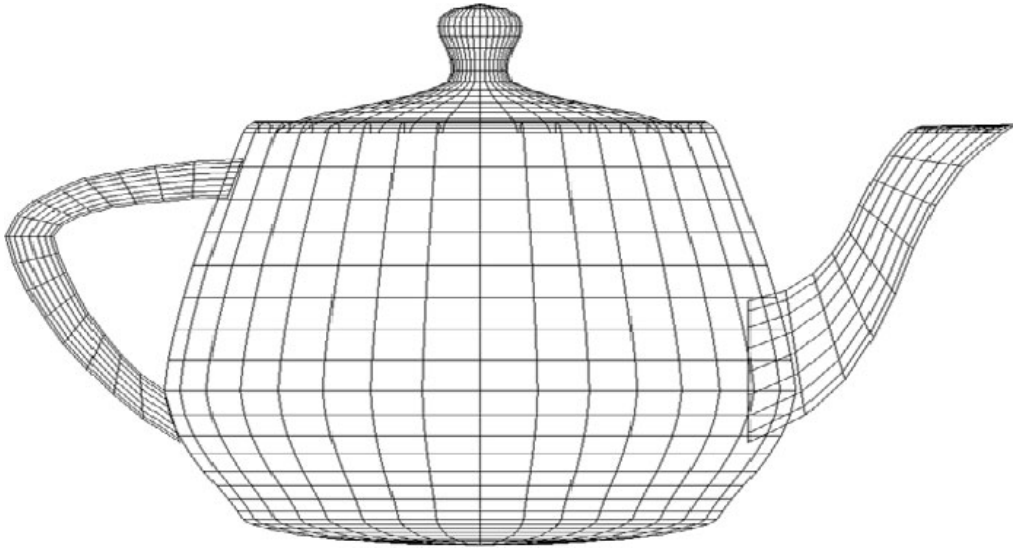
Rendering the Teapot

THE OHIO STATE UNIVERSITY
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Utah Teapot



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),
.
.
.
];
```

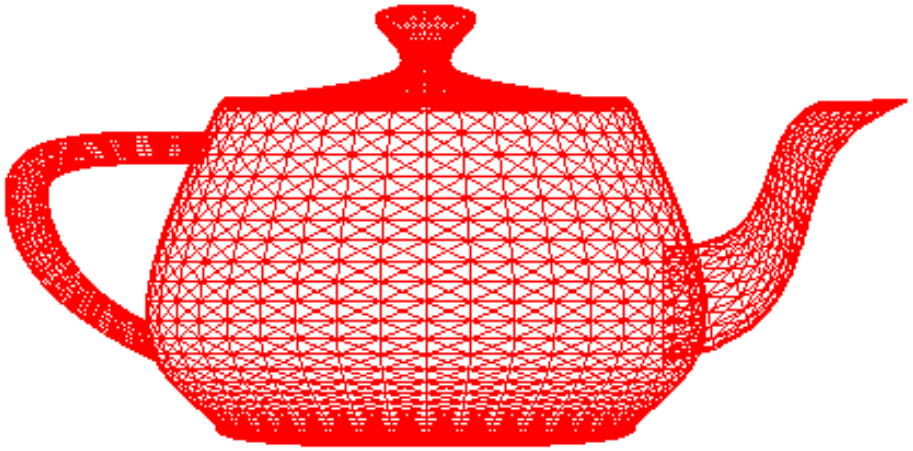

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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,
.
.
 ];
```


THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Evaluation of Polynomials



THE OHIO STATE UNIVERSITY
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Modeling

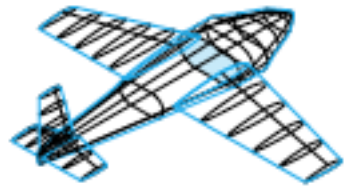


FIGURE 10.5 Model airplane

FIGURE 10.6 Cross-section curve.

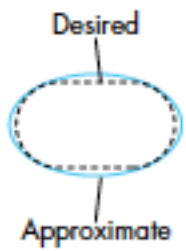


FIGURE 10.7 Approximation of cross-section curve.

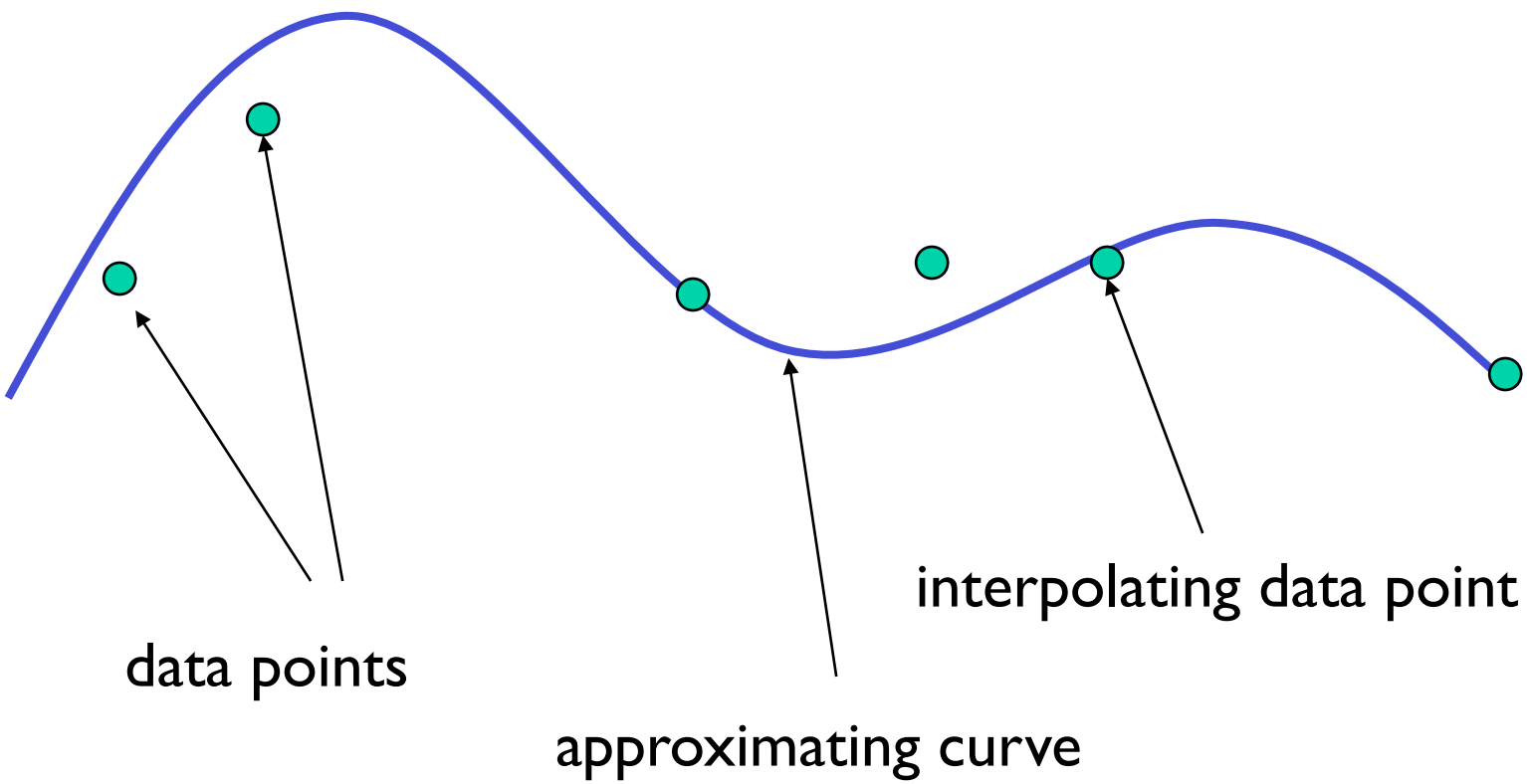
Topics

- Introduce types of curves and surfaces
  - Explicit
  - Implicit
  - Parametric
- Discuss Modeling and Approximations

Escaping Flatland

- Lines and flat polygons
  - Fit well with graphics hardware
  - Mathematically simple
- But world is not flat
  - Need curves and curved surfaces
  - At least at the application level
  - Render them approximately with flat primitives

Modeling with Curves



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Good Representation?

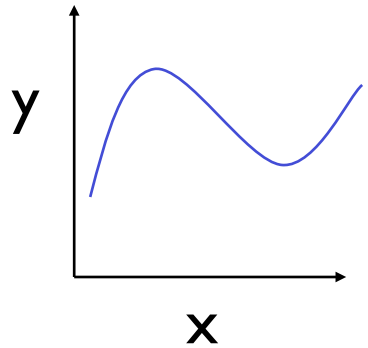
- Properties
  - Stable
  - Smooth
  - Easy to evaluate
  - Must we interpolate or can we just come close to data?
  - Do we need derivatives?

Explicit Representation

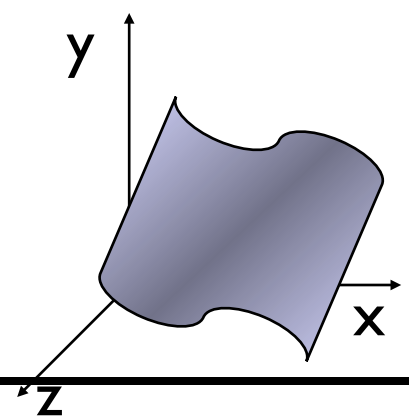
- Function

  $$y=f(x)$$

- 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

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

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

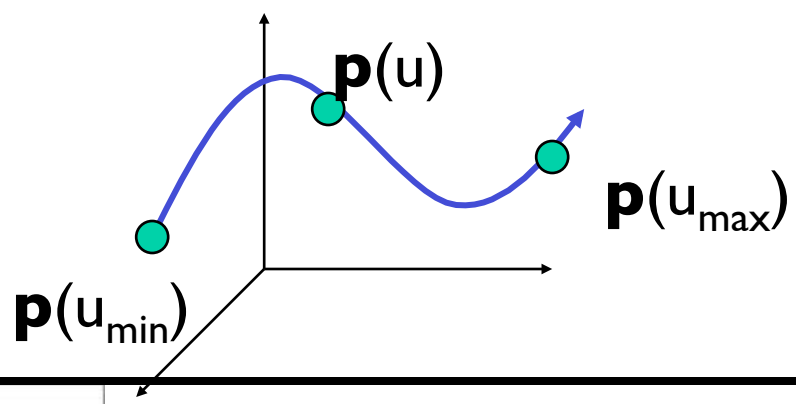
Parametric Curves

- 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

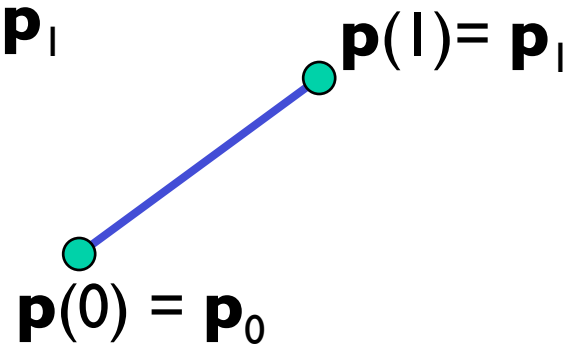


Parametric Lines

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

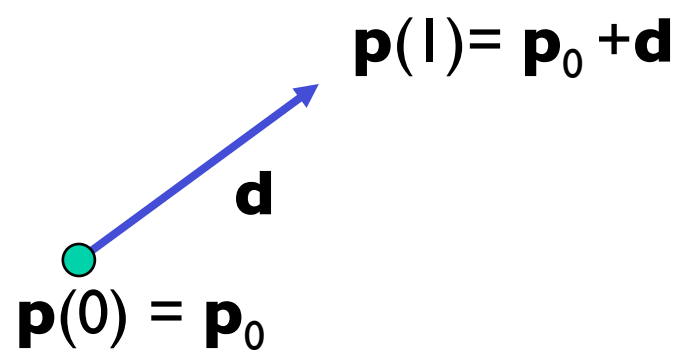
Line connecting two points $\mathbf{p}_0$ and $\mathbf{p}_1$

