Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Curves and Surfaces

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

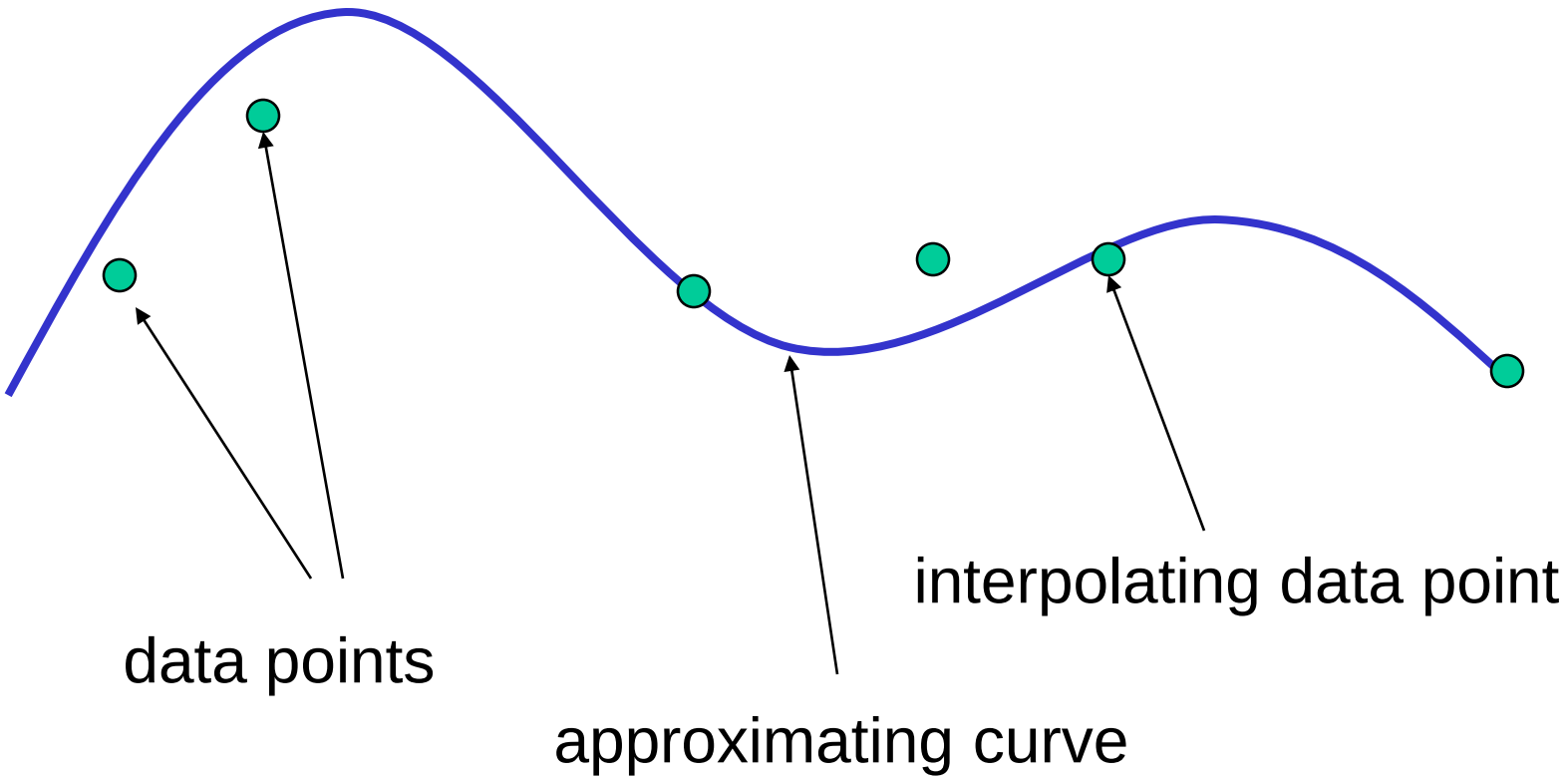
Objectives

- Introduce types of curves and surfaces
  - Explicit
  - Implicit
  - Parametric
  - Strengths and weaknesses
- Discuss Modeling and Approximations
  - Conditions
  - Stability

Escaping Flatland

- Until now we have worked with flat entities such as lines and flat polygons
  - Fit well with graphics hardware
  - Mathematically simple
- But the world is not composed of flat entities
  - Need curves and curved surfaces
  - May only have need at the application level
  - Implementation can render them approximately with flat primitives

Modeling with Curves



What Makes a Good Representation?

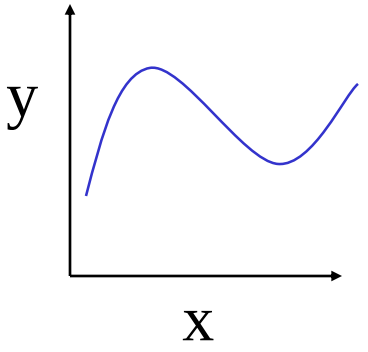
- There are many ways to represent curves and surfaces
- Want a representation that is
  - Stable
  - Smooth
  - Easy to evaluate
  - Must we interpolate or can we just come close to data?
  - Do we need derivatives?

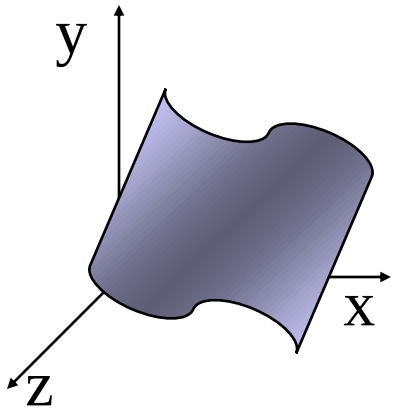
Explicit Representation

- Most familiar form of curve in 2D

$$y=f(x)$$

- Cannot represent all curves
  - Vertical lines
  - Circles
- Extension to 3D
  - $y=f(x)$, $z=g(x)$
  - The form $z = f(x,y)$ defines a surface





Implicit Representation

- Two dimensional curve(s)

$$g(x,y)=0$$

- Much more robust
  - All lines $ax+by+c=0$
  - Circles $x^2+y^2-r^2=0$
- Three dimensions $g(x,y,z)=0$ defines a surface
  - Intersect two surface to get a curve
- In general, we cannot solve for points that satisfy

Algebraic Surface

$$\sum_i \sum_j \sum_k x^i y^j z^k = 0$$

- Quadric surface $2 \geq i+j+k$
- At most 10 terms
- Can solve intersection with a ray by reducing problem to solving quadratic equation

The University of New Mexico

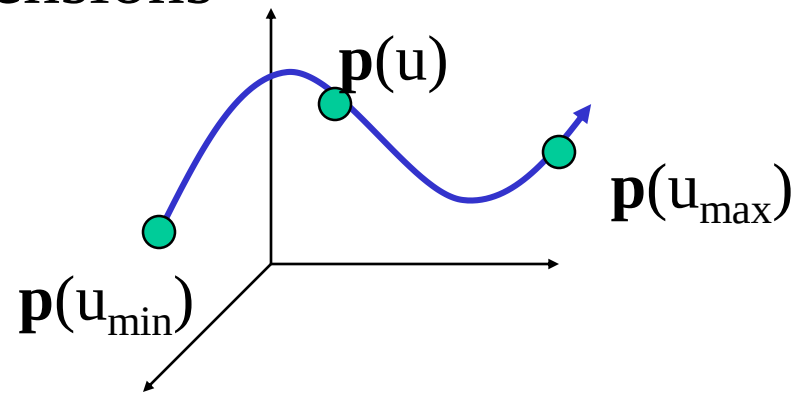
Parametric Curves

- Separate equation for each spatial variable

$$x=x(u)$$
$$y=y(u)$$
$$z=z(u)$$

$$\mathbf{p}(u)=[x(u), y(u), z(u)]^T$$

- For $u_{max} \geq u \geq u_{min}$ we trace out a curve in two or three dimensions



Selecting Functions

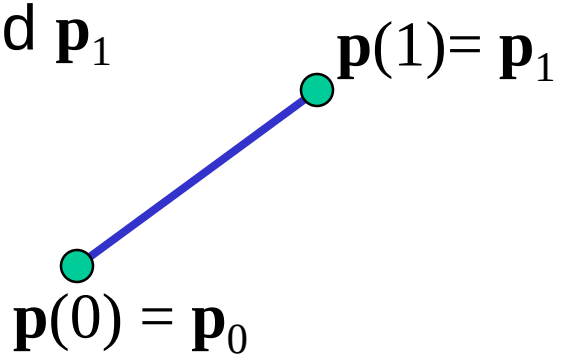
- Usually we can select “good” functions
  - not unique for a given spatial curve
  - Approximate or interpolate known data
  - Want functions which are easy to evaluate
  - Want functions which are easy to differentiate
    - Computation of normals
    - Connecting pieces (segments)
  - Want functions which are smooth

Parametric Lines

We can normalize u to be over the interval (0,1)

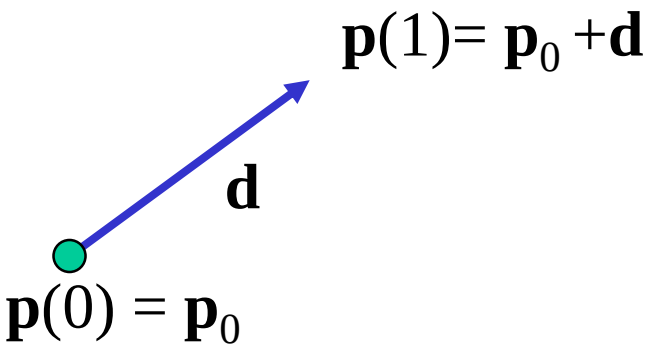
Line connecting two points $\mathbf{p}_0$ and $\mathbf{p}_1$

$$\mathbf{p}(u)=(1-u)\mathbf{p}_0+u\mathbf{p}_1$$



Ray from $\mathbf{p}_0$ in the direction $\mathbf{d}$

$$\mathbf{p}(u)=\mathbf{p}_0+u\mathbf{d}$$

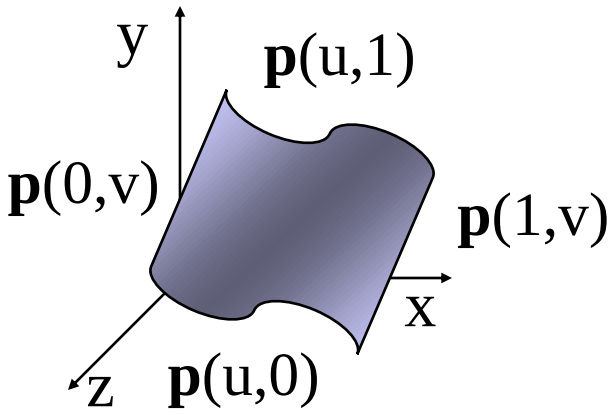


Parametric Surfaces

- Surfaces require 2 parameters

$$x = x(u,v)$$
$$y = y(u,v)$$
$$z = z(u,v)$$

$$\mathbf{p}(u,v) = [x(u,v),\ y(u,v),\ z(u,v)]^T$$



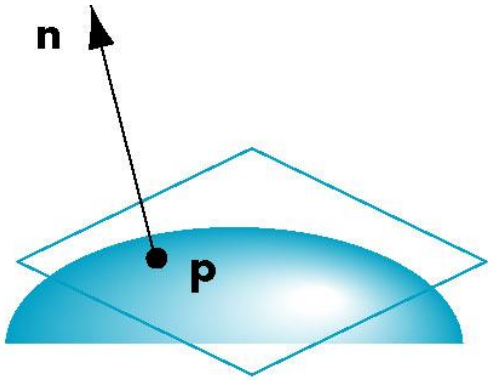
- Want same properties as curves:
  - Smoothness
  - Differentiability
  - Ease of evaluation

