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

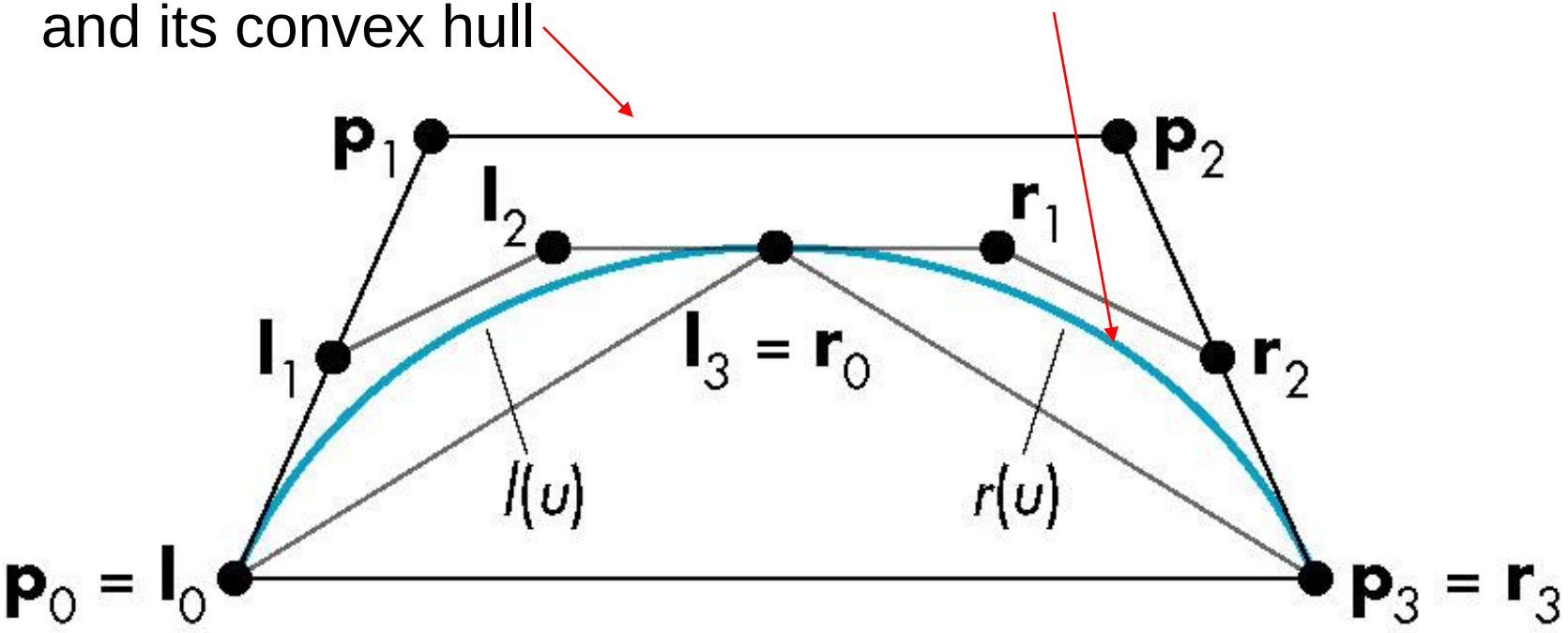
deCasteljau Recursion

- 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”