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



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

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

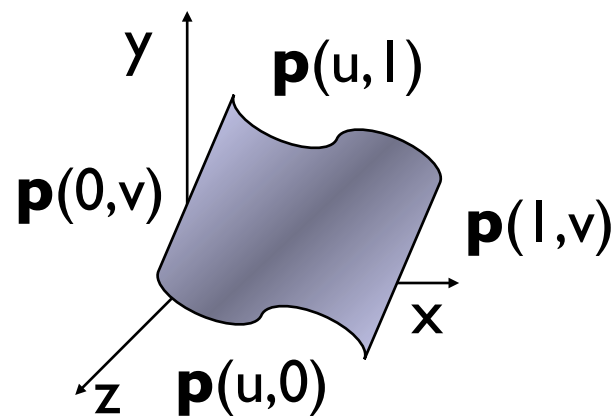


Parametric Surfaces

- Surfaces require 2 parameters

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

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



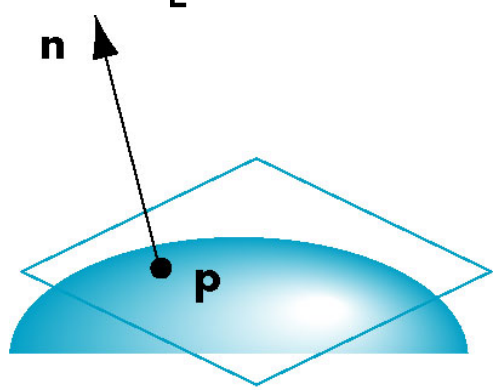
- Want same properties as curves:
  - Smoothness
  - Differentiability
  - Ease of evaluation

Normals

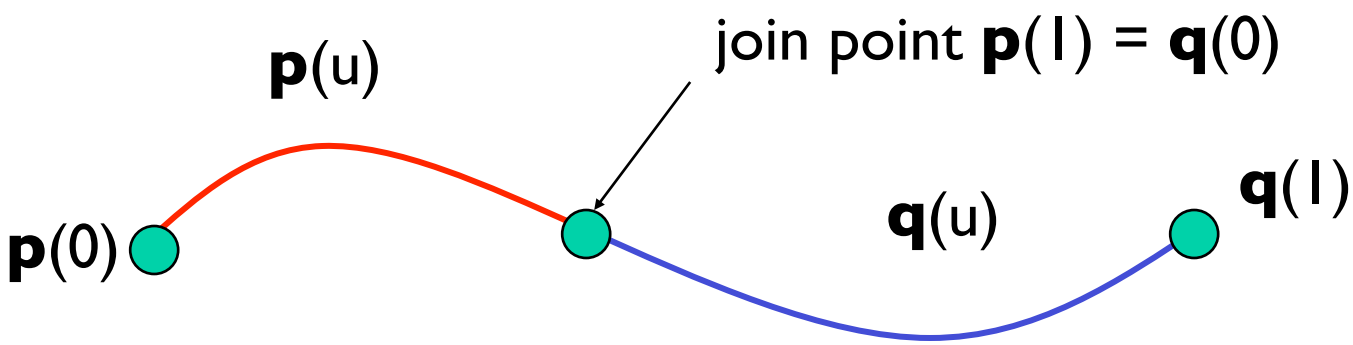
We can differentiate with respect to u and v to obtain the normal at any point **p**

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

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



Curve Segments



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Parametric Polynomial Curves

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

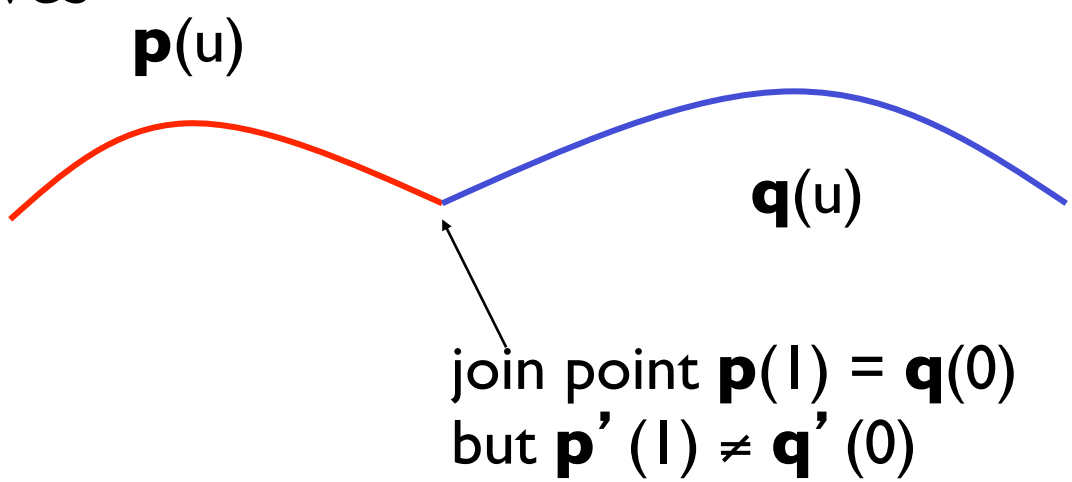
- If N=M=L, we need to determine 3(N+1) coefficients

- Curves for x, y and z are independent,
  we can define each independently in an identical manner

- We will use the form
  where p can be any of x, y, z $\quad \mathrm{p}(u) = \sum_{k=0}^{L} c_k u^k$

Why Polynomials

- Easy to evaluate
- Continuous and differentiable everywhere
  - Continuity at join points including continuity of derivatives



Cubic Polynomials

- N=M=L=3,

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

- 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

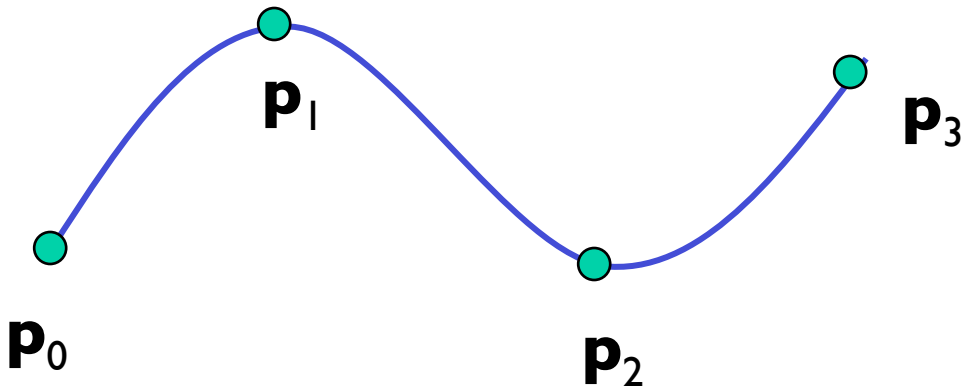
Matrix-Vector Form

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

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

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

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

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

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

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

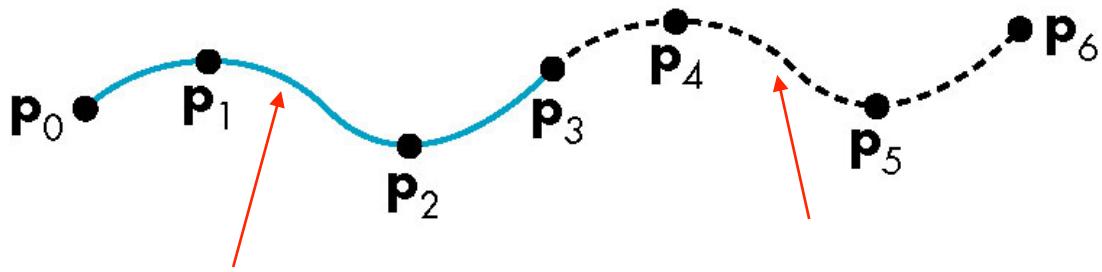
Solving for **c** we find the *interpolation matrix*

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

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

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)

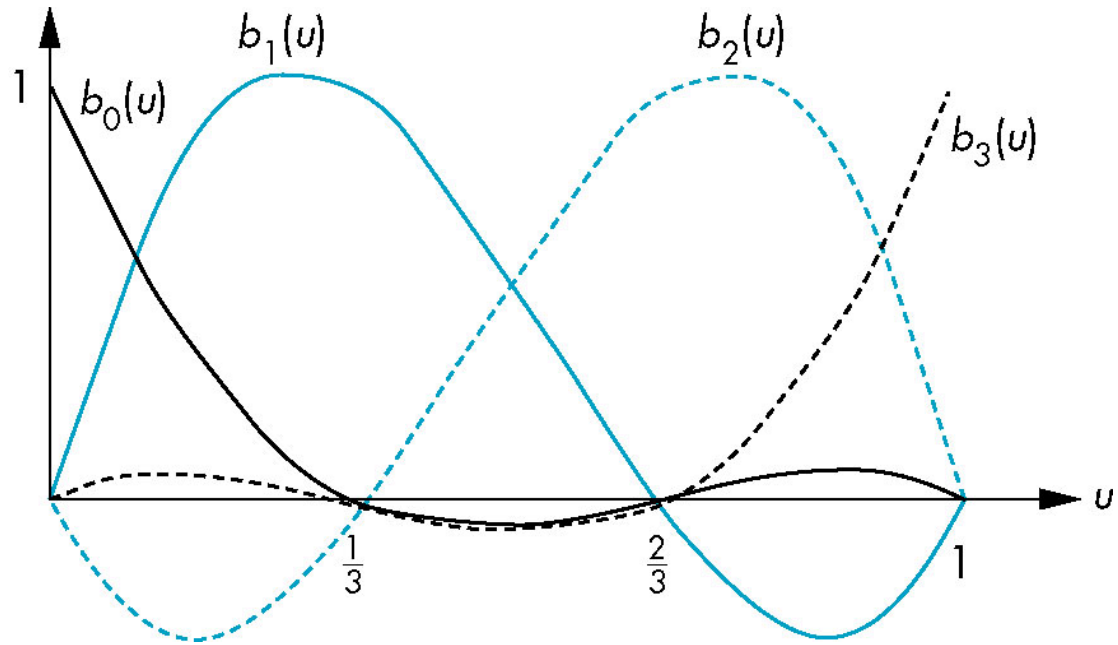
$$p(u)=\mathbf{u}^T\mathbf{c}=\mathbf{u}^T\mathbf{M}_I\mathbf{p} = \mathbf{b}(u)^T\mathbf{p}$$

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 – NOT GOOD



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

As Opposed to …

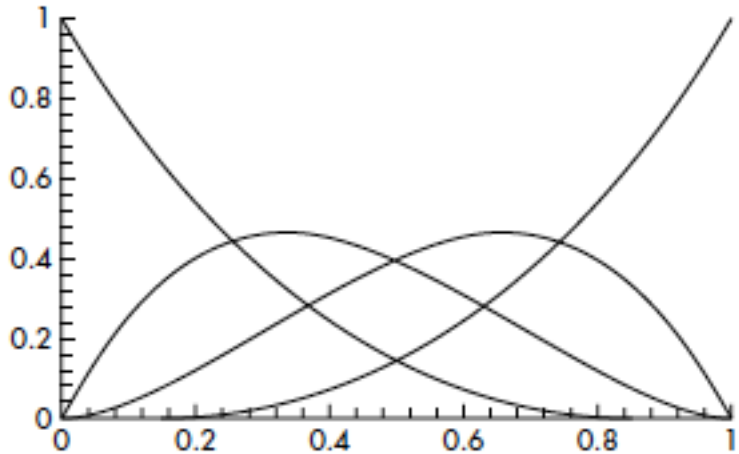
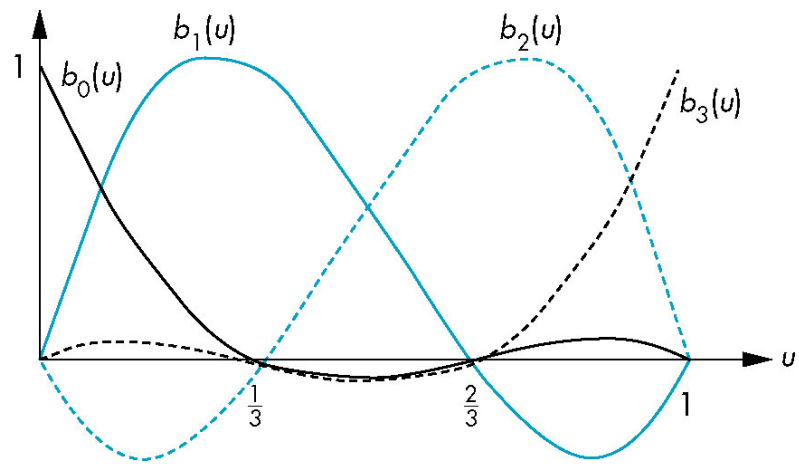


FIGURE 10.18 Blending polynomials for the Bézier cubic.