Normals

We can differentiate with respect to u and v to obtain the normal at any point **p**

$$\frac{\partial \mathbf{p}(u,v)}{\partial u} = \begin{bmatrix} \partial \mathrm{x}(u,v)/\partial u \\ \partial \mathrm{y}(u,v)/\partial u \\ \partial \mathrm{z}(u,v)/\partial u \end{bmatrix} \qquad \frac{\partial \mathbf{p}(u,v)}{\partial v} = \begin{bmatrix} \partial \mathrm{x}(u,v)/\partial v \\ \partial \mathrm{y}(u,v)/\partial v \\ \partial \mathrm{z}(u,v)/\partial v \end{bmatrix}$$

$$\mathbf{n} = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}$$



Parametric Planes

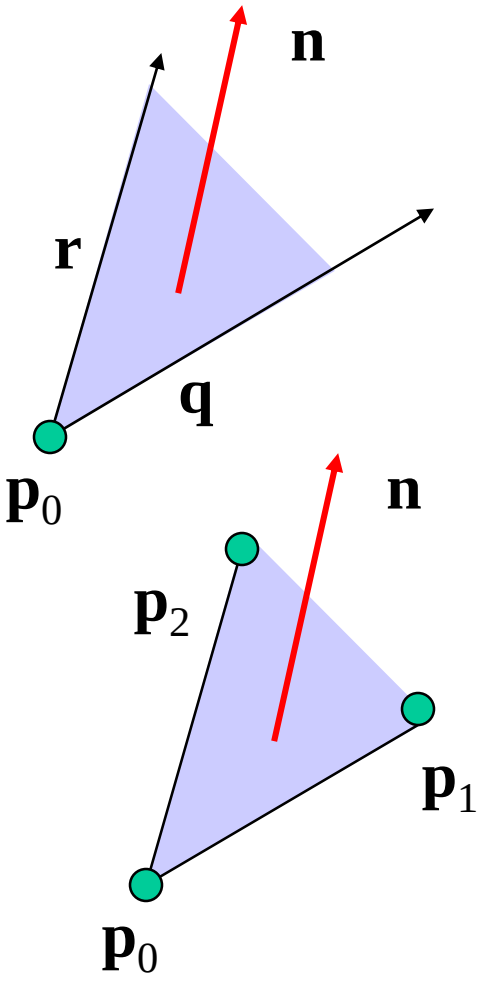
point-vector form

$$\mathbf{p}(u,v)=\mathbf{p}_0+u\mathbf{q}+v\mathbf{r}$$

$$\mathbf{n} = \mathbf{q} \times \mathbf{r}$$

three-point form

$$\mathbf{q} = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{r} = \mathbf{p}_2 - \mathbf{p}_0$$

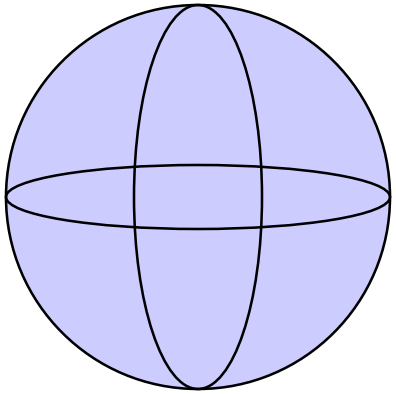


The University of New Mexico

Parametric Sphere

$$x(u,v) = r \cos\theta \sin\phi$$
$$y(u,v) = r \sin\theta \sin\phi$$
$$z(u,v) = r \cos\phi$$

$$360 \geq \theta \geq 0$$
$$180 \geq \phi \geq 0$$



θ constant: circles of constant longitude

ϕ constant: circles of constant latitude

differentiate to show $\mathbf{n} = \mathbf{p}$

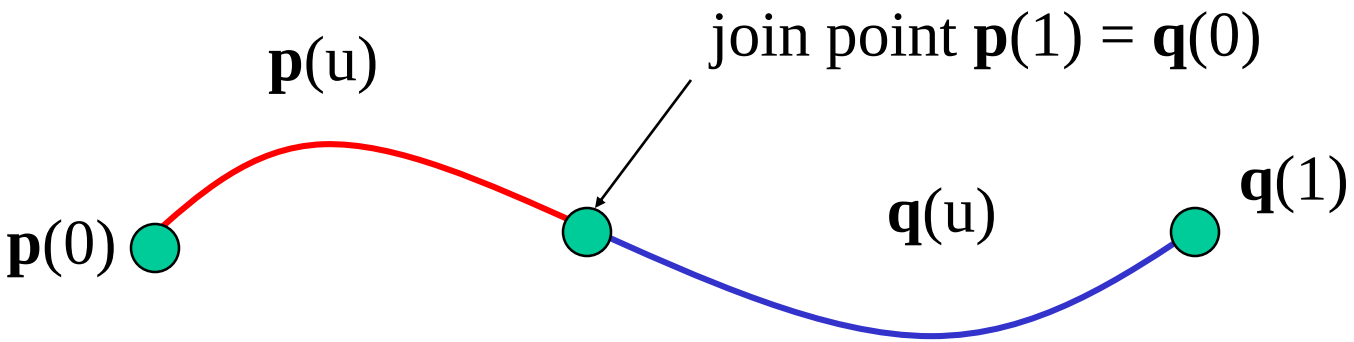
The University of New Mexico

Curve Segments

- After normalizing u, each curve is written

  $\mathbf{p}(u)=[x(u), y(u), z(u)]^T, \quad 1 \geq u \geq 0$

- In classical numerical methods, we design a single global curve
- In computer graphics and CAD, it is better to design small connected curve *segments*



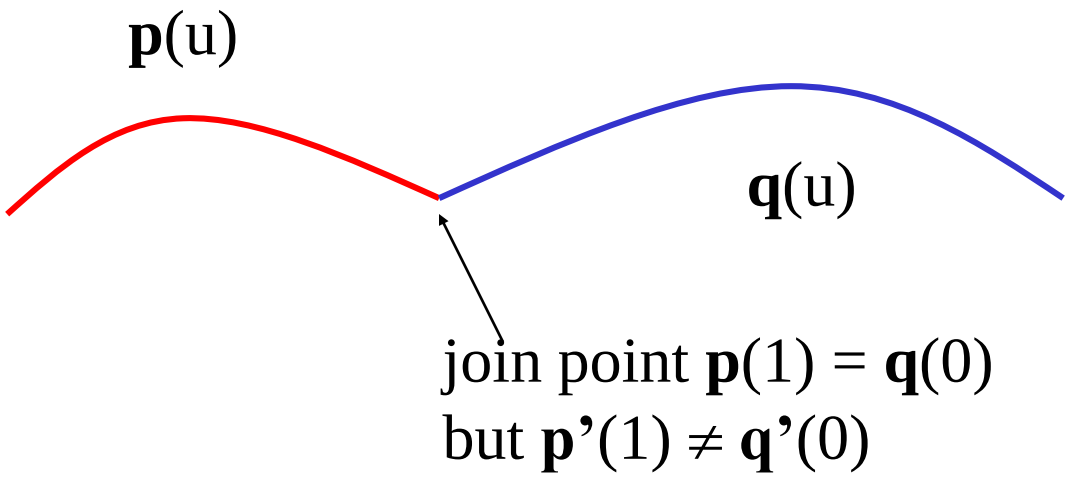
Parametric Polynomial Curves

$$x(u) = \sum_{i=0}^{N} c_{xi} u^i \quad y(u) = \sum_{j=0}^{M} c_{yj} u^j \quad z(u) = \sum_{k=0}^{L} c_{zk} u^k$$

- If N=M=K, we need to determine 3(N+1) coefficients
- Equivalently we need 3(N+1) independent conditions
- Noting that the curves for x, y and z are independent, we can define each independently in an identical manner
  - We will use the form $p(u) = \sum_{k=0}^{L} c_k u^k$ where p can be any of x, y, z

Why Polynomials

- Easy to evaluate
- Continuous and differentiable everywhere
  - Must worry about continuity at join points including continuity of derivatives



Cubic Parametric Polynomials

- N=M=L=3, gives balance between ease of evaluation and flexibility in design

$$\mathrm{p}(u) = \sum_{k=0}^{3} c_k u^k$$

- Four coefficients to determine for each of x, y and z
- Seek four independent conditions for various values of u resulting in 4 equations in 4 unknowns for each of x, y and z
  - Conditions are a mixture of continuity requirements at the join points and conditions for fitting the data

Cubic Polynomial Surfaces

$$\mathbf{p}(u,v)=[x(u,v),\ y(u,v),\ z(u,v)]^T$$

where

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} c_{ij}\, u^i\, v^j$$

p is any of x, y or z

Need 48 coefficients (3 independent sets of 16) to determine a surface patch

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Designing Parametric Cubic Curves

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Objectives

- Introduce the types of curves
  - Interpolating
  - Hermite
  - Bezier
  - B-spline
- Analyze their performance

Matrix-Vector Form

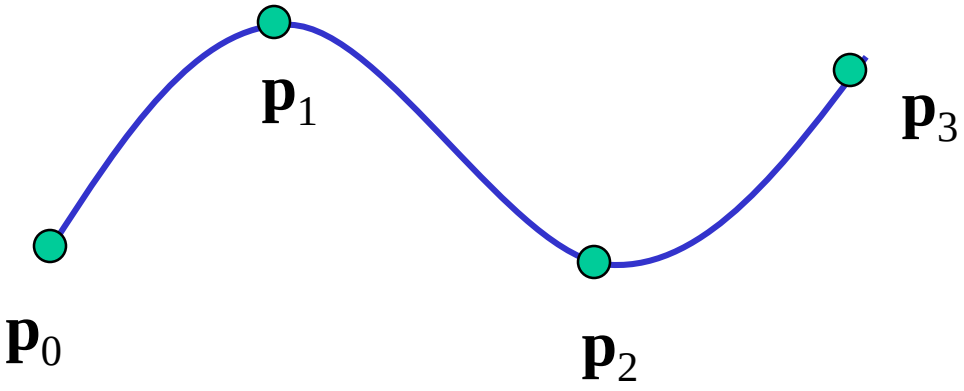
$$p(u) = \sum_{k=0}^{3} c_k u^k$$

define $\mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$ $\mathbf{u} = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}$

then $p(u) = \mathbf{u}^T \mathbf{c} = \mathbf{c}^T \mathbf{u}$

The University of New Mexico

Interpolating Curve



Given four data (control) points $\mathbf{p}_0$, $\mathbf{p}_1$,$\mathbf{p}_2$, $\mathbf{p}_3$
determine cubic $\mathbf{p}(u)$ which passes through them

Must find $\mathbf{c}_0$,$\mathbf{c}_1$,$\mathbf{c}_2$, $\mathbf{c}_3$

Interpolation Equations

apply the interpolating conditions at u=0, 1/3, 2/3, 1

$$p_0=p(0)=c_0$$
$$p_1=p(1/3)=c_0+(1/3)c_1+(1/3)^2c_2+(1/3)^3c_2$$
$$p_2=p(2/3)=c_0+(2/3)c_1+(2/3)^2c_2+(2/3)^3c_2$$
$$p_3=p(1)=c_0+c_1+c_2+c_2$$

or in matrix form with $\mathbf{p} = [p_0\ p_1\ p_2\ p_3]^T$

$$\mathbf{p=Ac} \qquad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \left(\frac{1}{3}\right) & \left(\frac{1}{3}\right)^2 & \left(\frac{1}{3}\right)^3 \\ 1 & \left(\frac{2}{3}\right) & \left(\frac{2}{3}\right)^2 & \left(\frac{2}{3}\right)^3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Interpolation Matrix