The University of New Mexico

Splitting a Cubic Bezier

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

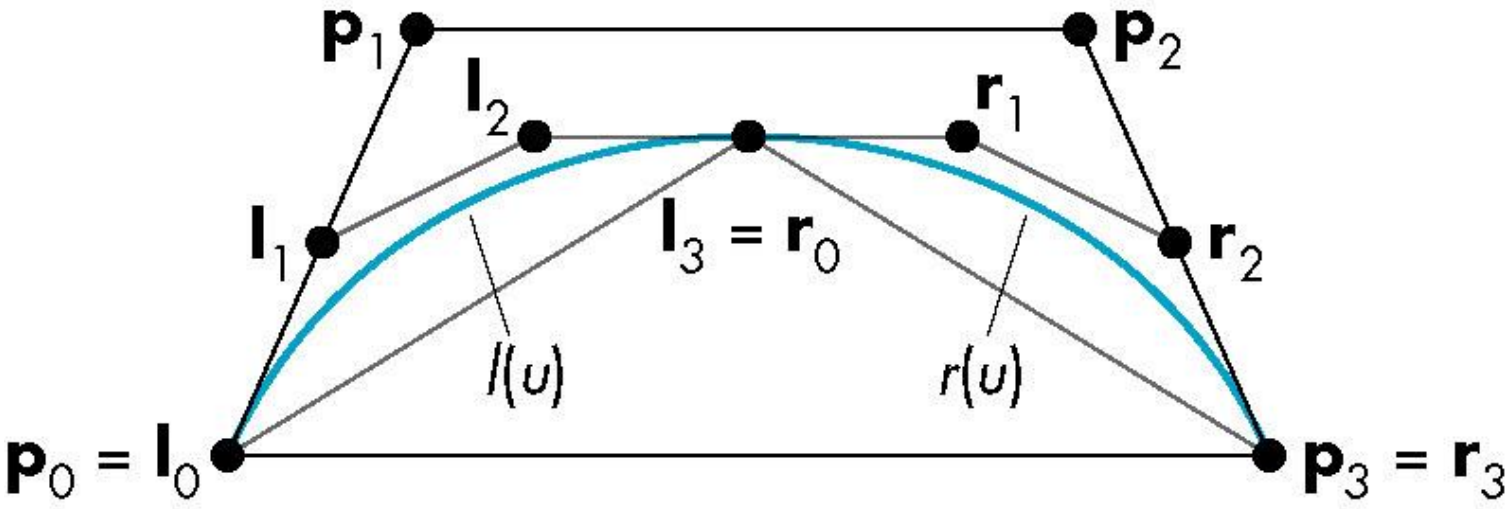


Consider left half $l(u)$ and right half $r(u)$

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

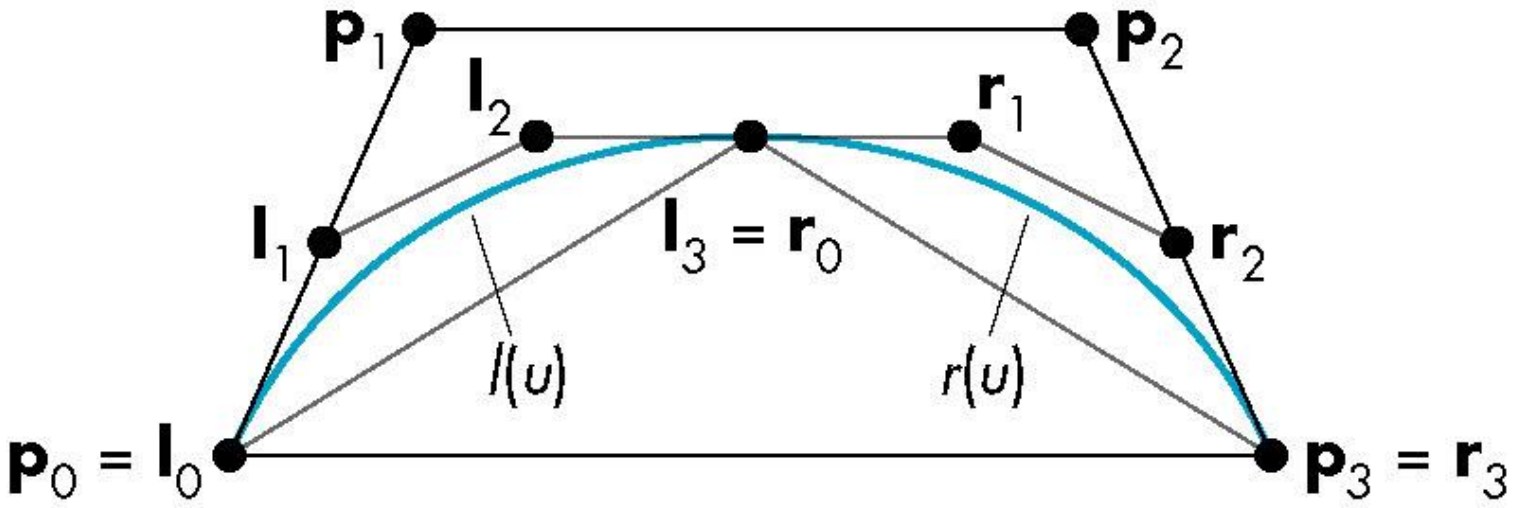


The University of New Mexico

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.