Parametric Surface

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix}$$

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

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

$$\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v.$$

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix} = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{c}_{ij} u^i v^j.$$

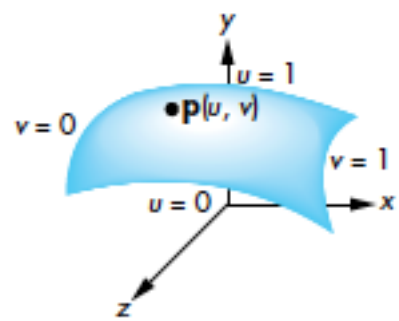


FIGURE 10.4 Surface patch.

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

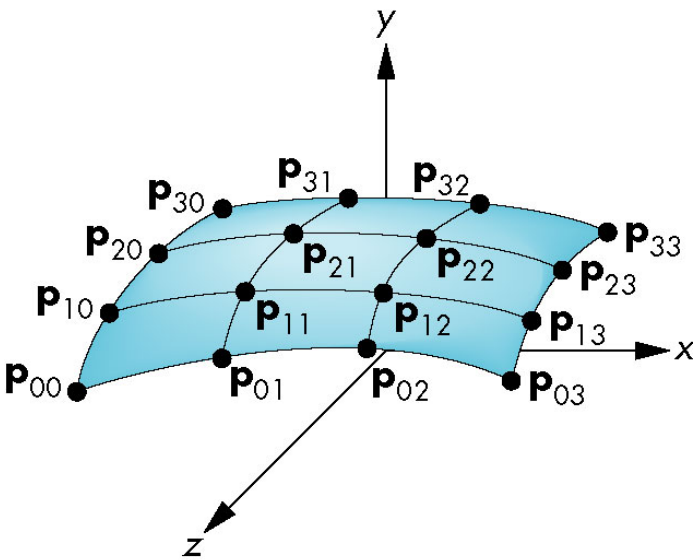
Need 48 coefficients (3 independent sets of 16) to determine a surface patch

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

Bezier and Spline Curves and Surfaces

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Bezier

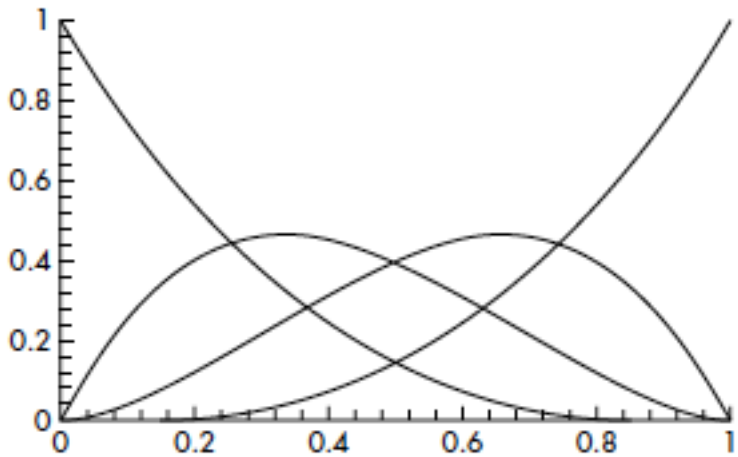


FIGURE 10.18 Blending polynomials for the Bézier cubic.

$$\mathbf{p}(u) = \sum_{i=0}^{3} b_i(u)\mathbf{p}_i,$$

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

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

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

Bezier Surface Patches

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

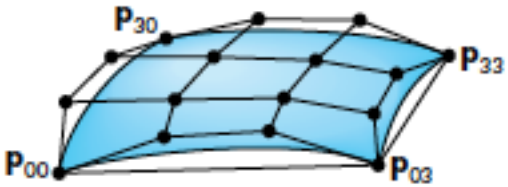
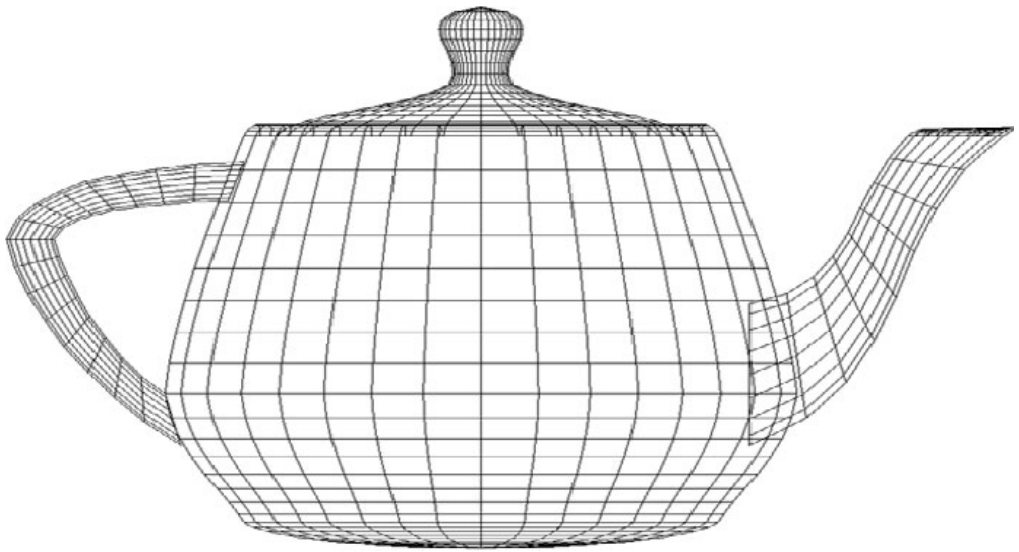


FIGURE 10.20 Bézier patch.

Utah Teapot – Bezier Avatar

Available as a list of 306 3D vertices and the indices that define 32 Bezier patches

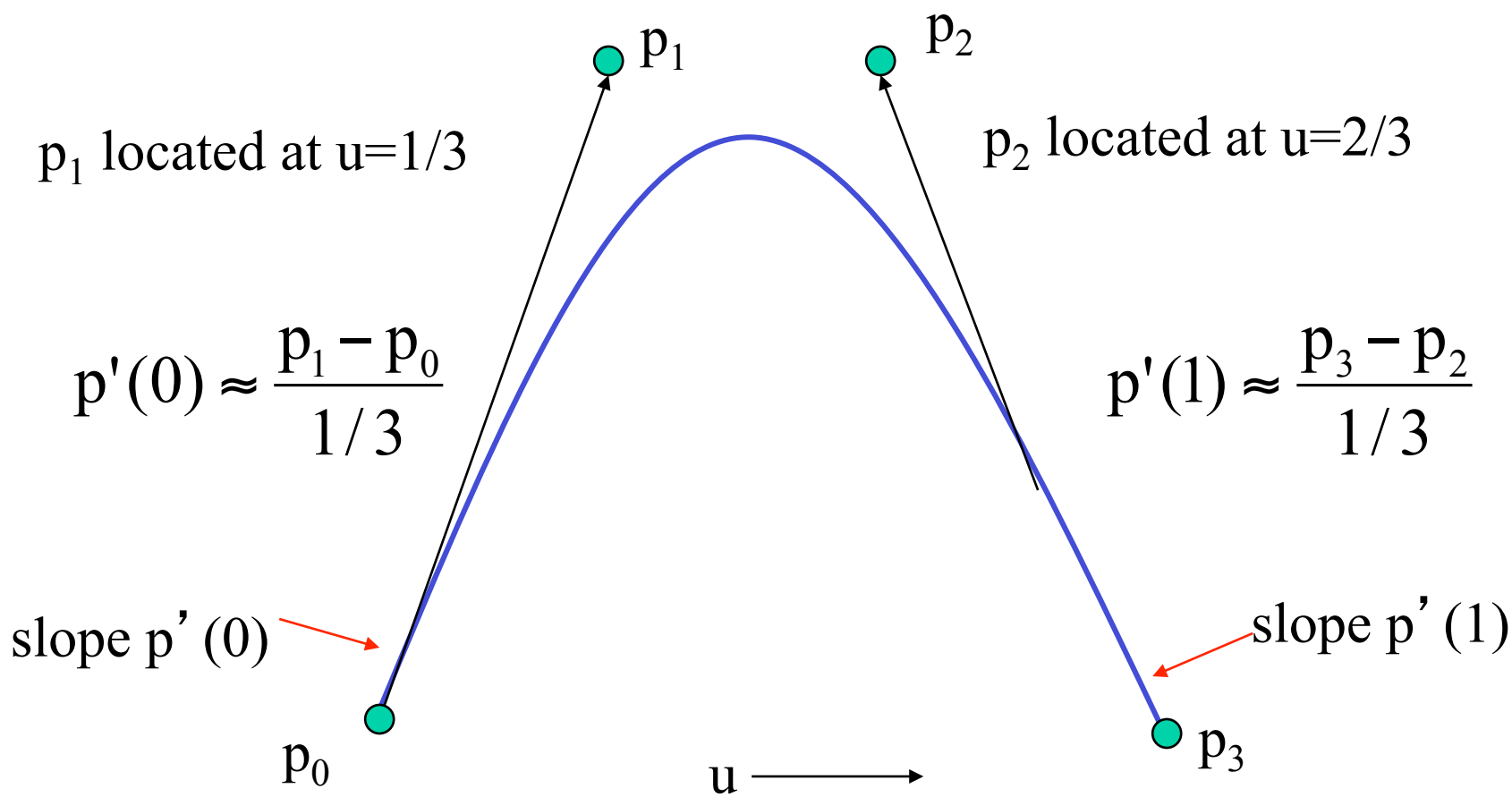


THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Bezier

- Do not usually have derivative data
- Bezier suggested using the same 4 data points as with the cubic interpolating curve to approximate the derivatives

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

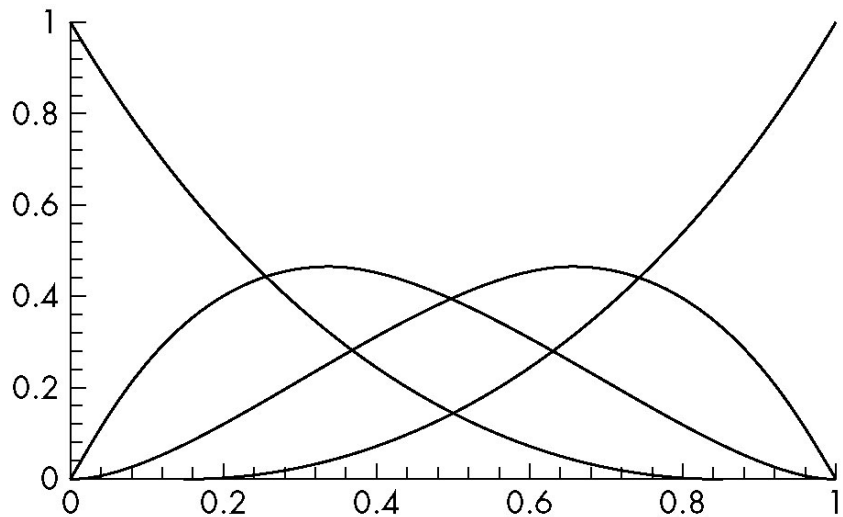
$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

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

blending functions

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

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

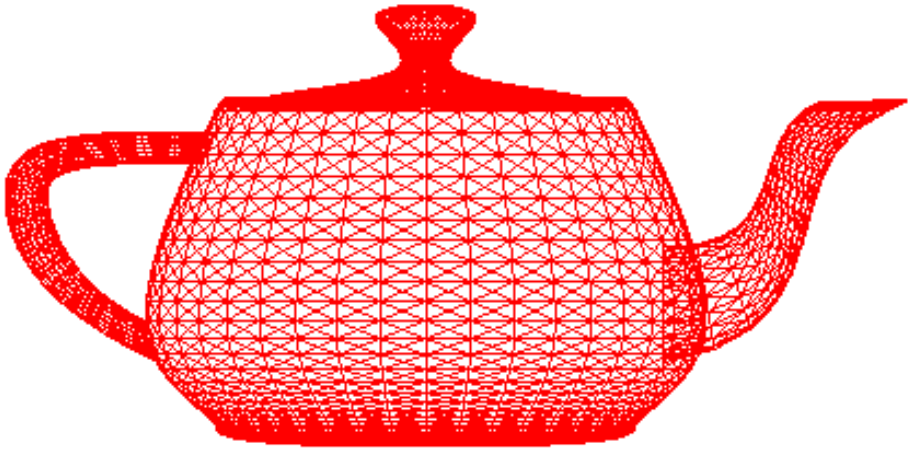
Blending polynomials for any degree *d*

- All zeros at 0 and 1
- For any degree they all sum to 1
- They are all between 0 and 1 inside (0,1)