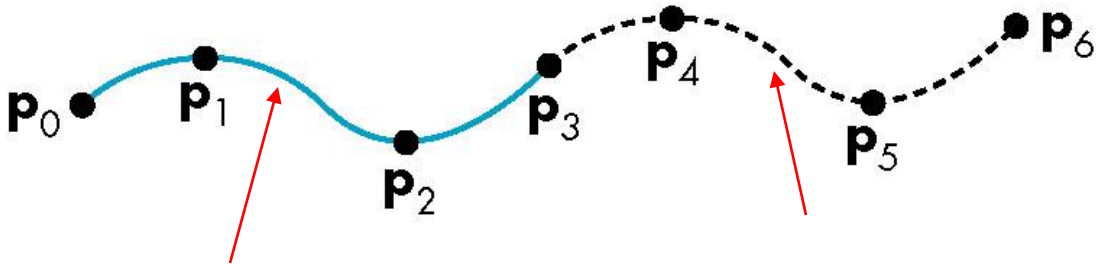
Solving for $\mathbf{c}$ we find the *interpolation matrix*

$$\mathbf{M}_I = \mathbf{A}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix}$$

$$\mathbf{c} = \mathbf{M}_I \mathbf{p}$$

Note that $\mathbf{M}_I$ does not depend on input data and can be used for each segment in x, y, and z

Interpolating Multiple Segments



use $\mathbf{p} = [p_0\ p_1\ p_2\ p_3]^T$

use $\mathbf{p} = [p_3\ p_4\ p_5\ p_6]^T$

Get continuity at join points but not continuity of derivatives

Blending Functions

Rewriting the equation for p(u)

$$p(u)=\mathbf{u}^T\mathbf{c}=\mathbf{u}^T\mathbf{M}_I\mathbf{p} = \mathbf{b}(u)^T\mathbf{p}$$

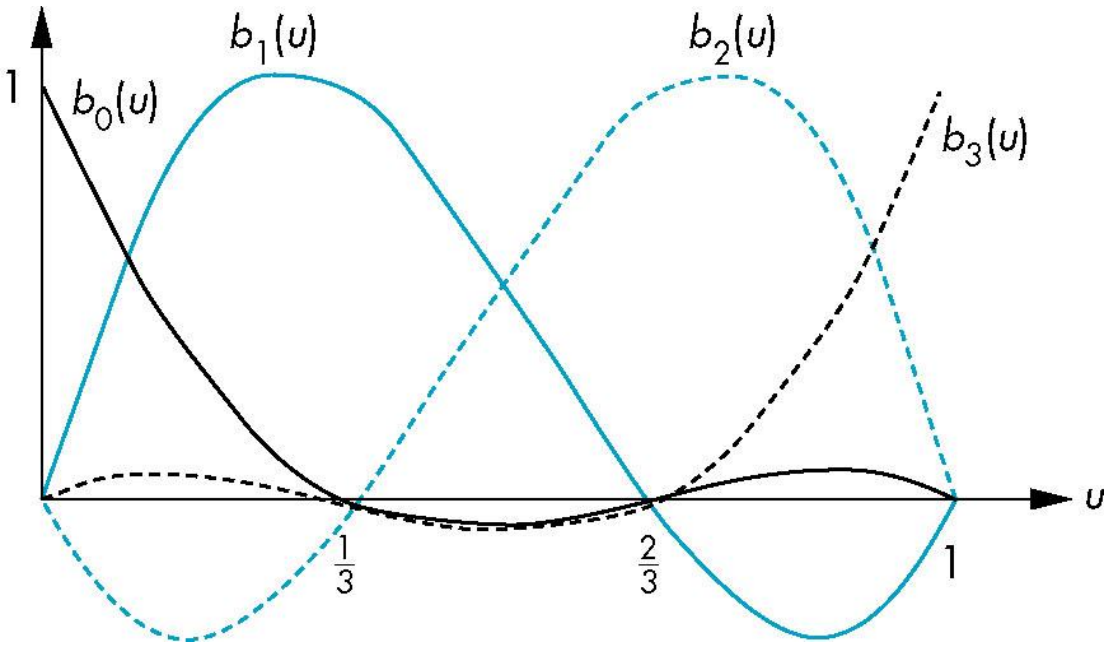
where $b(u) = [b_0(u)\ b_1(u)\ b_2(u)\ b_3(u)]^T$ is an array of *blending polynomials* such that

$$p(u) = b_0(u)p_0 + b_1(u)p_1 + b_2(u)p_2 + b_3(u)p_3$$

$$b_0(u) = -4.5(u-1/3)(u-2/3)(u-1)$$
$$b_1(u) = 13.5u\ (u-2/3)(u-1)$$
$$b_2(u) = -13.5u\ (u-1/3)(u-1)$$
$$b_3(u) = 4.5u\ (u-1/3)(u-2/3)$$

Blending Functions

- These functions are not smooth
  - Hence the interpolation polynomial is not smooth



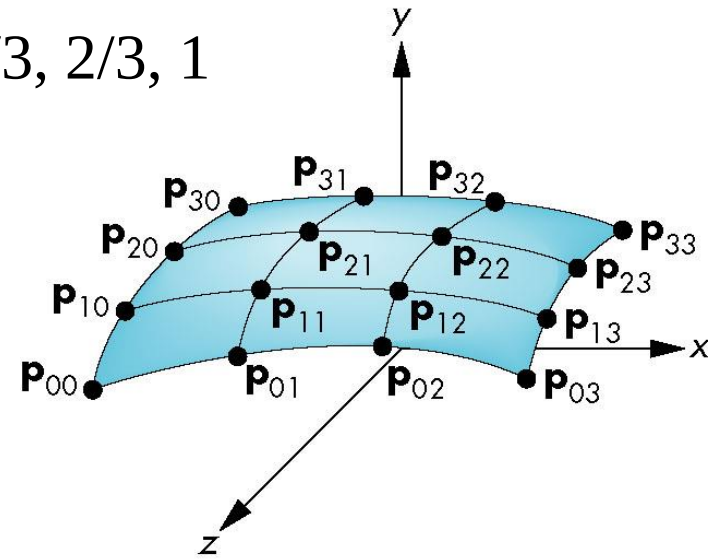
The University of New Mexico

Interpolating Patch

$$p(u,v) = \sum_{i=o}^{3} \sum_{j=0}^{3} c_{ij} u^i v^j$$

Need 16 conditions to determine the 16 coefficients c_{ij}

Choose at u,v = 0, 1/3, 2/3, 1



Matrix Form

Define $\mathbf{v} = [1 \; v \; v^2 \; v^3]^T$

$\mathbf{C} = [c_{ij}] \qquad \mathbf{P} = [p_{ij}]$

$p(u,v) = \mathbf{u}^T C \mathbf{v}$

If we observe that for constant u (v), we obtain interpolating curve in v (u), we can show

$\mathbf{C} = M_I P M_I$

$p(u,v) = u^T M_I P M_I^T v$

Blending Patches

$$p(u,v) = \sum_{i=o}^{3} \sum_{j=0}^{3} b_i(u)\, b_j(v)\, p_{ij}$$

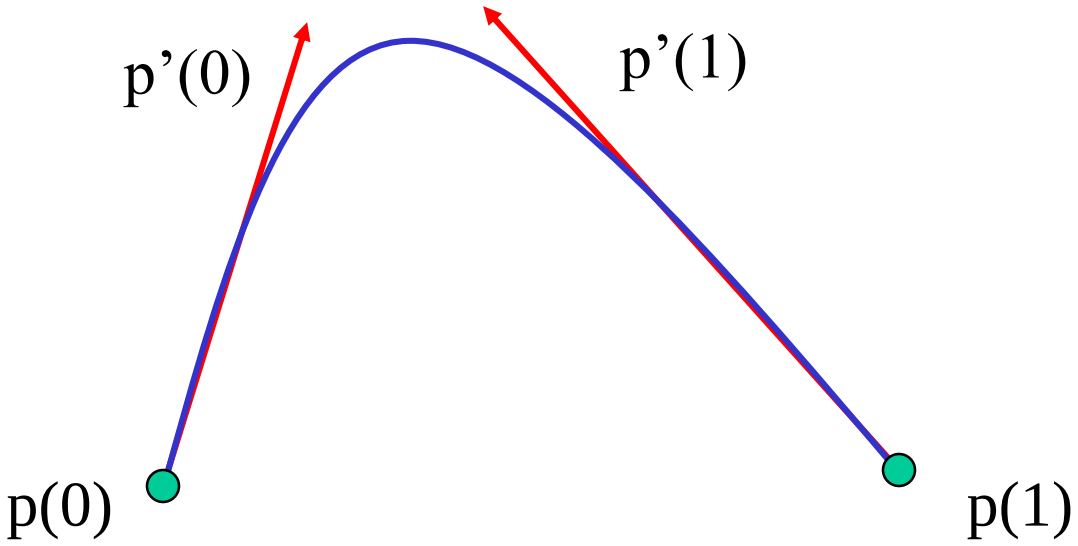
Each $b_i(u)b_j(v)$ is a blending patch

Shows that we can build and analyze surfaces from our knowledge of curves

Other Types of Curves and Surfaces

- How can we get around the limitations of the interpolating form
  - Lack of smoothness
  - Discontinuous derivatives at join points
- We have four conditions (for cubics) that we can apply to each segment
  - Use them other than for interpolation
  - Need only come close to the data

Hermite Form



Use two interpolating conditions and two derivative conditions per segment

Ensures continuity and first derivative continuity between segments

Equations

Interpolating conditions are the same at ends

$$p(0) = p_0 = c_0$$
$$p(1) = p_3 = c_0+c_1+c_2+c_3$$

Differentiating we find $p'(u) = c_1+2uc_2+3u^2c_3$

Evaluating at end points

$$p'(0) = p'_0 = c_1$$
$$p'(1) = p'_3 = c_1+2c_2+3c_3$$

Matrix Form

$$\mathbf{q} = \begin{bmatrix} p_0 \\ p_3 \\ p'_0 \\ p'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \mathbf{c}$$

Solving, we find $\mathbf{c}=\mathbf{M}_H\mathbf{q}$ where $\mathbf{M}_H$ is the Hermite matrix

$$\mathbf{M}_H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix}$$

Blending Polynomials

$$p(u) = \mathbf{b}(u)^T \mathbf{q}$$

$$\mathbf{b}(u) = \begin{bmatrix} 2u^3 - 3u^2 + 1 \\ -2u^3 + 3u^2 \\ u^3 - 2u^2 + u \\ u^3 - u^2 \end{bmatrix}$$

Although these functions are smooth, the Hermite form is not used directly in Computer Graphics and CAD because we usually have control points but not derivatives

However, the Hermite form is the basis of the Bezier form

Parametric and Geometric Continuity

- We can require the derivatives of x, y,and z to each be continuous at join points (*parametric continuity*)
- Alternately, we can only require that the tangents of the resulting curve be continuous (*geometry continuity*)
- The latter gives more flexibility as we have need satisfy only two conditions rather than three at each join point