The University of New Mexico

Equations

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

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

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

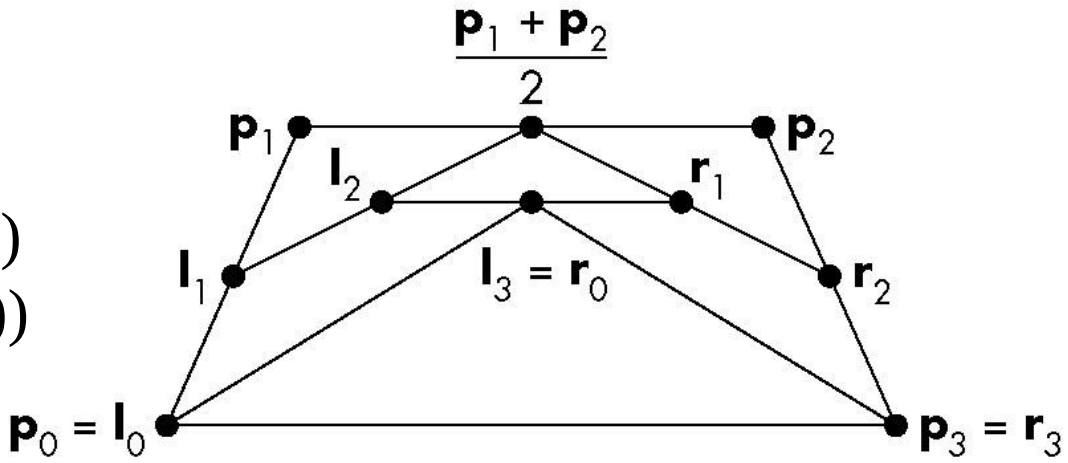
Matching slopes, taking into account that $l(u)$ and $r(u)$ only go over half the distance as $p(u)$