Rendering Curves and Surfaces

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Evaluation of Polynomials



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Evaluating Polynomials

- Polynomial curve – evaluate polynomial at many points and form an approximating polyline
- Surfaces – approximating mesh of triangles or quadrilaterals

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

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

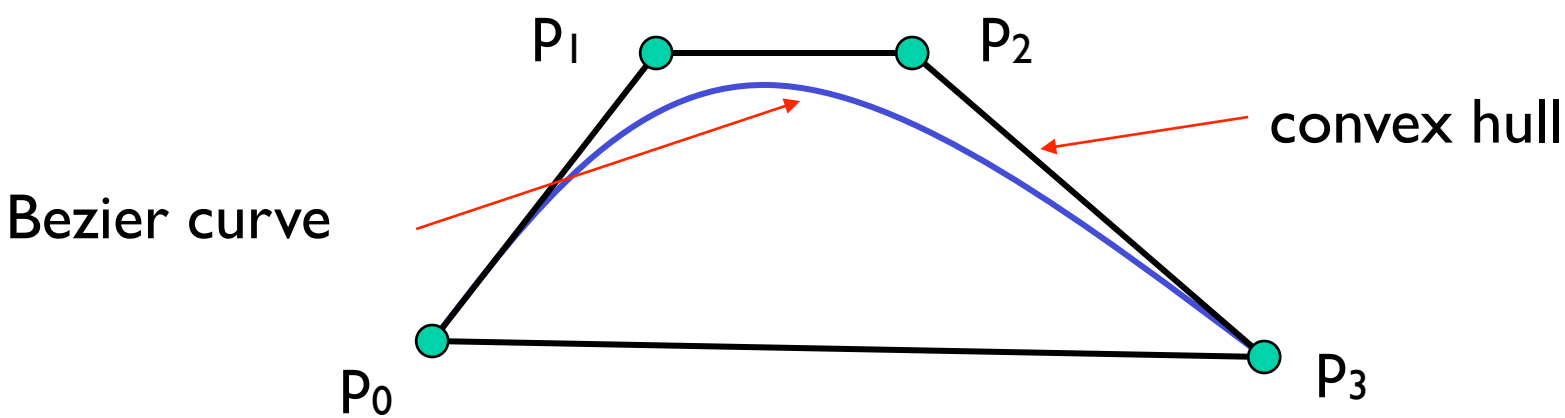
$$\mathbf{b}(u) = \mathbf{M}_B^T \mathbf{u} = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}.$$

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

Evaluate without Computation?

Convex Hull Property

- Bezier curves lie in the convex hull of their control points
- Do not interpolate all the data; cannot be too far away

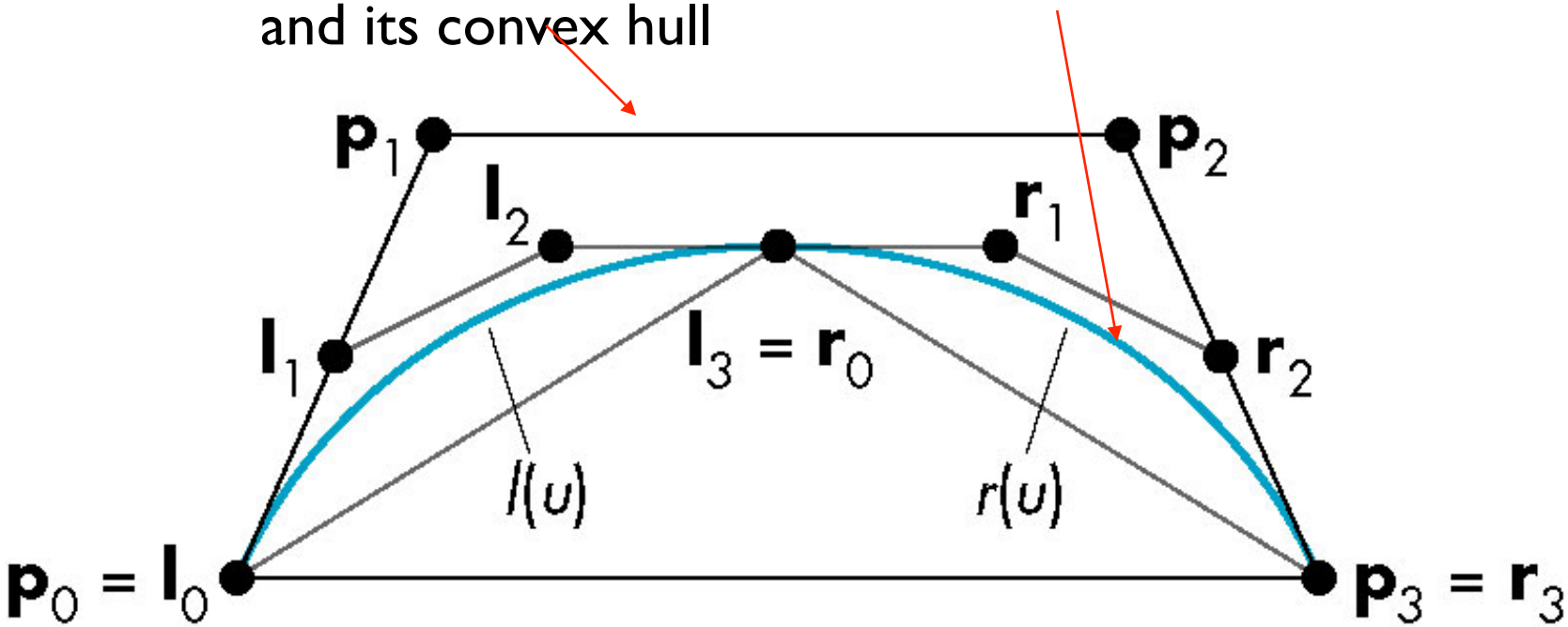


deCasteljau Recursion

- Use convex hull property 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

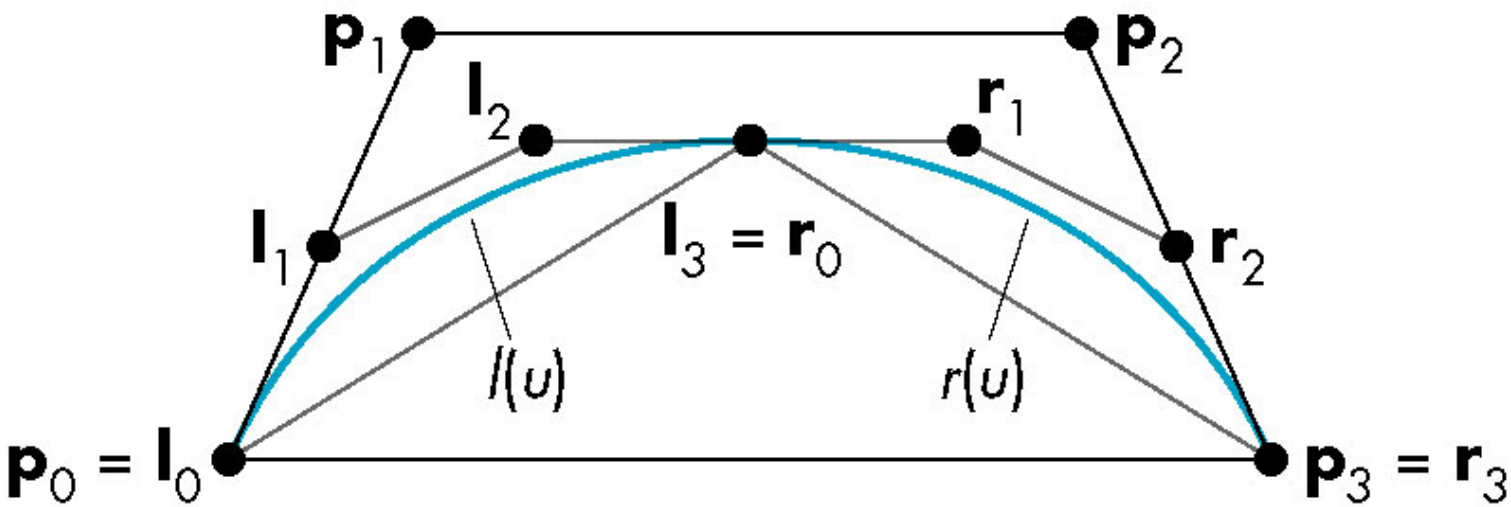
p_0, p_1, p_2, p_3 determine a cubic Bezier polynomial and its convex hull



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

l(u) and r(u)

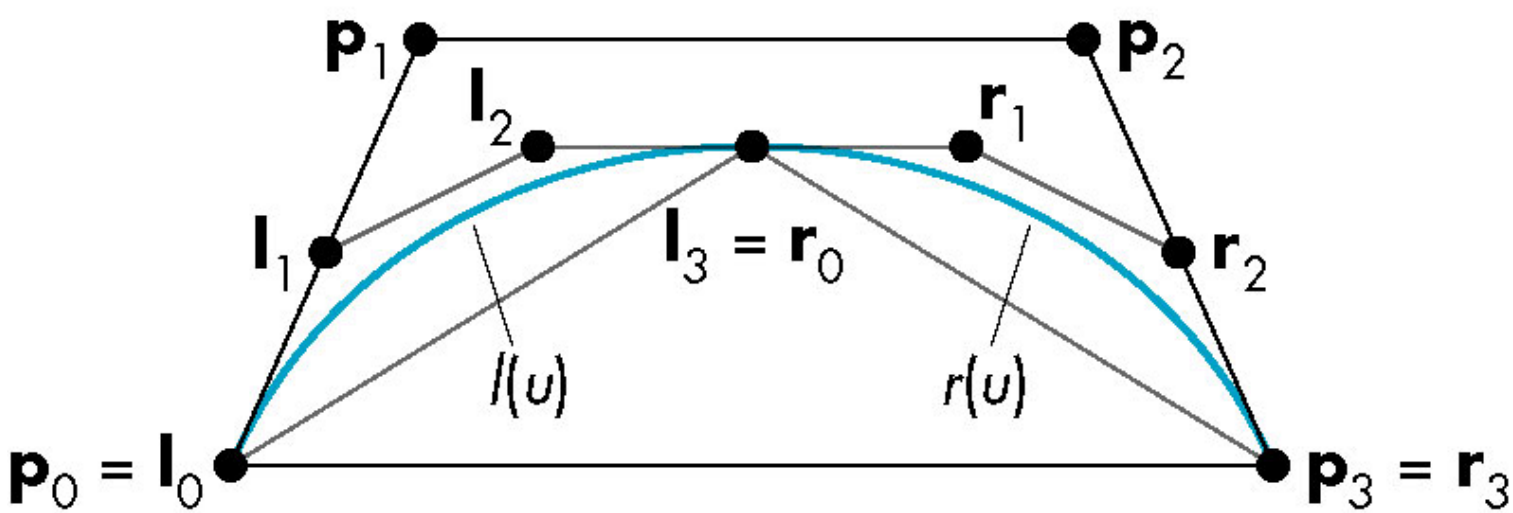
Since l(u) and r(u) are Bezier curves, we should be able to find two sets of control points $\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ that determine them



Convex Hulls

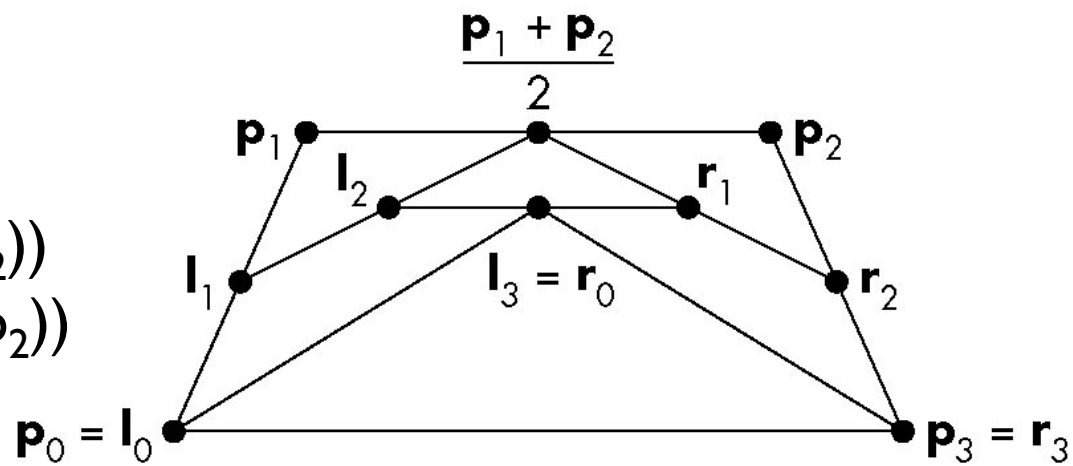
$\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$each have a convex hull that that is closer to p(u) than the convex hull of $\{p_0, p_1, p_2, p_3\}$ This is known as the *variation diminishing property*.

