:
  - treating rows of $V$ as control points for curves $V_0(u),..., V_{n}(u)$
  - treating $V_0(u),..., V_{n}(u)$ as control points for a curve parameterized by $v$
**Tensor product surfaces, cont.**

- Which control points are interpolated by the surface?

**Corners**

- Tensor product surfaces can be written out explicitly:

\[
S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} V_{ij} B_i^n(u) B_j^m(v)
\]

\[
= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M_{\text{Bézier}} V M_{\text{Bézier}}^T
\]

where \( M_{\text{Bézier}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \)

**Matrix form**

**Tensor product B-spline surfaces**

- As with spline curves, we can piece together a sequence of Bézier surfaces to make a spline surface.

- If we enforce C2 continuity and local control, we get B-spline surfaces:
  - treat rows of \( B \) as control points to generate control points in \( u \)
  - treat those as control points in \( v \)
  - generate surface points from control points in \( v \)

**Tensor product B-splines, cont.**

- Which B-spline control points are interpolated by the surface?

  **In general, none**
**Trimmed NURBS surfaces**

- Uniform B-spline surfaces are a special case of NURBS surfaces.
- Sometimes, we want to have control over which parts of a NURBS surface get drawn.
- We can do this by **trimming** the $u$-$v$ domain.
  - Define a closed curve in the $u$-$v$ domain (a trim curve).
  - Do not draw the surface points inside of this curve.

- It's really hard to maintain continuity in these regions, especially while animating.

---

**Building more complex models**

- Bezier and B-spline surfaces require a regular control point network.
- Wouldn't it be nice to use arbitrary meshes as a control mesh?

---

**Subdivision surfaces**

- We defined Bezier curves through subdivision.
  - Let's generalize that idea for surfaces.
- Iteratively refine a **control polyhedron** (or control mesh) to produce the limit surface using splitting and averaging steps:
  - Splitting creates a denser mesh (usually doesn't change shape).
  - Averaging moves control points (usually makes the mesh smoother).

- There are two types of splitting steps:
  - Vertex schemes
  - Face schemes

---

**Vertex schemes**

- A vertex surrounded by $n$ faces is split into $n$ subvertices, one for each face:

- Doo-Sabin subdivision:
**Face schemes**

- Each quadrilateral face is split into four subfaces:

  ![Original](image1) ![After splitting](image2)

- Catmull-Clark subdivision:

  ![Subdivision](image3)

**Face schemes, cont.**

- Each triangular face is split into four subfaces:

  ![Original](image4) ![After splitting](image5)

- Loop subdivision:

  ![Subdivision](image6)

**Averaging step**

- We can use masks for the averaging step:
  - (these masks for Loop’s scheme)

  ![Masks](image7)

- We can use masks for the averaging step:
  - (these masks for Loop’s scheme)

  ![Masks](image8)

- Where \( c_0(n) = \frac{n(1 - a(n))}{a(n)} \)
  - \( a(n) = \frac{5}{4} \) (3 + \(2\cos(2\pi/n))^2\)
  - \( c_i = 1 \)

- New vertex location is a weighted sum:
  - \( v'_0 = \sum_{i=0}^{n} c_i v_i / \sum_{i=0}^{n} c_i \)

- If you average only new vertices, surface interpolates the control points.

**Adding creases (sharp edges)**

- We can tag the mesh:
  - smooth/sharp edge
  - smooth/cusp/corner/dart vertex
  - The subdivision mask depends on the edge/vertex types.
**Trick for semi-sharp creases**

- Here's an example using Catmull-Clark surfaces of the kind found in Geri's Game
- Notice that the creases are not infinitely sharp
- The trick:
  - perform a few subdivisions using the sharp masks
  - change the masks back to smooth

**Rendering subdivision surfaces**

- The real surface is defined as the limit of the subdivision process
- Don't need to go quite that deep…
- At any level can calculate the limit
  - location
  - tangents => normal
- Typically a few levels of subdivision are enough

**Editing the surfaces**