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

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

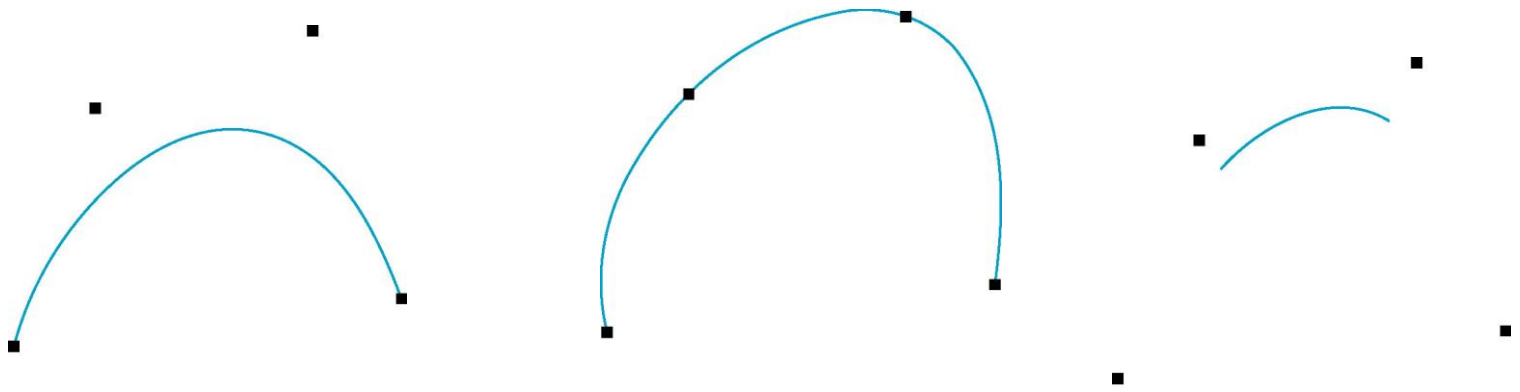
Interpolating to Bezier $\mathbf{M}_B^{-1}\mathbf{M}_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{5}{6} & 3 & -\frac{3}{2} & \frac{1}{3} \\ \frac{1}{3} & -\frac{3}{2} & 3 & -\frac{5}{6} \\ 0 & 0 & 0 & 1 \end{bmatrix}$