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.



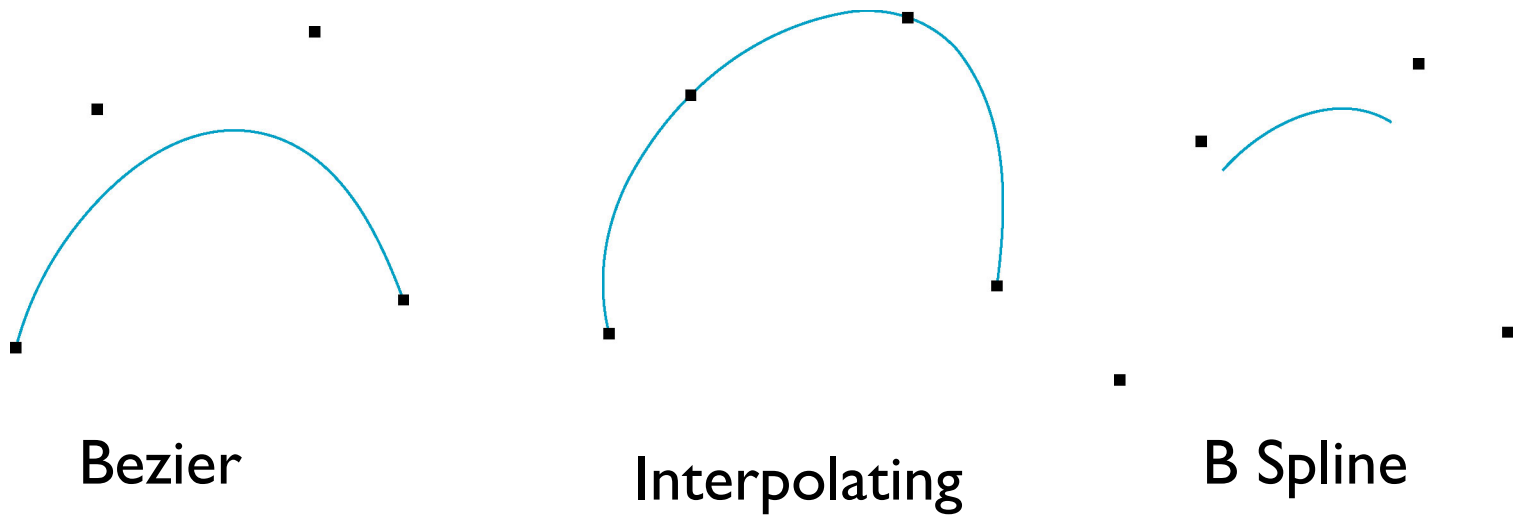
Efficient Form

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



Requires only shifts and adds!

Bezier in general



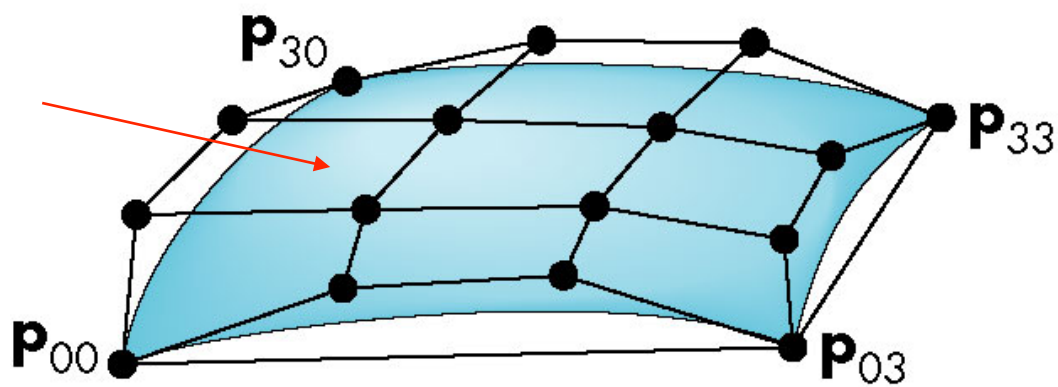
THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Bezier Patches

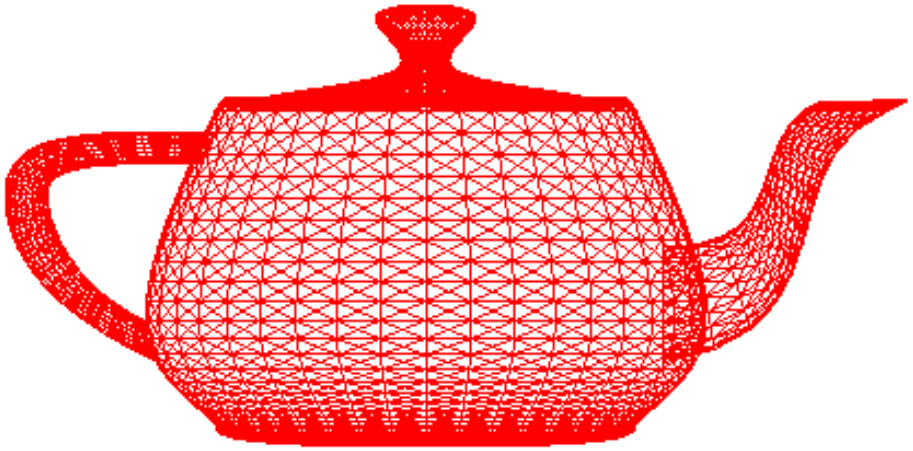
Using same data array $\mathbf{P}=[p_{ij}]$ as with interpolating form

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

Patch lies in convex hull

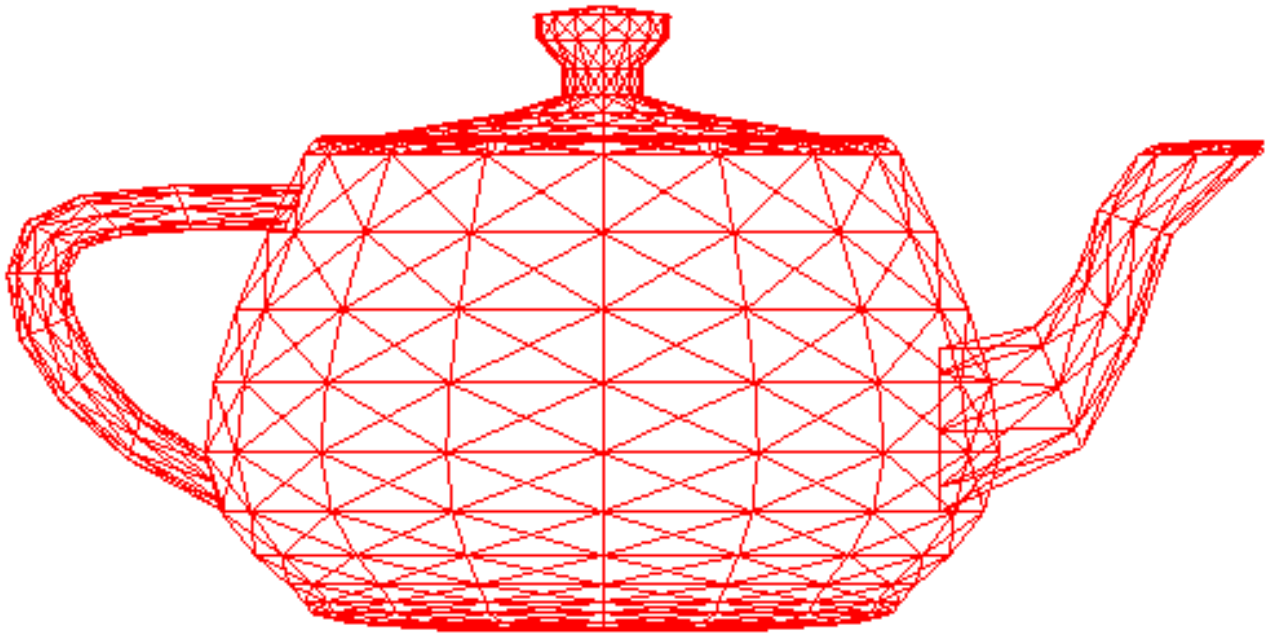


Evaluation of Polynomials



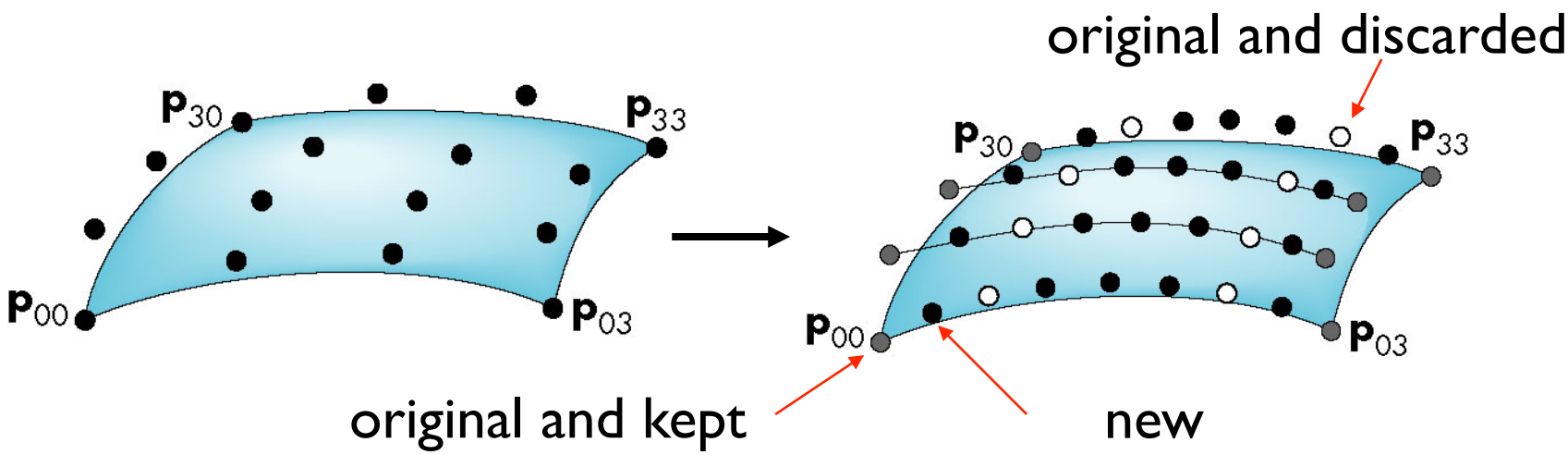
THE OHIO STATE UNIVERSITY
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

deCasteljau Recursion



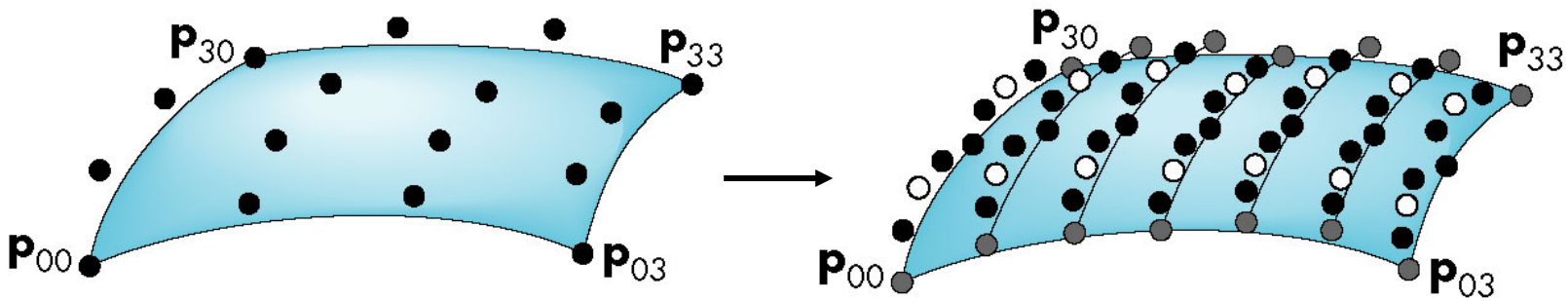
Surfaces

- 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

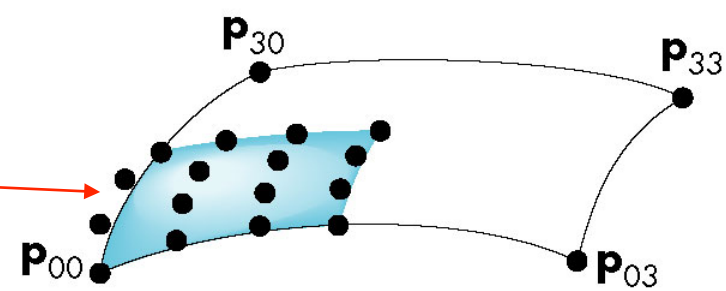


Second Subdivision

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



16 final points for
1 of 4 patches created



Angel's Gifts – Chapter 11, 7e

http://www.cs.unm.edu/~angel/WebGL/7E/11/

vertices.js: three versions of the data vertex data

patches.js: teapot patch data

teapot1: wire frame teapot by recursive subdivision of Bezier curves

teapot2: wire frame teapot using polynomial evaluation

teapot3: same as teapot2 with rotation

teapot4: shaded teapot using polynomial evaluation and exact normals

teapot5: shaded teapot using polynomial evaluation and normals computed for each triangle