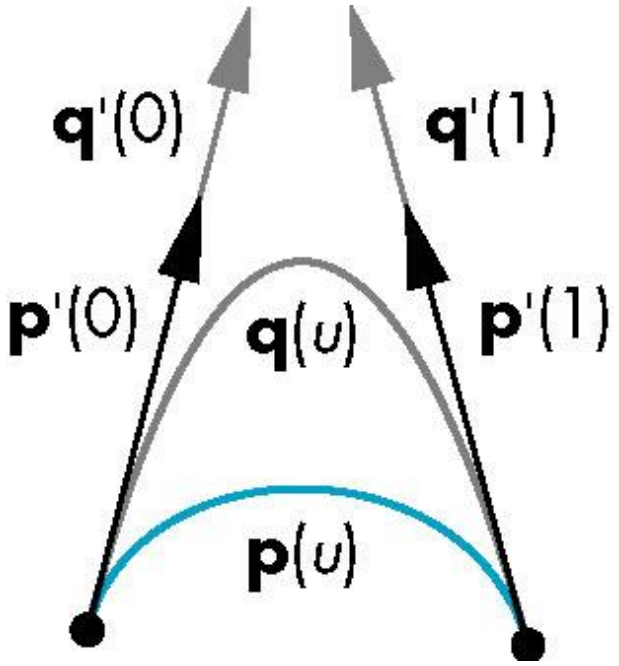
The University of New Mexico

Example

- Here the p and q have the same tangents at the ends of the segment but different derivatives
- Generate different

  Hermite curves
- This techniques is used

in drawing applications



Higher Dimensional Approximations

- The techniques for both interpolating and Hermite curves can be used with higher dimensional parametric polynomials
- For interpolating form, the resulting matrix becomes increasingly more ill-conditioned and the resulting curves less smooth and more prone to numerical errors
- In both cases, there is more work in rendering the resulting polynomial curves and surfaces

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Bezier and Spline Curves and Surfaces

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

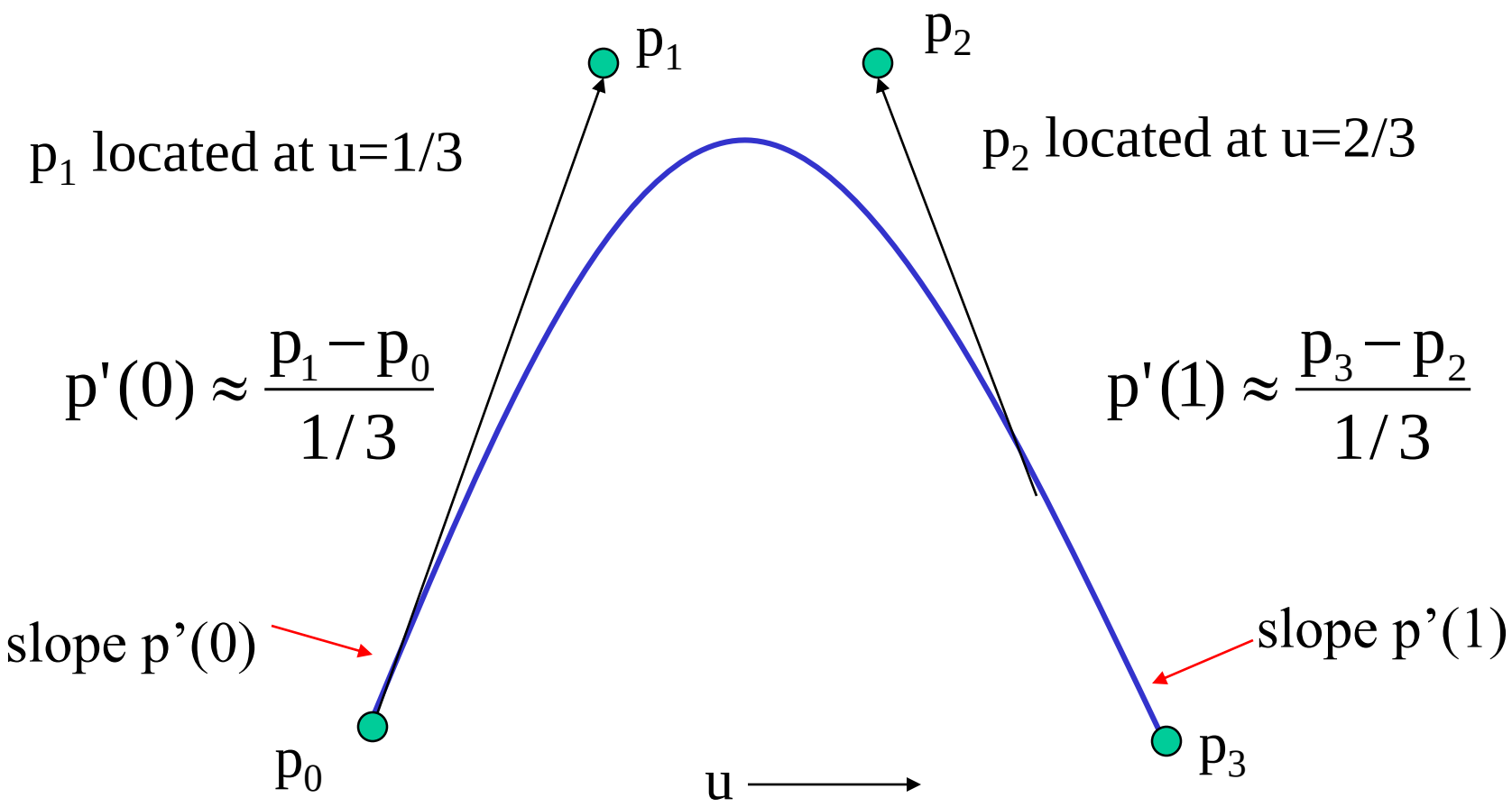
Objectives

- Introduce the Bezier curves and surfaces
- Derive the required matrices
- Introduce the B-spline and compare it to the standard cubic Bezier

Bezier's Idea

- In graphics and CAD, we do not usually have derivative data
- Bezier suggested using the same 4 data points as with the cubic interpolating curve to approximate the derivatives in the Hermite form

Approximating Derivatives



Equations

Interpolating conditions are the same

$$p(0) = p_0 = c_0$$
$$p(1) = p_3 = c_0+c_1+c_2+c_3$$

Approximating derivative conditions

$$p'(0) = 3(p_1 - p_0) = c_0$$
$$p'(1) = 3(p_3 - p_2) = c_1+2c_2+3c_3$$

Solve four linear equations for $\mathbf{c}=\mathbf{M}_B\mathbf{p}$

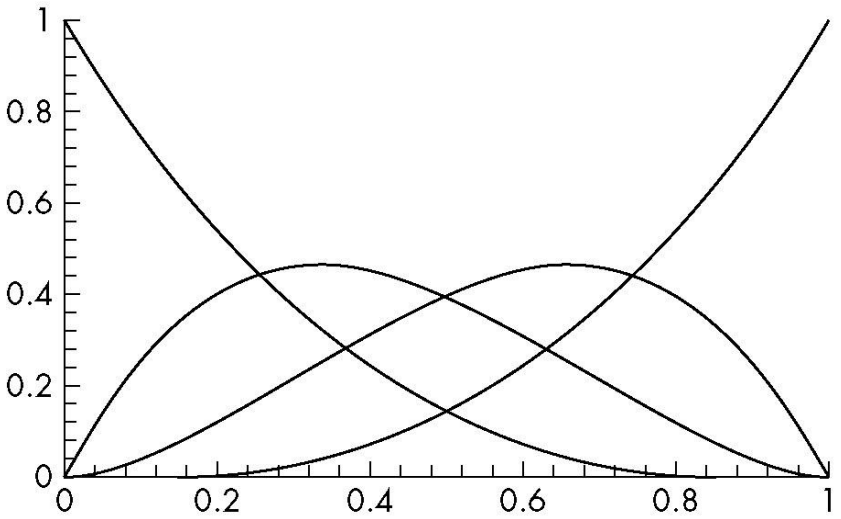
Bezier Matrix

$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

$$p(u) = \mathbf{u}^T\mathbf{M}_B\mathbf{p} = \mathbf{b}(u)^T\mathbf{p}$$

Blending Functions

$$\mathbf{b}(u) = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}$$



Note that all zeros are at 0 and 1 which forces the functions to be smooth over (0,1)

Bernstein Polynomials

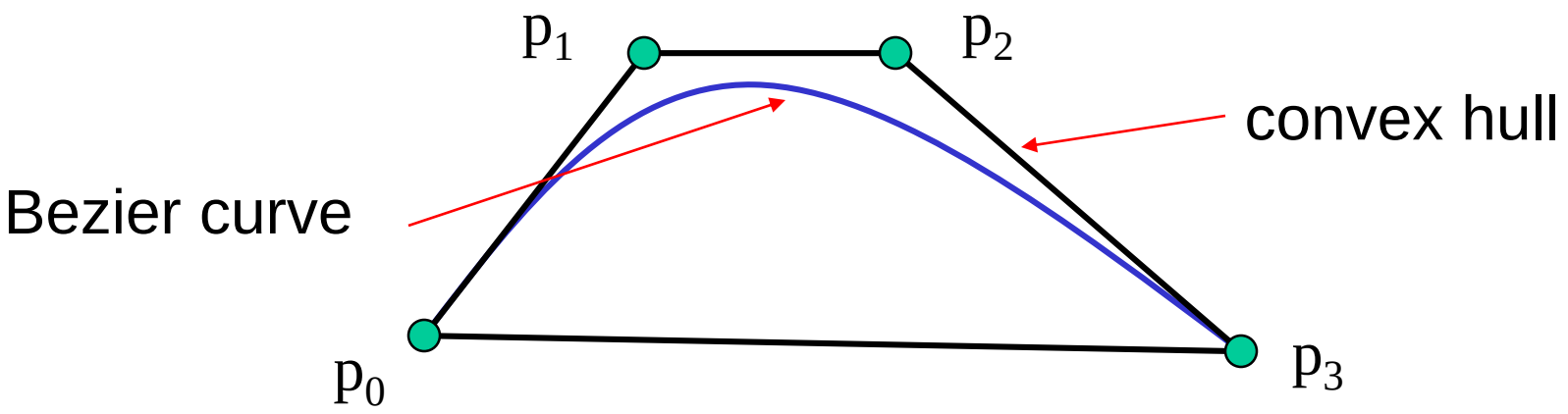
- The blending functions are a special case of the Bernstein polynomials

$$b_{\mathrm{kd}}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k}$$

- These polynomials give the blending polynomials for any degree Bezier form
  - All zeros at 0 and 1
  - For any degree they all sum to 1
  - They are all between 0 and 1 inside (0,1)

Convex Hull Property

- The properties of the Bernstein polynomials ensure that all Bezier curves lie in the convex hull of their control points
- Hence, even though we do not interpolate all the data, we cannot be too far away



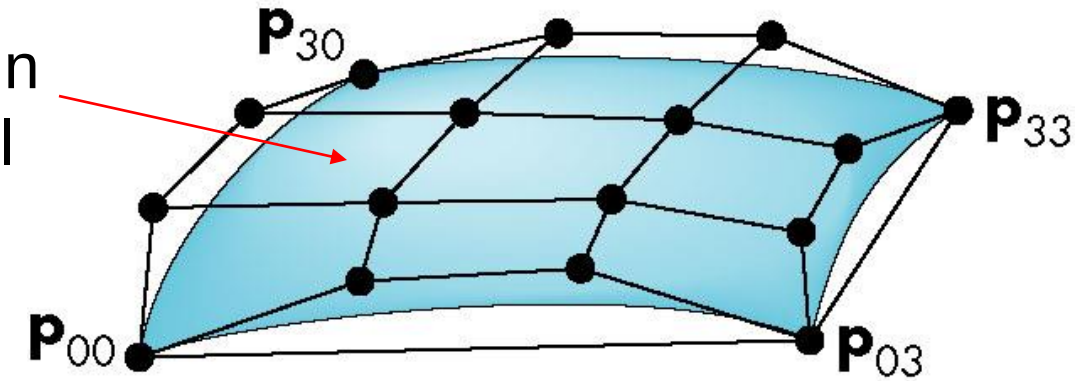
The University of New Mexico

Bezier Patches

Using same data array $\mathbf{P}=[p_{ij}]$ as with interpolating form

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u)\, b_j(v)\, p_{ij} = u^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B^T v$$