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

The University of New Mexico

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



Bezier

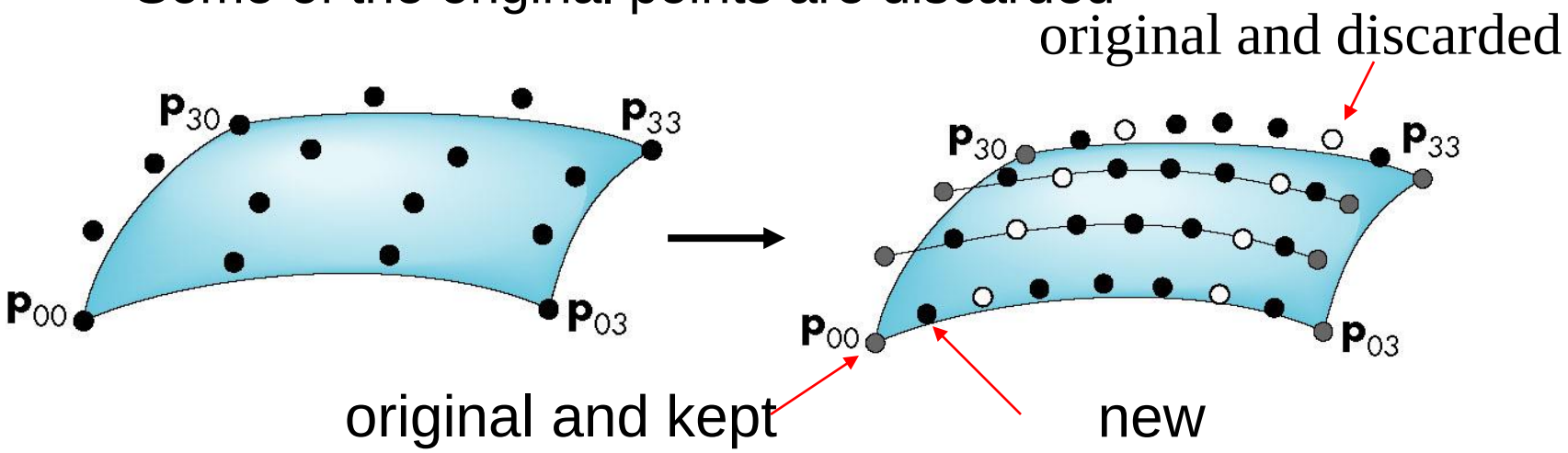
Interpolating

B Spline

The University of New Mexico

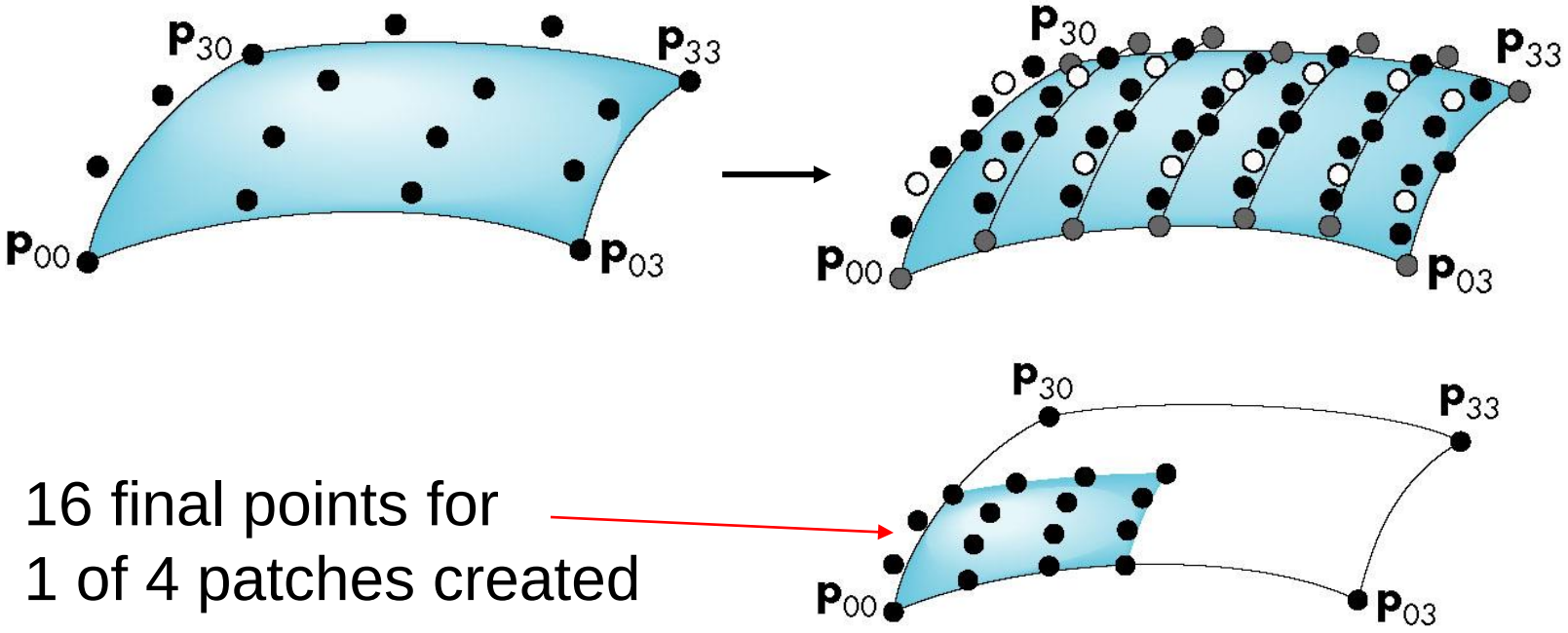
Surfaces

- 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

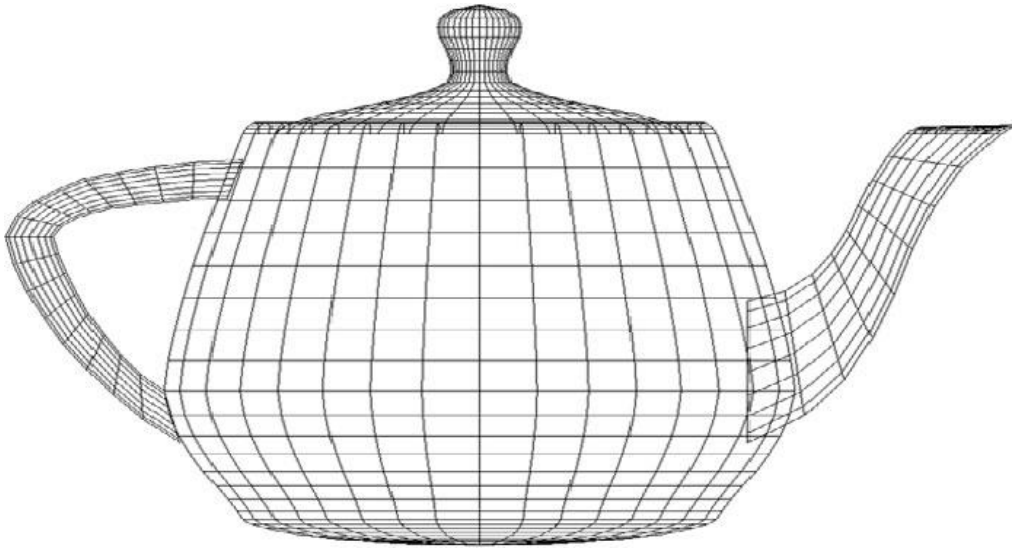