vertices.js

http://www.cs.unm.edu/~angel/WebGL/7E/11/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),
.
.
.
];
```


THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

patches.js

http://www.cs.unm.edu/~angel/WebGL/7E/11/patches.js

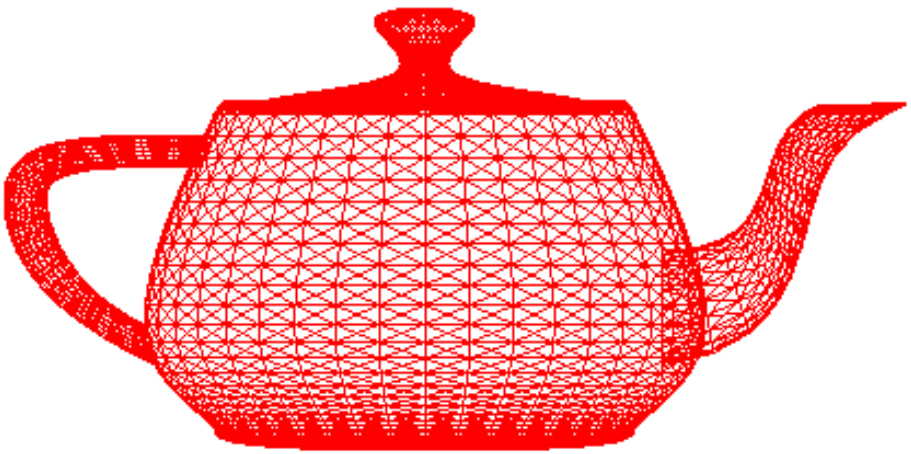
```
var numTeapotPatches = 32;
var indices = new Array(numTeapotPatches);
    indices[0] = [0, 1, 2, 3,
    4, 5, 6, 7,
    8, 9, 10, 11,
    12, 13, 14, 15
    ];
    indices[1] = [3, 16, 17, 18,
.
.
 ];
```

Patch Reader

http://www.cs.unm.edu/~angel/WebGL/7E/11/patchReader.js

Bezier Function – Teapot 2

```
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

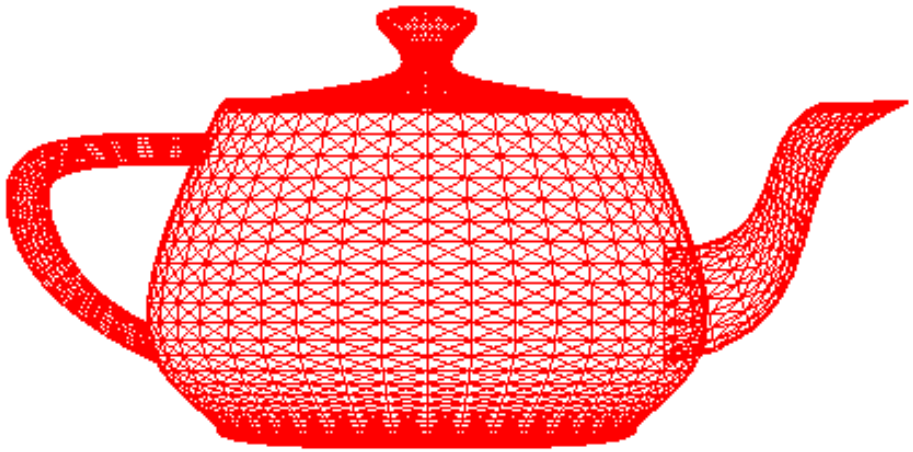
```
var h = 1.0/numDivisions;

  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

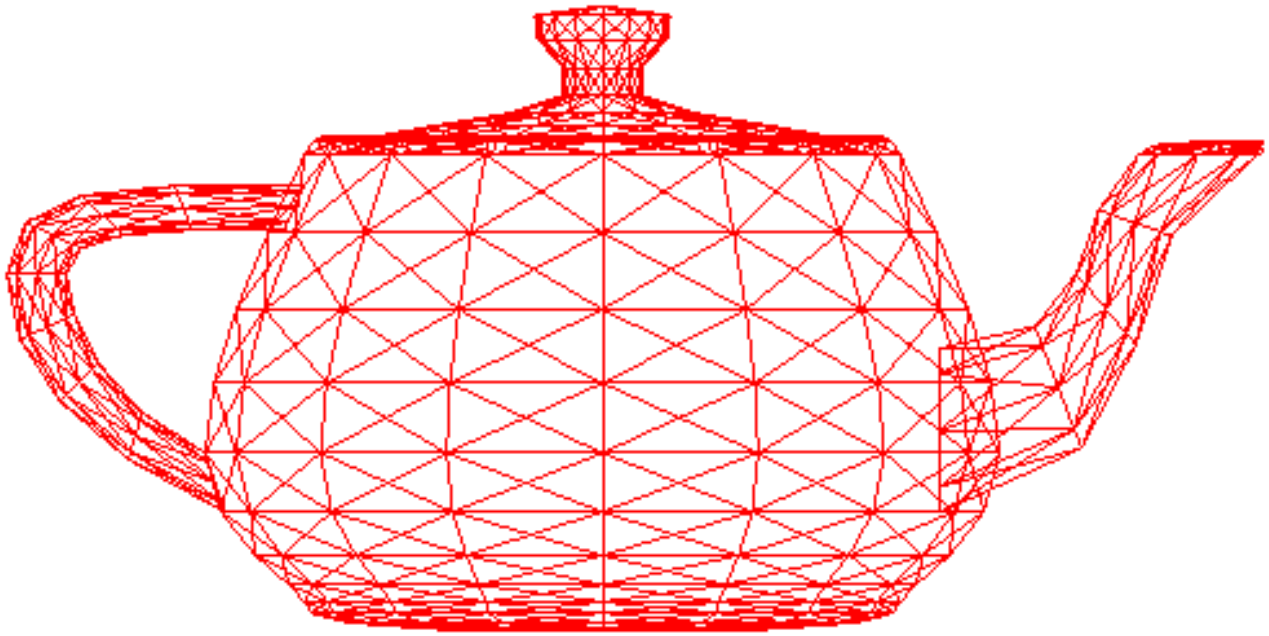
```
for ( var n = 0; n < numTeapotPatches; n++ ) {
   var data = new Array(numDivisions+1);
   for(var j = 0; j<= numDivisions; j++) data[j] = new Array(numDivisions+1);
   for(var i=0; i<=numDivisions; i++) for(var j=0; j<= numDivisions; j++) {
        data[i][j] = vec4(0,0,0,1);
      var u = i*h;
      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



http://www.cs.unm.edu/~angel/WebGL/7E/11/teapot1.html

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;
}
```

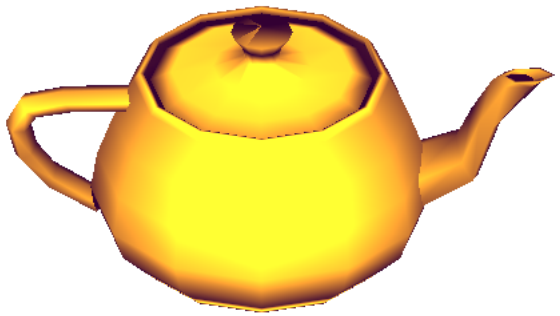
Vertex Shader

```
<script id="vertex-shader" type="x-
shader/x-vertex">

attribute  vec4 vPosition;

void main()
{
mat4 scale = mat4( 0.3, 0.0, 0.0, 0.0,
                   0.0,  0.3, 0.0, 0.0,
                   0.0,  0.0, 0.3, 0.0,
                   0.0,  0.0, 0.0, 1.0 );

   gl_Position = scale*vPosition;
}
</script>
```

Fragment Shader

```
<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

void main()
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);;
}
</script>
```

Adding Shading



http://www.cs.unm.edu/~angel/WebGL/7E/11/teapot4.html

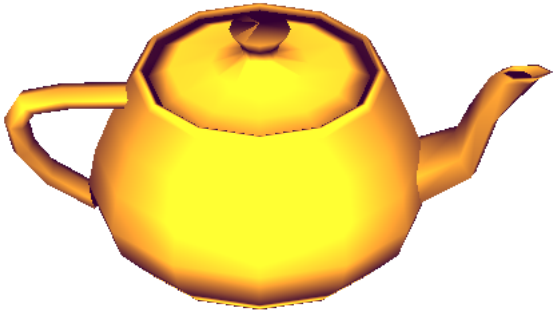
Face Normals - http://www.cs.unm.edu/~angel/WebGL/7E/11/teapot5.html

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;
}
```



Vertex Shader

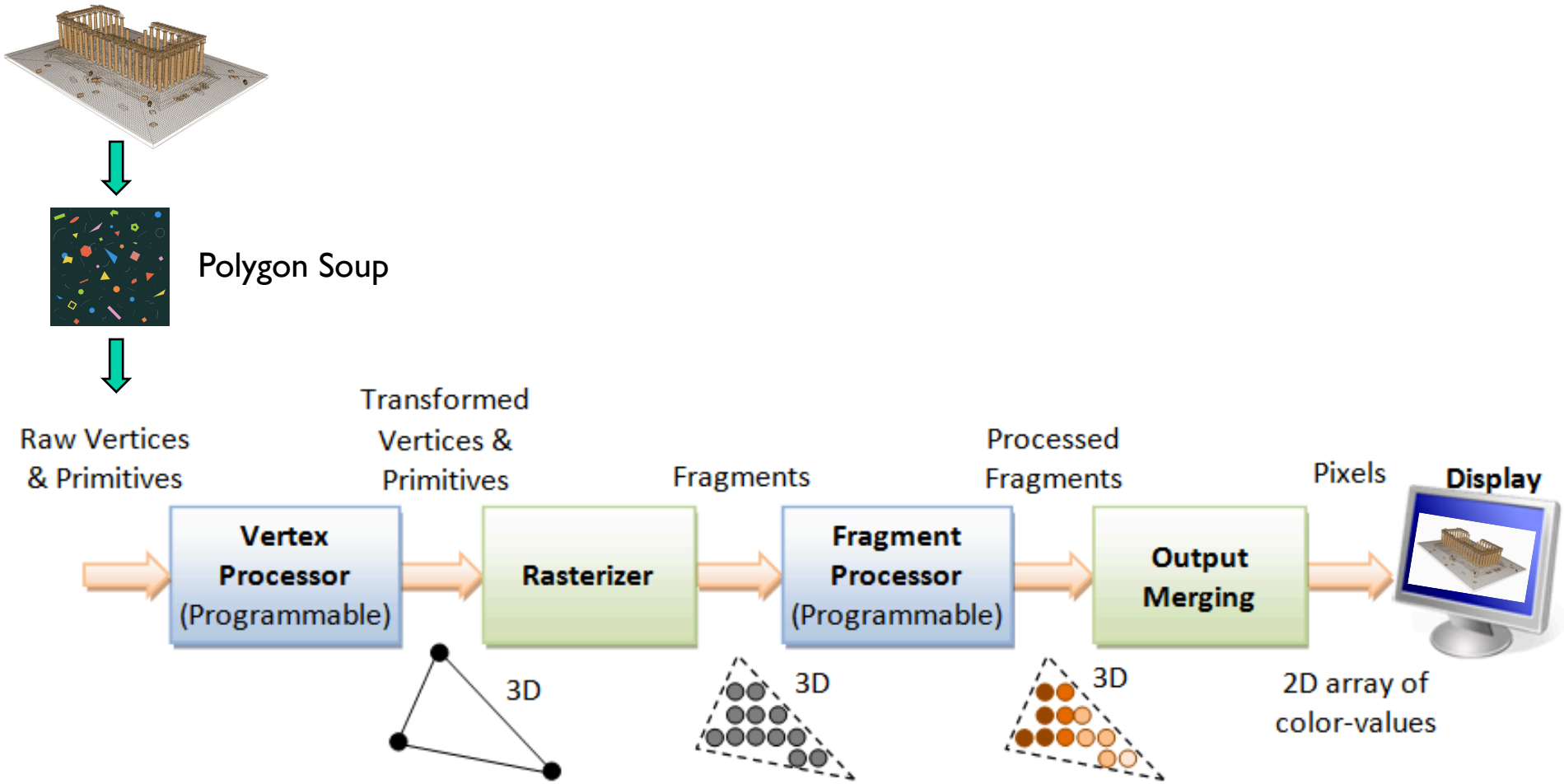
```
<script id="vertex-shader" type="x-shader/x-vertex”>
attribute  vec4 vPosition; attribute  vec4 vNormal; varying vec4 fColor;
uniform vec4 ambientProduct, diffuseProduct, specularProduct;
uniform mat4 modelViewMatrix; uniform mat4 projectionMatrix; uniform vec4 lightPosition;
uniform float shininess; uniform mat3 normalMatrix;
void main()
{   vec3 pos = (modelViewMatrix * vPosition).xyz;
    vec3 light = lightPosition.xyz; vec3 L = normalize( light - pos );
    vec3 E = normalize( -pos );   vec3 H = normalize( L + E );
    // Transform vertex normal into eye coordinates
    vec3 N = normalize( normalMatrix*vNormal.xyz);
    // Compute terms in the illumination equation
    vec4 ambient = ambientProduct;
    float Kd = max( dot(L, N), 0.0 );   vec4  diffuse = Kd*diffuseProduct;
    float Ks = pow( max(dot(N, H), 0.0), shininess ); vec4  specular = Ks * specularProduct;
    if( dot(L, N) < 0.0 ) {
                specular = vec4(0.0, 0.0, 0.0, 1.0);
    }
    gl_Position = projectionMatrix * modelViewMatrix * vPosition;
    fColor = ambient + diffuse +specular;
    fColor.a = 1.0;
} </script>
```