Patch lies in convex hull



The University of New Mexico

Analysis

- Although the Bezier form is much better than the interpolating form, we have the derivatives are not continuous at join points
- Can we do better?
  - Go to higher order Bezier
    - More work
    - Derivative continuity still only approximate
    - Supported by fixed function OpenGL
  - Apply different conditions
    - Tricky without letting order increase

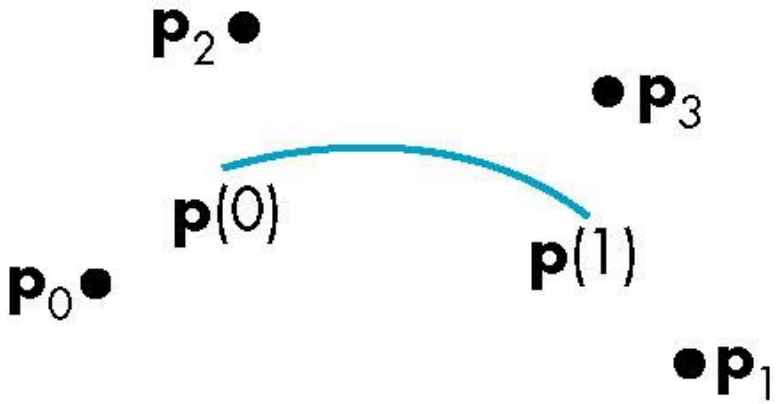
B-Splines

- Basis splines: use the data at $\mathbf{p}=[p_{i-2}\ p_{i-1}\ p_i\ p_{i-1}]^T$ to define curve only between p_{i-1} and p_i
- Allows us to apply more continuity conditions to each segment
- For cubics, we can have continuity of function, first and second derivatives at join points
- Cost is 3 times as much work for curves
  - Add one new point each time rather than three
- For surfaces, we do 9 times as much work

Cubic B-spline

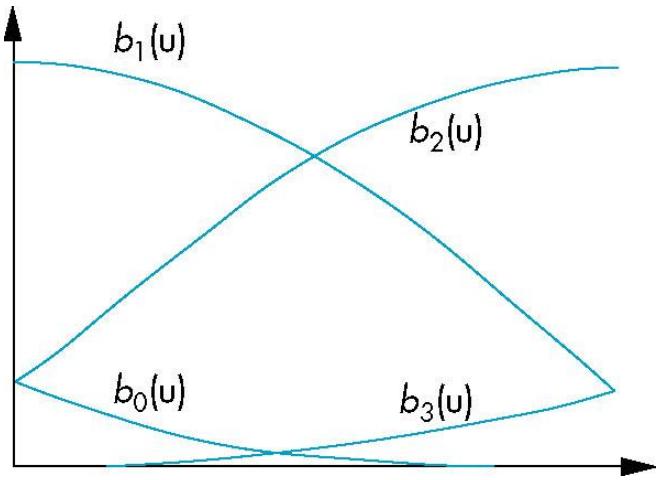
$$p(u) = \mathbf{u}^T \mathbf{M}_S \mathbf{p} = \mathbf{b}(u)^T \mathbf{p}$$

$$\mathbf{M}_S = \begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

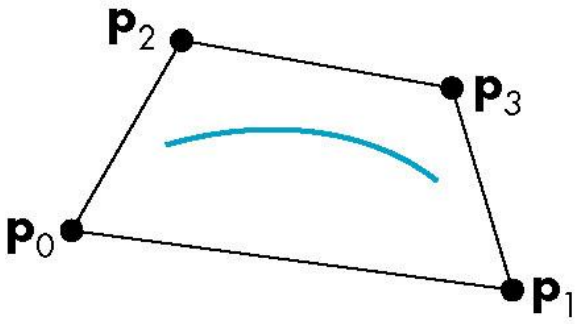


Blending Functions

$$\mathbf{b}(u) = \frac{1}{6}\begin{bmatrix} (1-u)^3 \\ 4-6u^2+3u^3 \\ 1+3u+3u^2-3u^2 \\ u^3 \end{bmatrix}$$



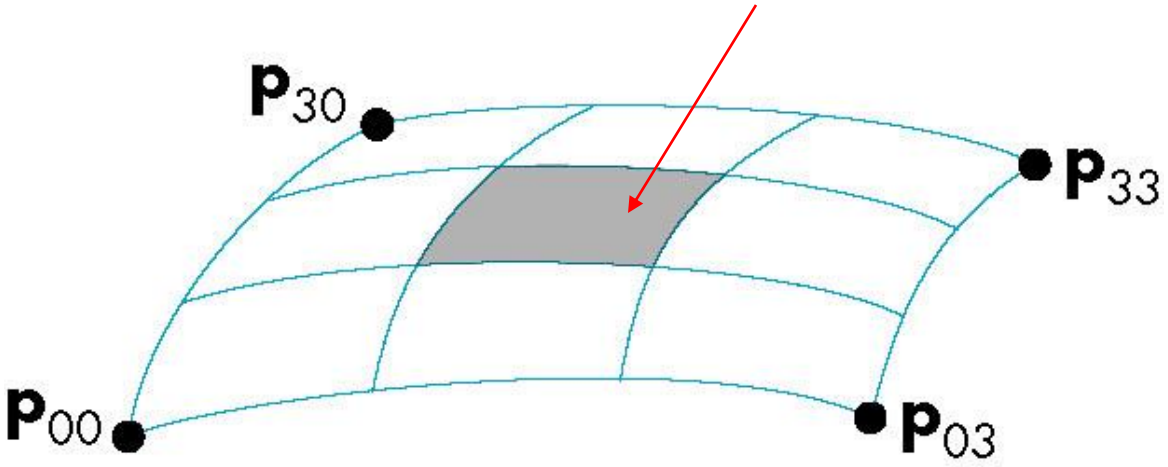
convex hull property



B-Spline Patches

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u)\, b_j(v)\, p_{ij} = u^T \mathbf{M}_S \mathbf{P} \mathbf{M}_S^T v$$

defined over only 1/9 of region



Splines and Basis

- If we examine the cubic B-spline from the perspective of each control (data) point, each interior point contributes (through the blending functions) to four segments
- We can rewrite p(u) in terms of the data points as

$$p(u) = \sum B_i(u)\, p_i$$

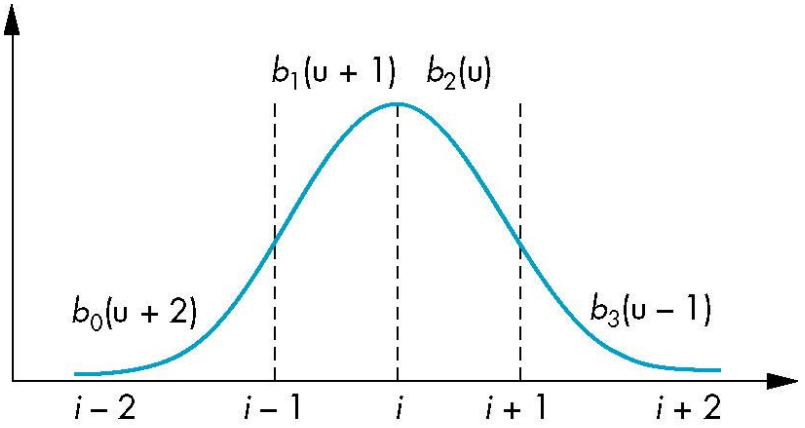
defining the basis functions $\{B_i(u)\}$

The University of New Mexico

Basis Functions

In terms of the blending polynomials

$$B_i(u) = \begin{cases} 0 & u < i-2 \\ b_0(u+2) & i-2 \le u < i-1 \\ b_1(u+1) & i-1 \le u < i \\ b_2(u) & i \le u < i+1 \\ b_3(u-1) & i+1 \le u < i+2 \\ 0 & u \ge i+2 \end{cases}$$



Generalizing Splines

- We can extend to splines of any degree
- Data and conditions to not have to given at equally spaced values (the *knots*)
  - Nonuniform and uniform splines
  - Can have repeated knots
    - Can force spline to interpolate points
- Cox-deBoor recursion gives method of evaluation

NURBS

- Nonuniform Rational B-Spline curves and surfaces add a fourth variable w to x,y,z
  - Can interpret as weight to give more importance to some control data
  - Can also interpret as moving to homogeneous coordinate
- Requires a perspective division
  - NURBS act correctly for perspective viewing
- Quadrics are a special case of NURBS

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Rendering Curves and Surfaces

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Objectives

- Introduce methods to draw curves
  - Approximate with lines
  - Finite Differences
- Derive the recursive method for evaluation of Bezier curves and surfaces
- Learn how to convert all polynomial data to data for Bezier polynomials

Evaluating Polynomials

- Simplest method to render a polynomial curve is to evaluate the polynomial at many points and form an approximating polyline
- For surfaces we can form an approximating mesh of triangles or quadrilaterals
- Use Horner's method to evaluate polynomials

$$p(u)=c_0+u(c_1+u(c_2+uc_3))$$

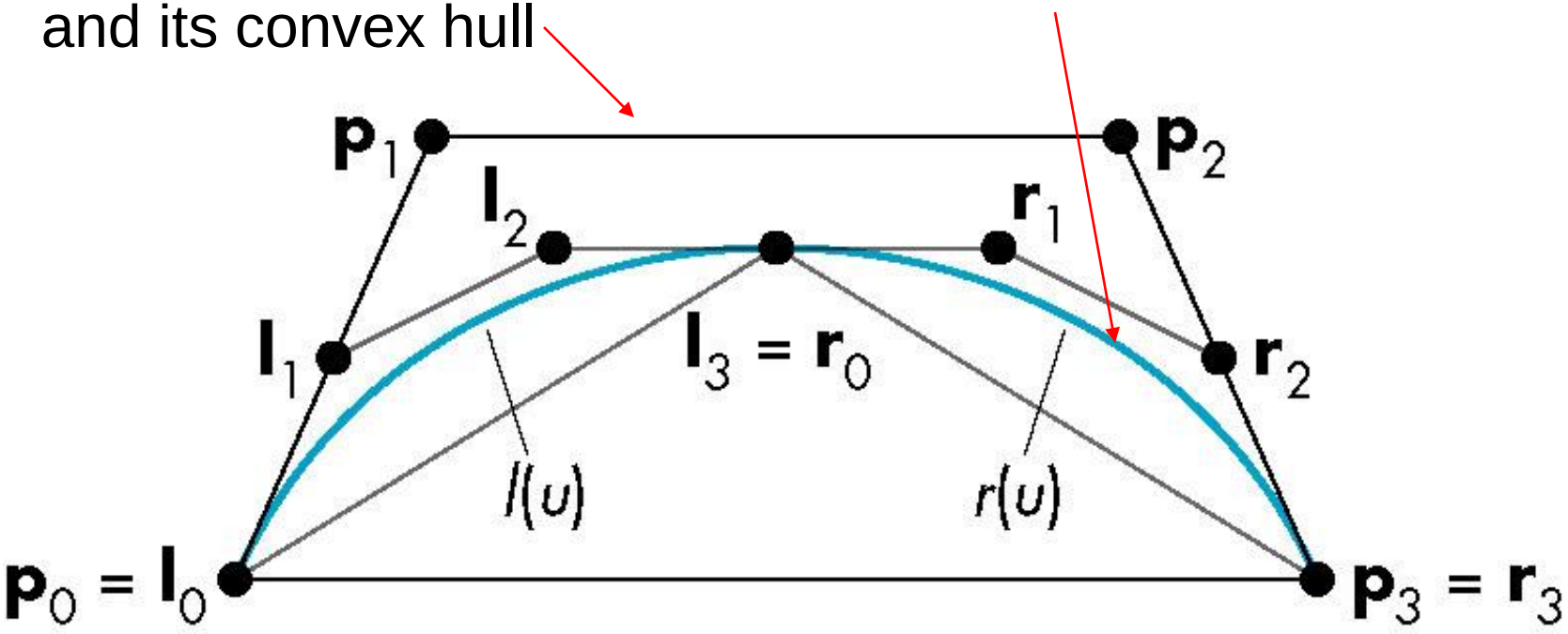
- 3 multiplications/evaluation for cubic

deCasteljau Recursion

- We can use the convex hull property of Bezier curves to obtain an efficient recursive method that does not require any function evaluations
  - Uses only the values at the control points
- Based on the idea that “any polynomial and any part of a polynomial is a Bezier polynomial for properly chosen control data”