Rendering Other Polynomials

- 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

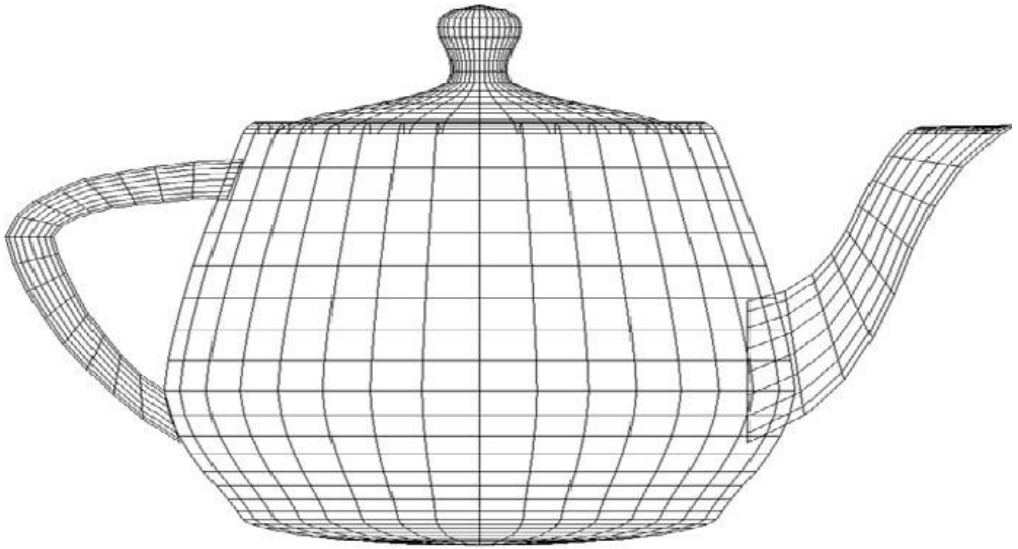
Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

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



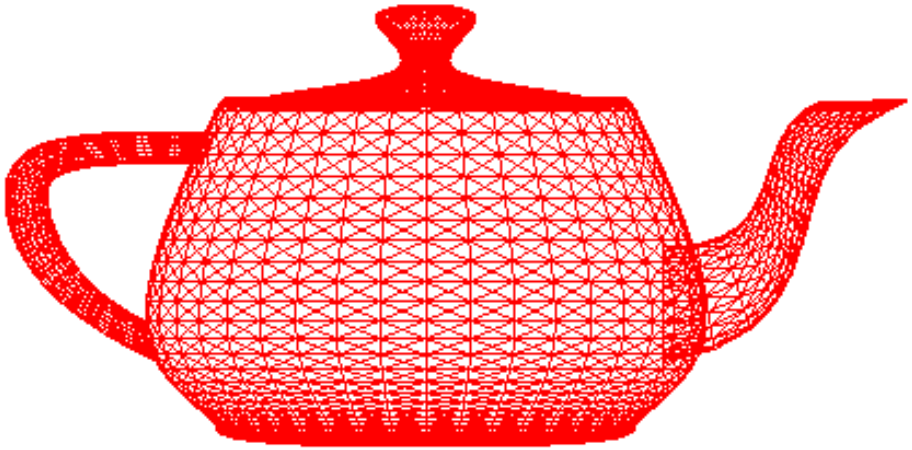
vertices.js

```
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]);
   }
```