Fragment Shader

```
<script id="fragment-shader" type="x-shader/x-
fragment">

precision mediump float;

varying vec4 fColor;

void
main()
{
    gl_FragColor = fColor;

}
</script>
```

Post Geometry Shaders

Pipeline


Polygon Soup
Raw Vertices & Primitives
Vertex Processor (Programmable)
Transformed Vertices & Primitives
3D
Rasterizer
Fragments
3D
Fragment Processor (Programmable)
Processed Fragments
3D
Output Merging
Pixels
2D array of color-values
Display

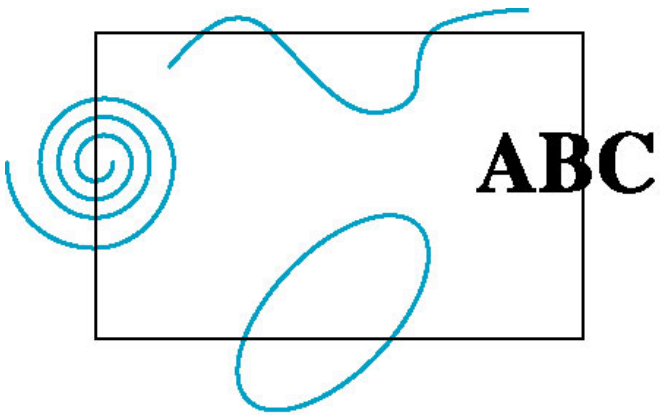
Pipeline


Vertices
Vertex Processor
Clipper and Primitive Assembler
Rasterizer
Fragment Processor
Pixels

Topics

- Clipping
- Scan conversion

Clipping

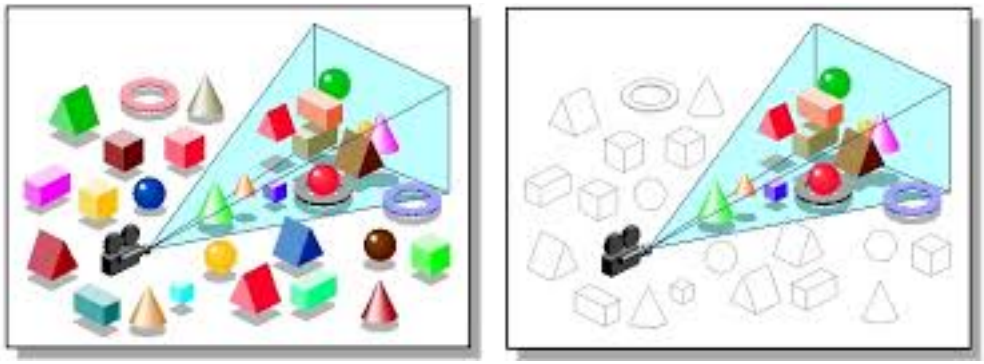


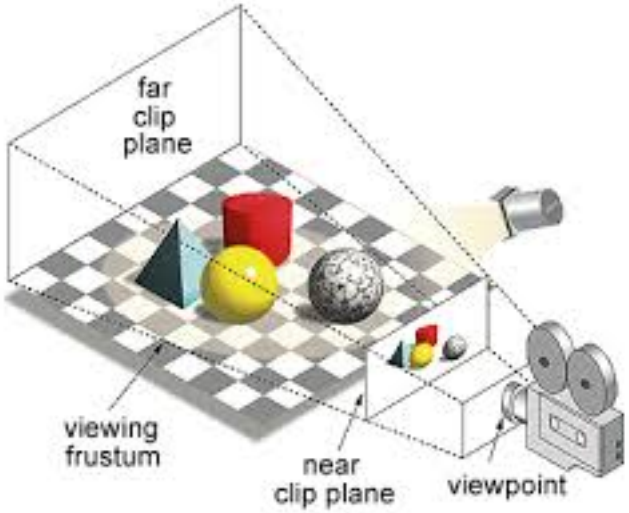
Clipping

- After geometric stage
  - vertices assembled into primitives

- Must clip primitives that are outside view frustum

Clipping

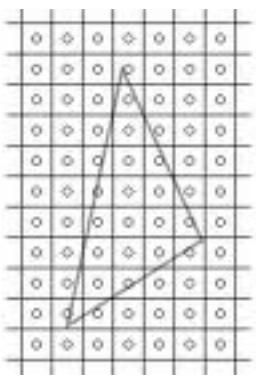


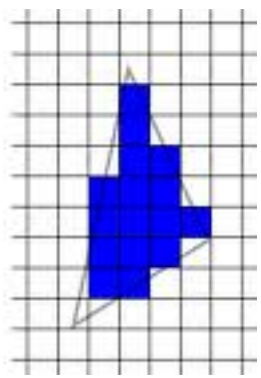


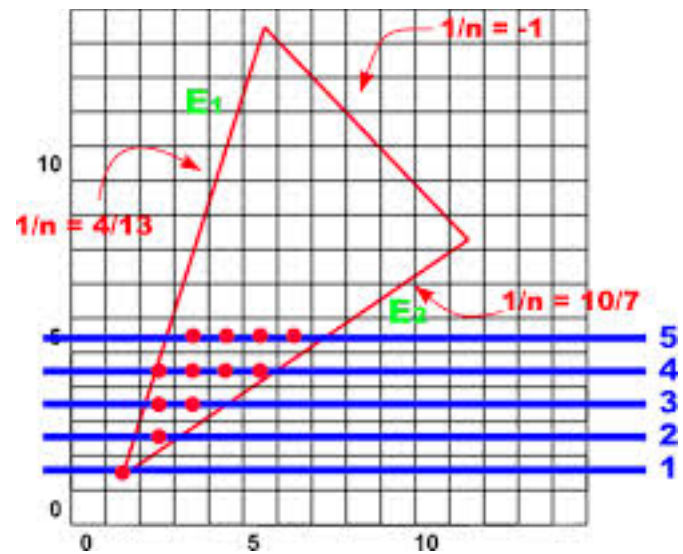
Scan Conversion

Which pixels can be affected by each primitive

- Fragment generation
- Rasterization or scan conversion







Additional Tasks

Some tasks deferred until fragment processing

- Hidden surface removal
- Antialiasing

Clipping

Contexts

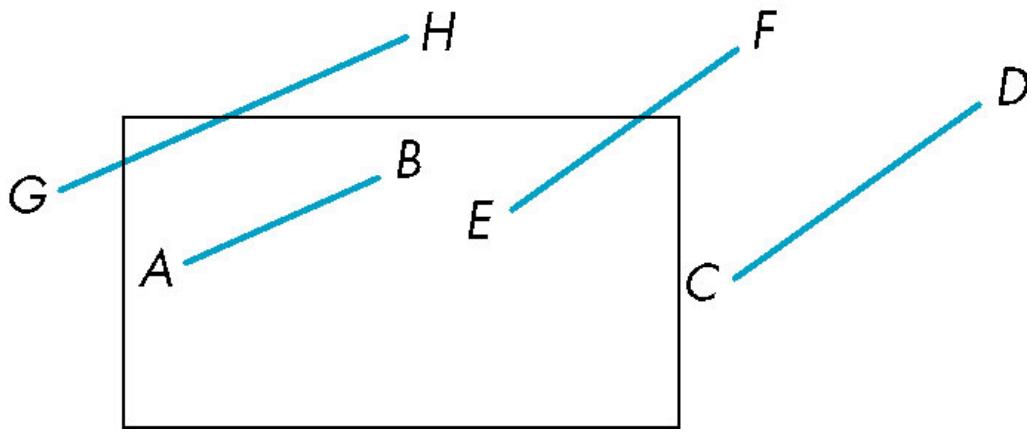
- 2D against clipping window

- 3D against clipping volume

2D Line Segments

Brute force:

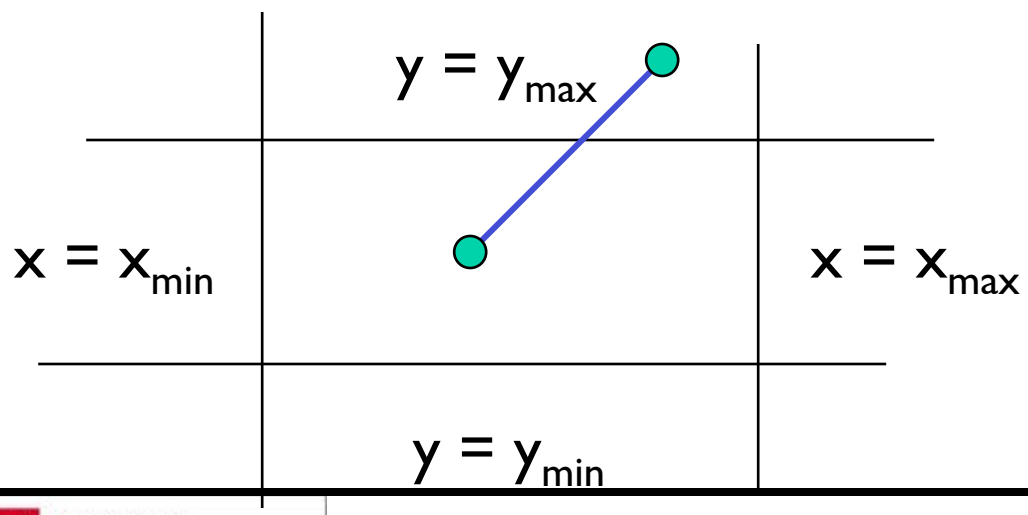
- compute intersections with all sides of clipping window
- Inefficient