Splitting a Cubic Bezier

p_0, p_1, p_2, p_3 determine a cubic Bezier polynomial and its convex hull

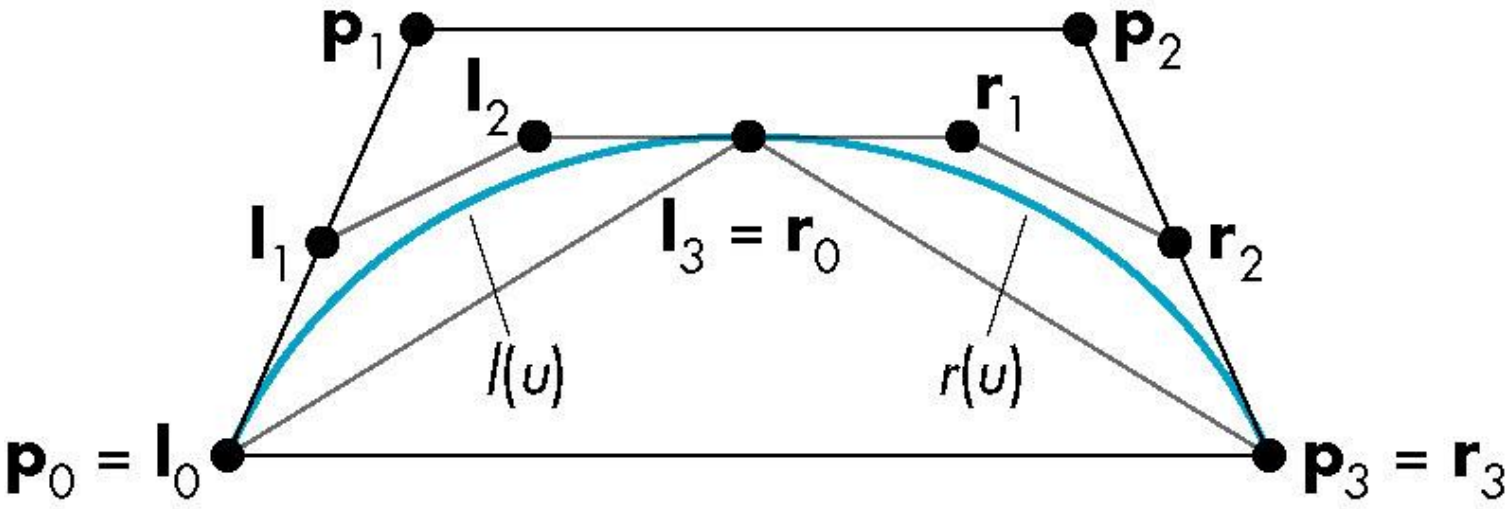


Consider left half $l(u)$ and right half $r(u)$

The University of New Mexico

l(u) and r(u)

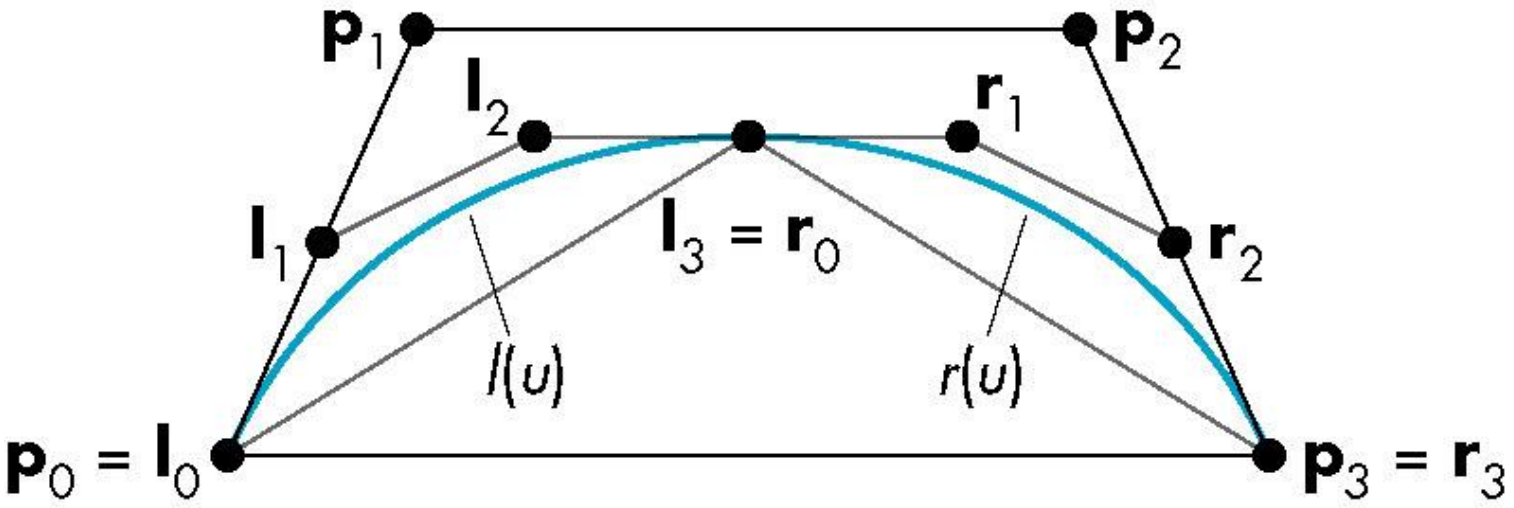
Since l(u) and r(u) are Bezier curves, we should be able to find two sets of control points $\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ that determine them



Convex Hulls

$\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$each have a convex hull that that is closer to p(u) than the convex hull of $\{p_0, p_1, p_2, p_3\}$ This is known as the *variation diminishing property*.

The polyline from l_0 to l_3 $(= r_0)$ to r_3 is an approximation to p(u). Repeating recursively we get better approximations.



Equations

Start with Bezier equations $p(u)=\mathbf{u}^T M_B \mathbf{p}$

$l(u)$ must interpolate $p(0)$ and $p(1/2)$

$$l(0) = l_0 = p_0$$
$$l(1) = l_3 = p(1/2) = 1/8(p_0 + 3 p_1 + 3 p_2 + p_3)$$

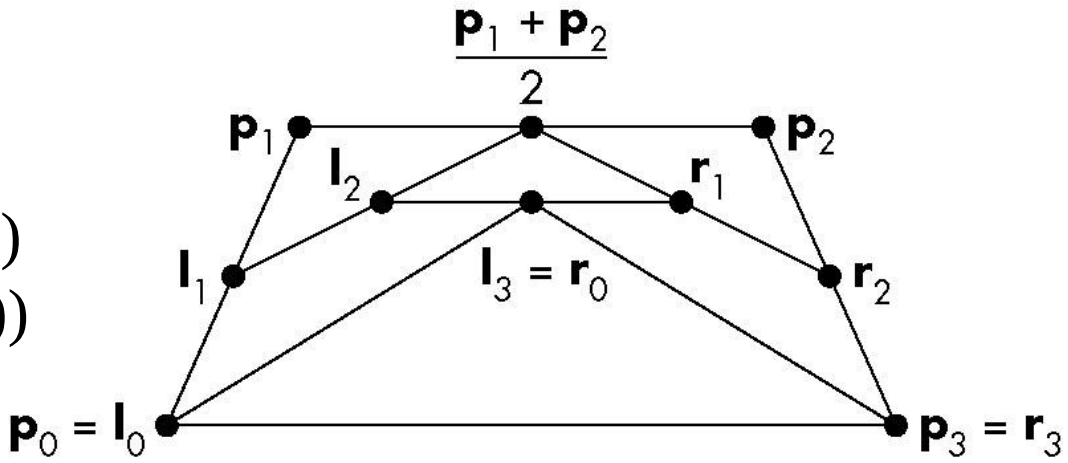
Matching slopes, taking into account that $l(u)$ and $r(u)$ only go over half the distance as $p(u)$

$$l'(0) = 3(l_1 - l_0) = p'(0) = 3/2(p_1 - p_0)$$
$$l'(1) = 3(l_3 - l_2) = p'(1/2) = 3/8(- p_0 - p_1 + p_2 + p_3)$$

Symmetric equations hold for $r(u)$

Efficient Form

$l_0 = p_0$
$r_3 = p_3$
$l_1 = ½(p_0 + p_1)$
$r_1 = ½(p_2 + p_3)$
$l_2 = ½(l_1 + ½(p_1 + p_2))$
$r_1 = ½(r_2 + ½(p_1 + p_2))$
$l_3 = r_0 = ½(l_2 + r_1)$



Requires only shifts and adds!

Every Curve is a Bezier Curve

- We can render a given polynomial using the recursive method if we find control points for its representation as a Bezier curve
- Suppose that $p(u)$ is given as an interpolating curve with control points $\mathbf{q}$

$$p(u)=\mathbf{u}^T\mathbf{M}_I\mathbf{q}$$

- There exist Bezier control points $\mathbf{p}$ such that

$$p(u)=\mathbf{u}^T\mathbf{M}_B\mathbf{p}$$

- Equating and solving, we find $\mathbf{p}=\mathbf{M}_B^{-1}\mathbf{M}_I$

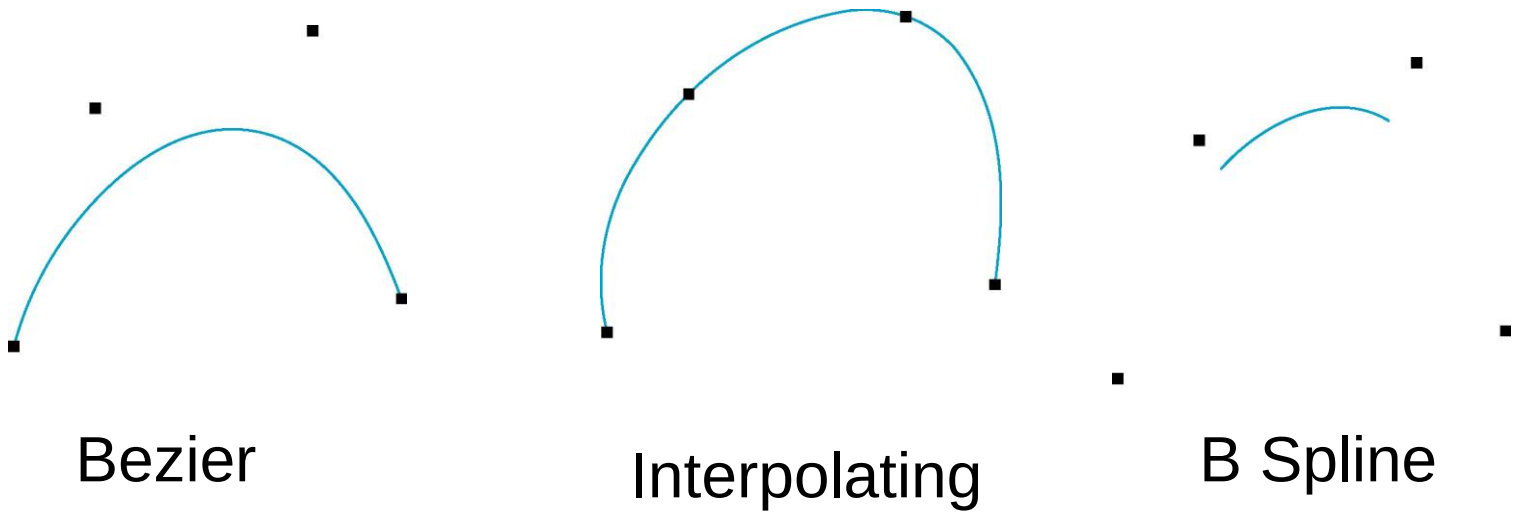
Matrices

Interpolating to Bezier $\mathbf{M}_B^{-1}\mathbf{M}_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{5}{6} & 3 & -\frac{3}{2} & \frac{1}{3} \\ \frac{1}{3} & -\frac{3}{2} & 3 & -\frac{5}{6} \\ 0 & 0 & 0 & 1 \end{bmatrix}$