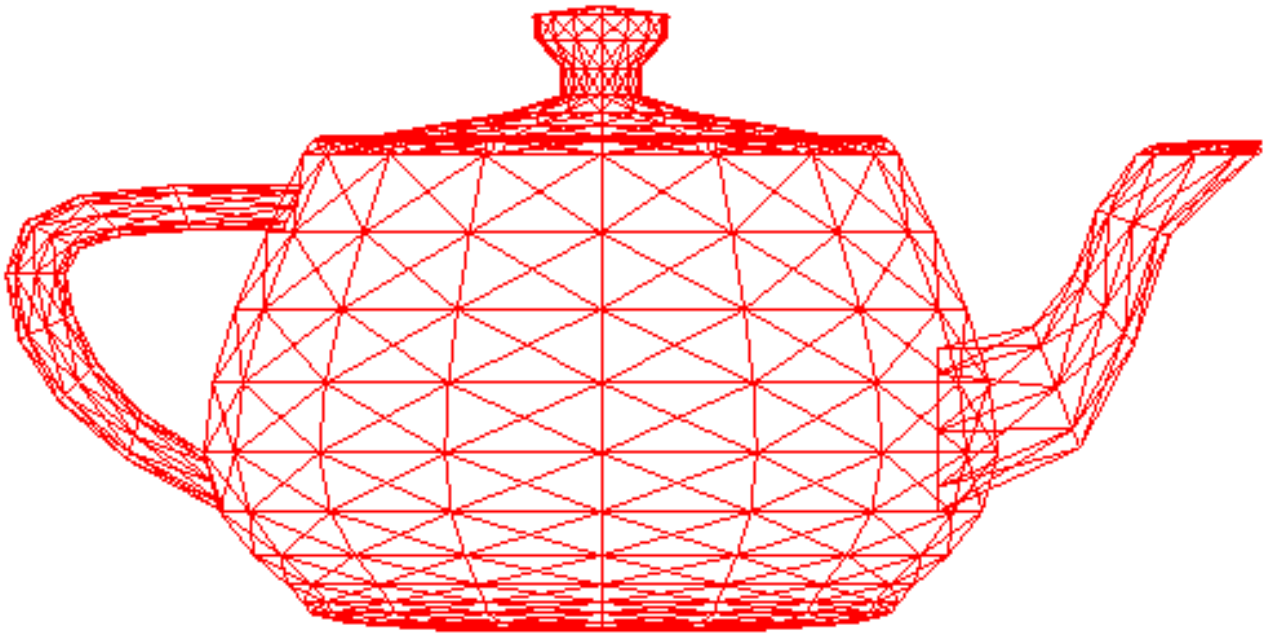
Vertex Data

```
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();
```

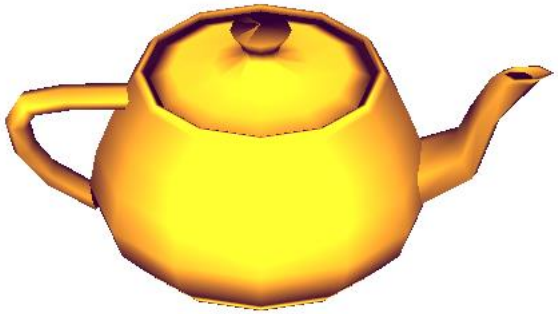
Divide Patch

```
            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