B-Spline to Bezier $\mathbf{M}_B^{-1}\mathbf{M}_S = \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}$

Example

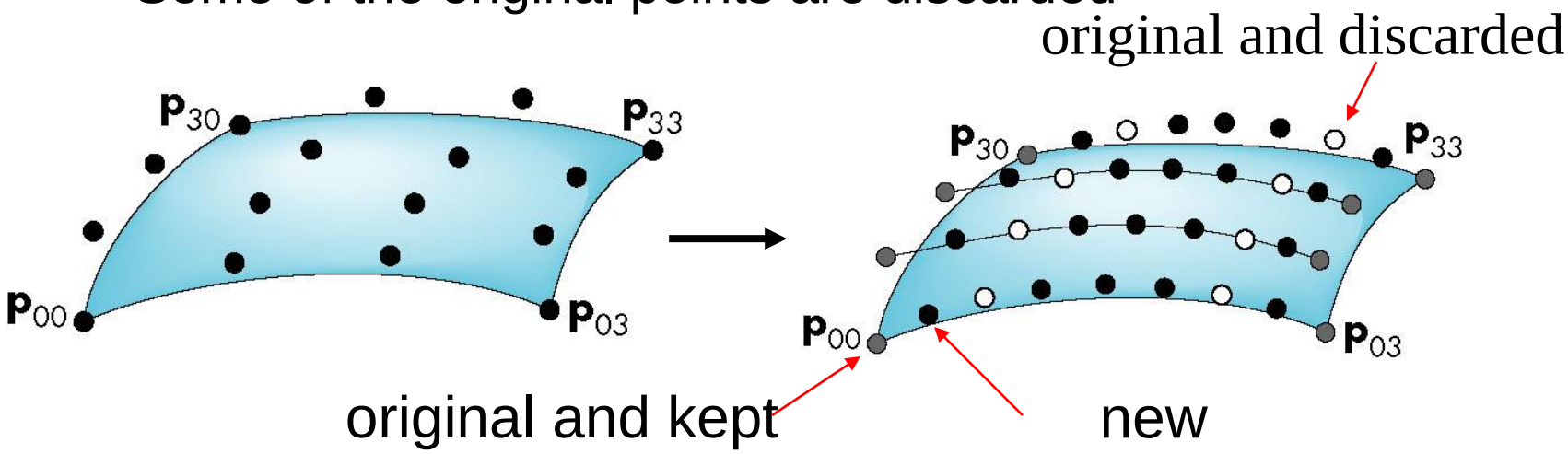
These three curves were all generated from the same original data using Bezier recursion by converting all control point data to Bezier control points



The University of New Mexico

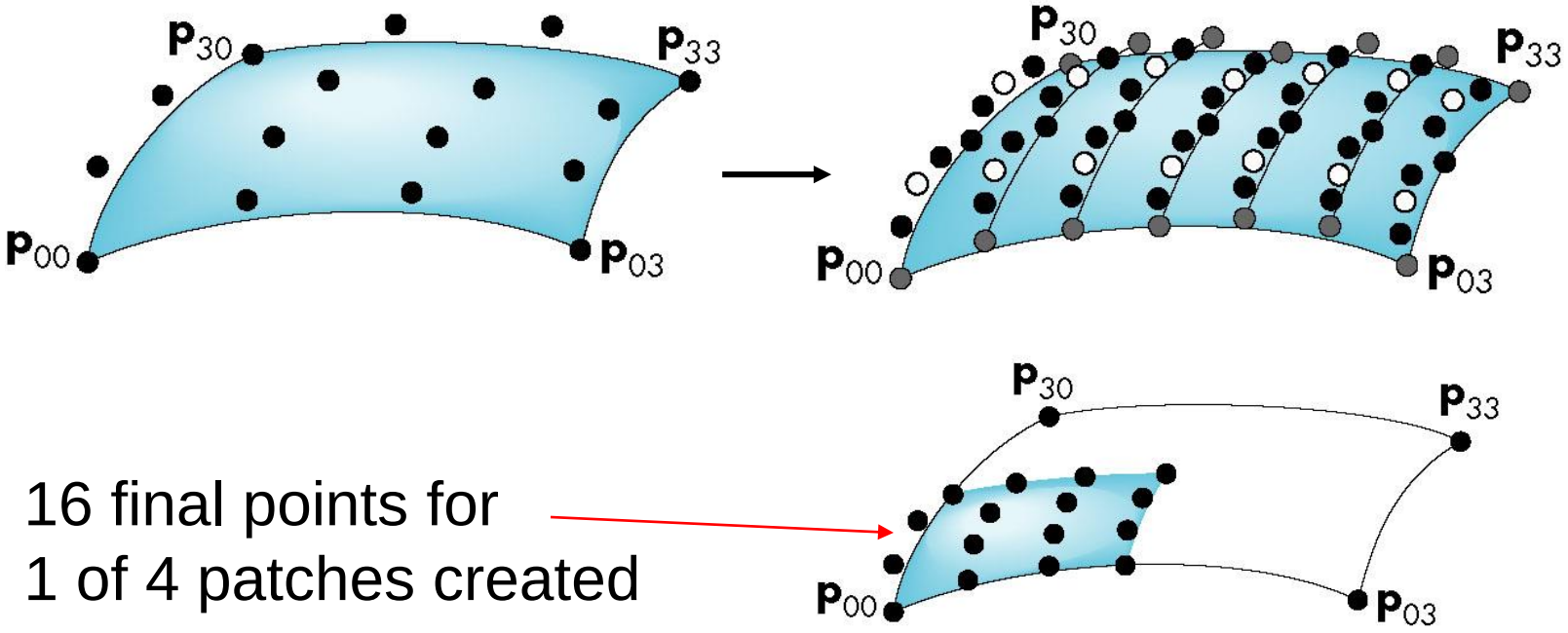
Surfaces

- Can apply the recursive method to surfaces if we recall that for a Bezier patch curves of constant u (or v) are Bezier curves in u (or v)
- First subdivide in u
  - Process creates new points
  - Some of the original points are discarded



Second Subdivision

- New points created by subdivision
- Old points discarded after subdivision
- Old points retained after subdivision



16 final points for
1 of 4 patches created

Normals

- For rendering we need the normals if we want to shade
  - Can compute from parametric equations

$$\mathbf{n} = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}$$

  - Can use vertices of corner points to determine
  - OpenGL can compute automatically

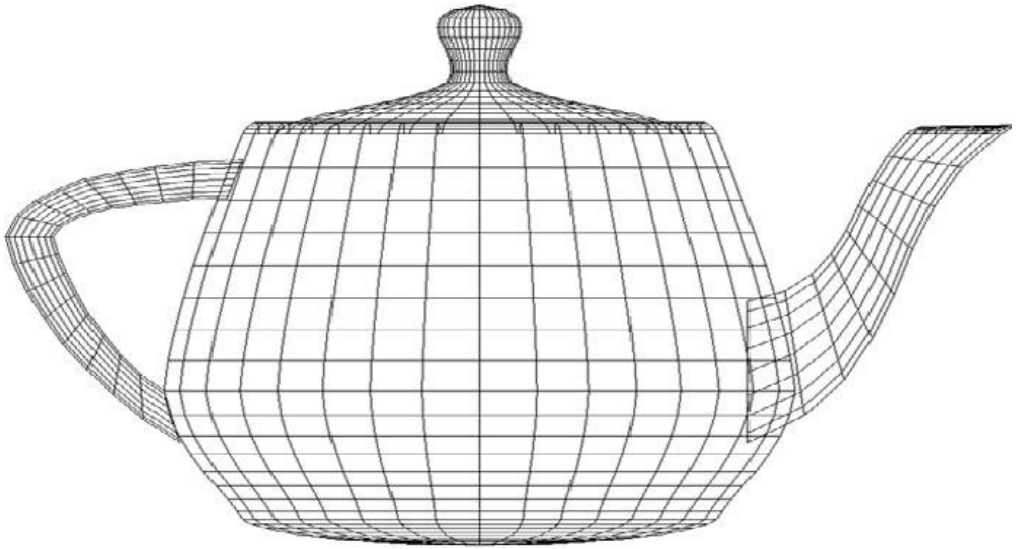
Rendering Other Polynomials

- Every polynomial is a Bezier polynomial for some set of control data
- We can use a Bezier renderer if we first convert the given control data to Bezier control data
  - Equivalent to converting between matrices
- Example: Interpolating to Bezier

$$M_B = M_I M_{BI}$$

Utah Teapot

- Most famous data set in computer graphics
- Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches



Quadrics

- Any quadric can be written as the quadratic form

  $\mathbf{p}^T\mathbf{A}\mathbf{p}+\mathbf{b}^T\mathbf{p}+c=0$ where $\mathbf{p}=[x, y, z]^T$

  with $\mathbf{A}$, $\mathbf{b}$ and c giving the coefficients

- Render by ray casting
  - Intersect with parametric ray $\mathbf{p}(\alpha)=\mathbf{p}_0+\alpha\mathbf{d}$ that passes through a pixel
  - Yields a scalar quadratic equation
    - No solution: ray misses quadric
    - One solution: ray tangent to quadric
    - Two solutions: entry and exit points

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Rendering the Teapot

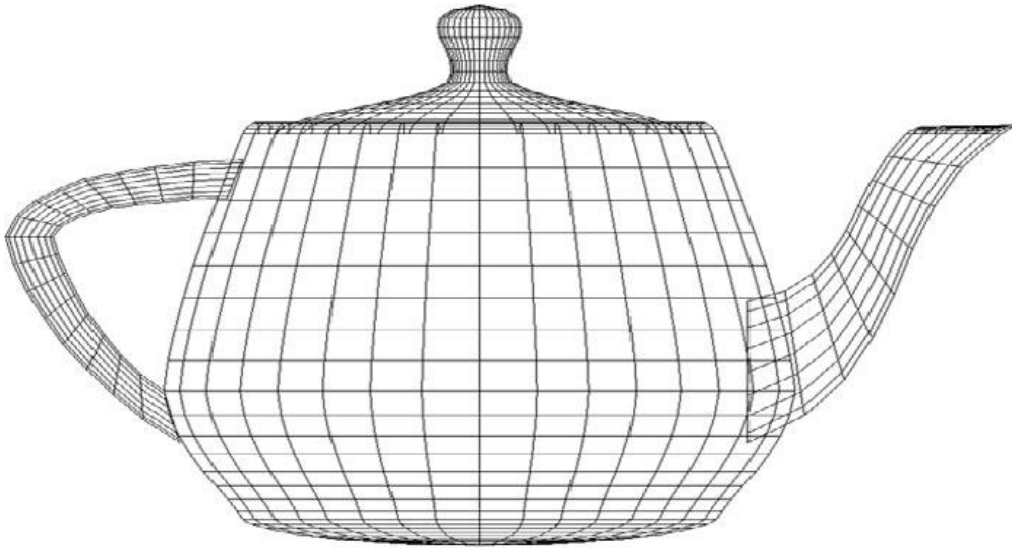
Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

Objectives

- Look at rendering with WebGL
- Use Utah teapot for examples
  - Recursive subdivision
  - Polynomial evaluation
  - Adding lighting

Utah Teapot

- Most famous data set in computer graphics
- Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches



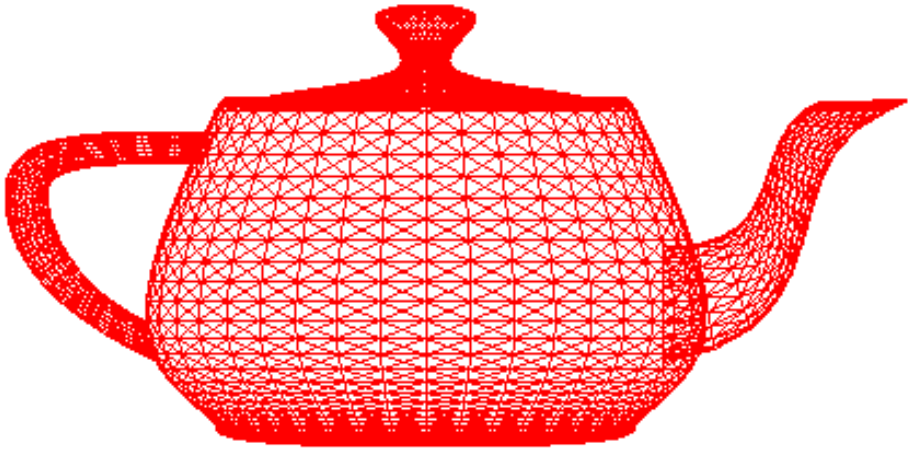
vertices.js

```
var numTeapotVertices = 306;
var vertices = [
vec3(1.4 , 0.0 , 2.4),
vec3(1.4 , -0.784 , 2.4),
vec3(0.784 , -1.4 , 2.4),
vec3(0.0 , -1.4 , 2.4),
vec3(1.3375 , 0.0 , 2.53125),
.
.
.
];
```

patches.js

```
var numTeapotPatches = 32;
var indices = new Array(numTeapotPatches);
    indices[0] = [0, 1, 2, 3,
    4, 5, 6, 7,
    8, 9, 10, 11,
    12, 13, 14, 15
    ];
    indices[1] = [3, 16, 17, 18,
.
.
 ];
```

Evaluation of Polynomials



Bezier Function

```
bezier = function(u) {
    var b = [];
    var a = 1-u;
    b.push(u*u*u);
    b.push(3*a*u*u);
    b.push(3*a*a*u);
    b.push(a*a*a);
   return b;
}
```

Patch Indices to Data

```
var h = 1.0/numDivisions;

   patch = new Array(numTeapotPatches);
   for(var i=0; i<numTeapotPatches; i++)
   patch[i] = new Array(16);
   for(var i=0; i<numTeapotPatches; i++)
       for(j=0; j<16; j++) {
          patch[i][j] = vec4([vertices[indices[i][j]][0],
           vertices[indices[i][j]][2],
              vertices[indices[i][j]][1], 1.0]);
   }
```

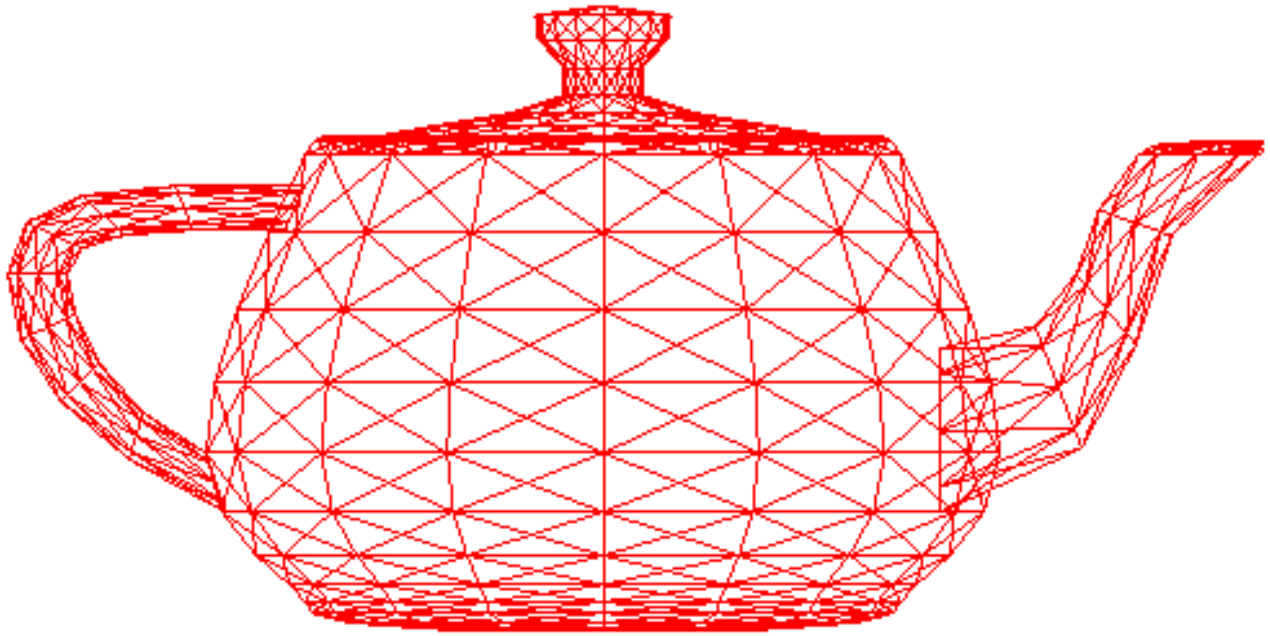
Vertex Data

```
for ( var n = 0; n < numTeapotPatches; n++ ) {
  var data = new Array(numDivisions+1);
  for(var j = 0; j<= numDivisions; j++) data[j] = new Array(numDivisions+1);
  for(var i=0; i<=numDivisions; i++) for(var j=0; j<= numDivisions; j++) {
      data[i][j] = vec4(0,0,0,1);
    var u = i*h;
    var v = j*h;
    var t = new Array(4);
    for(var ii=0; ii<4; ii++) t[ii]=new Array(4);
    for(var ii=0; ii<4; ii++) for(var jj=0; jj<4; jj++)
        t[ii][jj] = bezier(u)[ii]*bezier(v)[jj];
      for(var ii=0; ii<4; ii++) for(var jj=0; jj<4; jj++) {
            temp = vec4(patch[n][4*ii+jj]);
            temp = scale( t[ii][jj], temp);
            data[i][j] = add(data[i][j], temp);
    }
}
```

Quads

```
for(var i=0; i<numDivisions; i++)
   for(var j =0; j<numDivisions; j++) {
       points.push(data[i][j]);
       points.push(data[i+1][j]);
       points.push(data[i+1][j+1]);
       points.push(data[i][j]);
       points.push(data[i+1][j+1]);
       points.push(data[i][j+1]);
       index += 6;
       }
   }
```

Recursive Subdivision



Divide Curve

```
divideCurve = function( c, r , l){
 // divides c into left (l) and right ( r ) curve data
var mid = mix(c[1], c[2], 0.5);
   l[0] = vec4(c[0]);
   l[1] = mix(c[0], c[1], 0.5 );
   l[2] = mix(l[1], mid, 0.5 );
   r[3] = vec4(c[3]);
   r[2] = mix(c[2], c[3], 0.5 );
   r[1] = mix( mid, r[2], 0.5 );
   r[0] = mix(l[2], r[1], 0.5 );
   l[3] = vec4(r[0]);    return;
}
```

Divide Patch

```
dividePatch = function (p, count ) {
   if ( count > 0 ) {
    var a =  mat4();
    var b =  mat4();
    var t = mat4();
    var q = mat4();
    var r = mat4();
    var s = mat4();
// subdivide curves in u direction, transpose results, divide
// in u direction again (equivalent to subdivision in v)
        for ( var k = 0; k < 4; ++k ) {
            var pp = p[k];
            var aa = vec4();
            var bb = vec4();
```

Divide Patch

```
divideCurve( pp, aa, bb );
            a[k] = vec4(aa);
            b[k] = vec4(bb);
          }
      a = transpose( a );
      b = transpose( b );
            for ( var k = 0; k < 4; ++k ) {
            var pp = vec4(a[k]);
            var aa = vec4();
            var bb = vec4();
            divideCurve( pp, aa, bb );
             q[k] = vec4(aa);
             r[k] = vec4(bb);
            }
      for ( var k = 0; k < 4; ++k ) {
            var pp = vec4(b[k]);
            var aa = vec4();
```

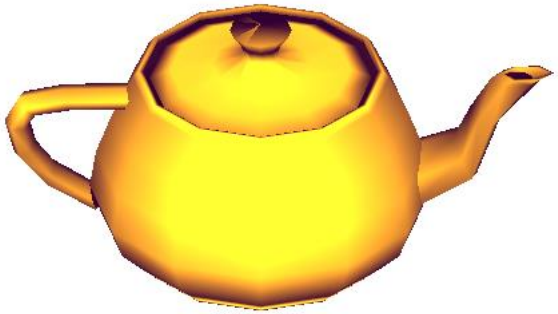
Divide Patch

```
            var bb = vec4();
            divideCurve( pp, aa, bb );
             t[k] = vec4(bb);
            }
// recursive division of 4 resulting patches
        dividePatch( q, count - 1 );
        dividePatch( r, count - 1 );
        dividePatch( s, count - 1 );
        dividePatch( t, count - 1 );
    }
    else {
        drawPatch( p );
    }
    return;
}
```

Draw Patch

```
drawPatch = function(p) {
    // Draw the quad (as two triangles) bounded by
    //   corners of the Bezier patch
    points.push(p[0][0]);
    points.push(p[0][3]);
    points.push(p[3][3]);
    points.push(p[0][0]);
    points.push(p[3][3]);
    points.push(p[3][0]);
    index+=6;
    return;
}
```

Adding Shading



Using Face Normals

```
var t1 = subtract(data[i+1][j], data[i][j]);
var t2  =subtract(data[i+1][j+1], data[i][j]);
var normal = cross(t1, t2);
normal = normalize(normal);
normal[3] =  0;
points.push(data[i][j]);          normals.push(normal);
points.push(data[i+1][j]);        normals.push(normal);
points.push(data[i+1][j+1]);      normals.push(normal);
points.push(data[i][j]);          normals.push(normal);
points.push(data[i+1][j+1]);      normals.push(normal);
points.push(data[i][j+1]);        normals.push(normal);
index+= 6;
```

Exact Normals

```
nbezier = function(u) {
    var b = [];
    b.push(3*u*u);
    b.push(3*u*(2-3*u));
    b.push(3*(1-4*u+3*u*u));
    b.push(-3*(1-u)*(1-u));
    return b;
}
```

Geometry Shader

- Basic limitation on rasterization is that each execution of a vertex shader is triggered by one vertex and can output only one vertex
- Geometry shaders allow a single vertex and other data to produce many vertices
- Example: send four control points to a geometry shader and it can produce as many points as needed for Bezier curve

Tessellation Shaders

- Can take many data points and produce triangles
- More complex since tessellation has to deal with inside/outside issues and topological issues such as holes
- Neither geometry or tessellation shaders supported by ES
- ES 3.1 (just announced) has compute shaders