Cohen-Sutherland Algorithm

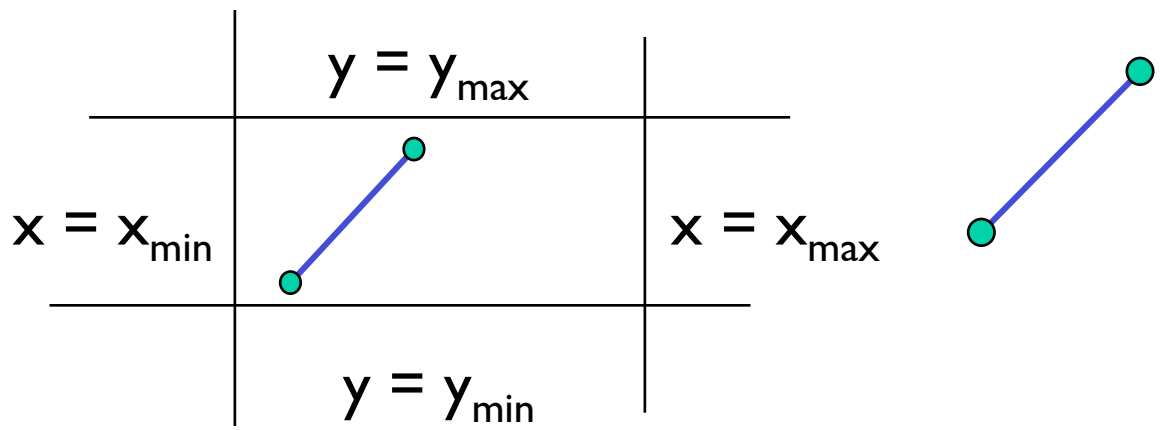
- Eliminate cases without computing intersections

- Start with four lines of clipping window



The Cases

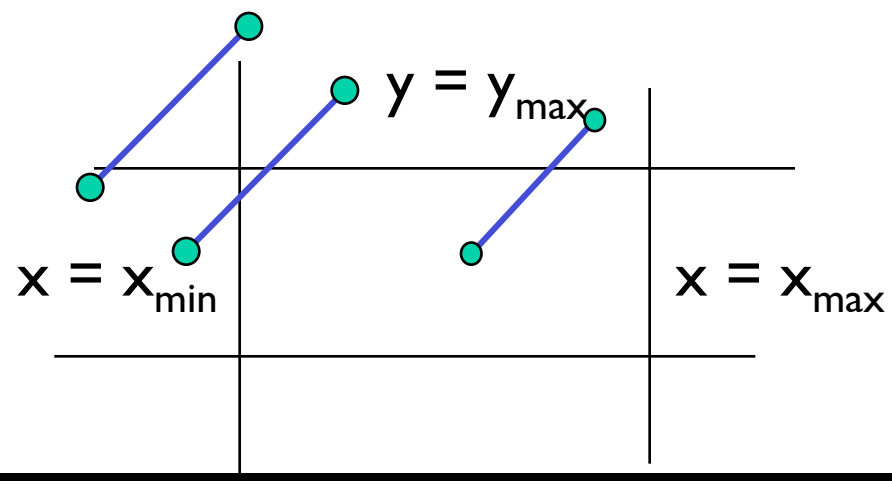
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

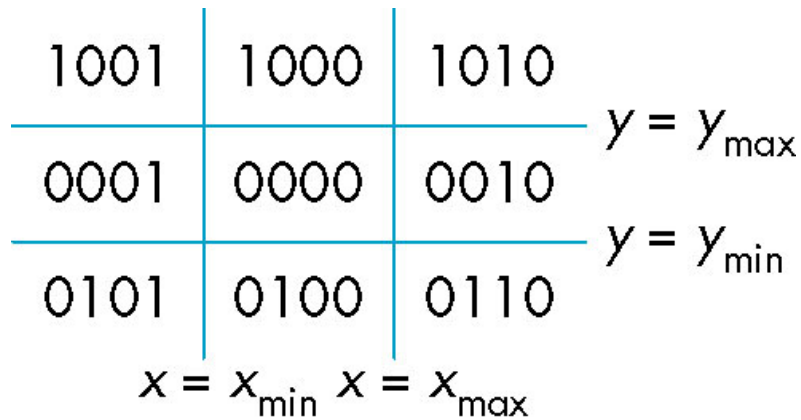
- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside
  - May have part inside
  - Must do at least one intersection



Defining Outcodes

- For each endpoint, define an outcode $b_0b_1b_2b_3$

$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions

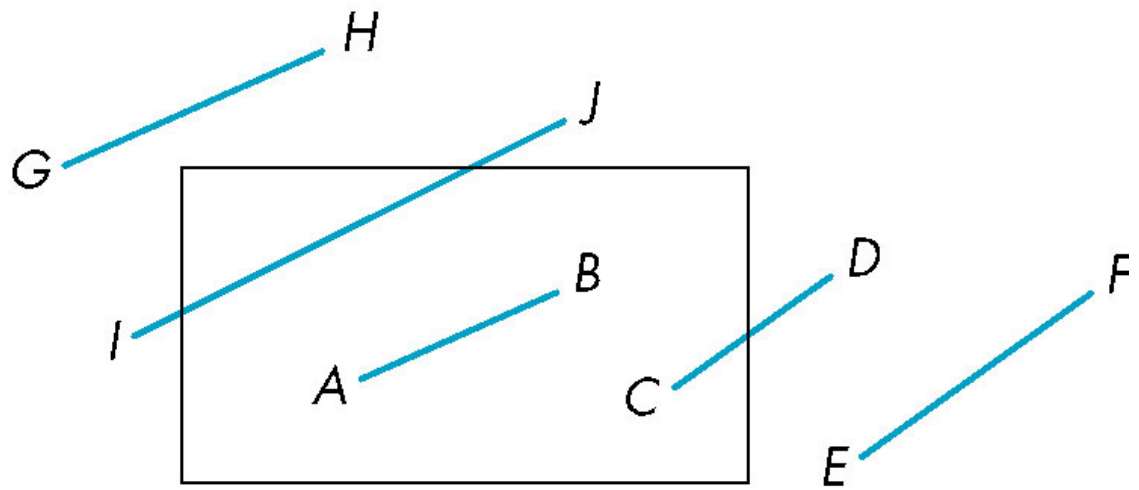
- Computation of outcode requires at most 4 comparisons

Using Outcodes

Consider the 5 cases below

AB: outcode(A) = outcode(B) = 0

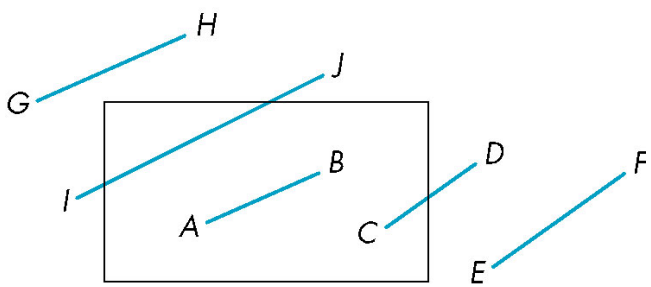
– Accept line segment



Using Outcodes

CD: outcode (C) = 0, outcode(D) ≠ 0

- Compute intersection
- Location of 1 in outcode(D) marks edge to intersect with



1001	1000	1010
0001	0000	0010
0101	0100	0110

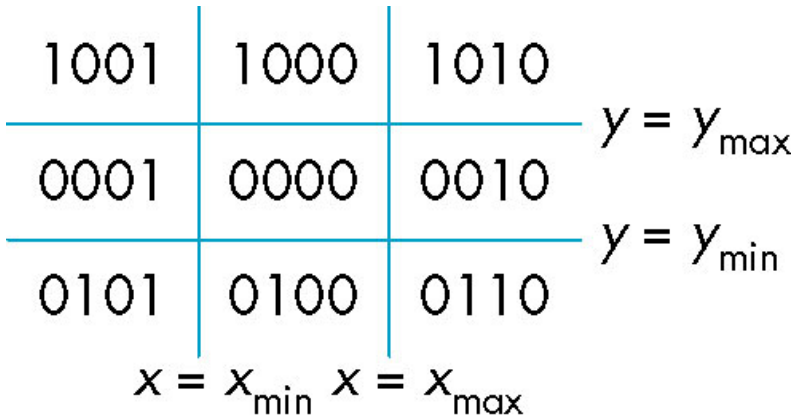
$y = y_{max}$

$y = y_{min}$

$x = x_{min}$ $x = x_{max}$

Using Outcodes

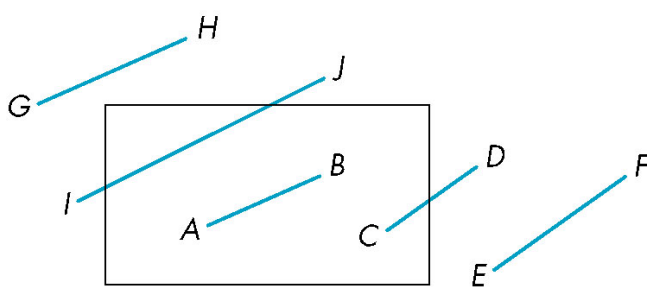
If there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two intersections



Using Outcodes

EF: outcode(E) logically ANDed with outcode(F) (bitwise) ≠ 0

- Both outcodes have a 1 bit in the same place
- Line segment is outside clipping window
- reject



1001	1000	1010
0001	0000	0010
0101	0100	0110

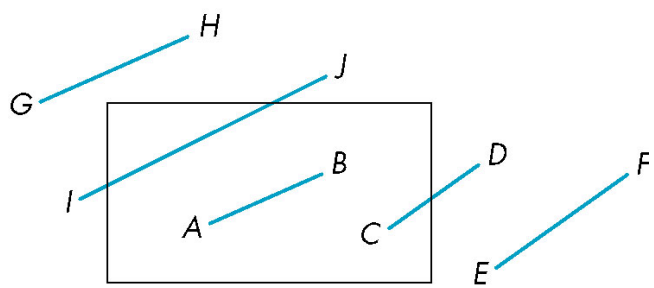
$y = y_{max}$

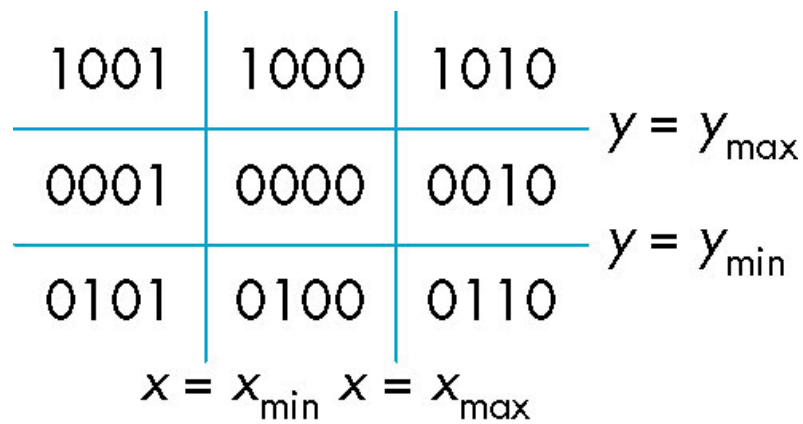
$y = y_{min}$

$x = x_{min}$ $x = x_{max}$

Using Outcodes

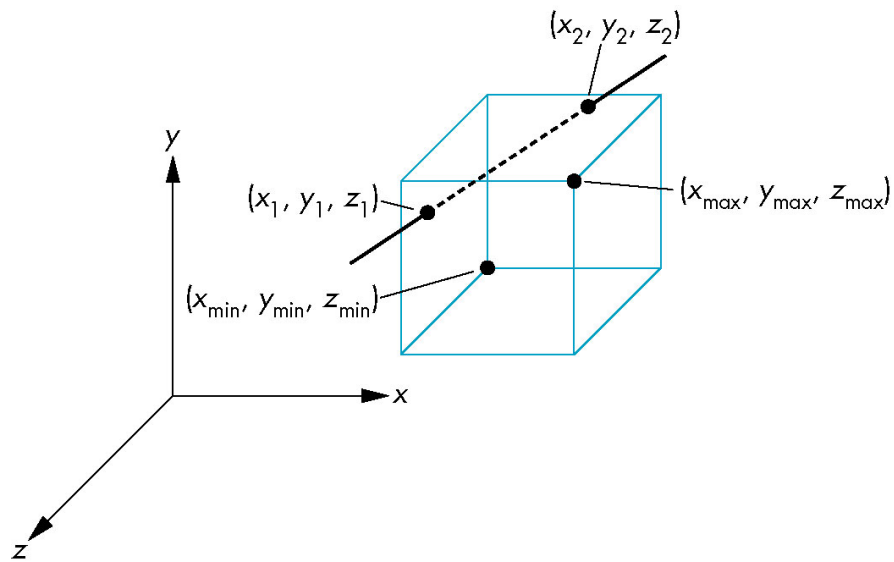
- GH and IJ
  - same outcodes, neither zero but logical AND yields zero
- Shorten line by intersecting with sides of window
- Compute outcode of intersection
  - new endpoint of shortened line segment
- Recurse algorithm

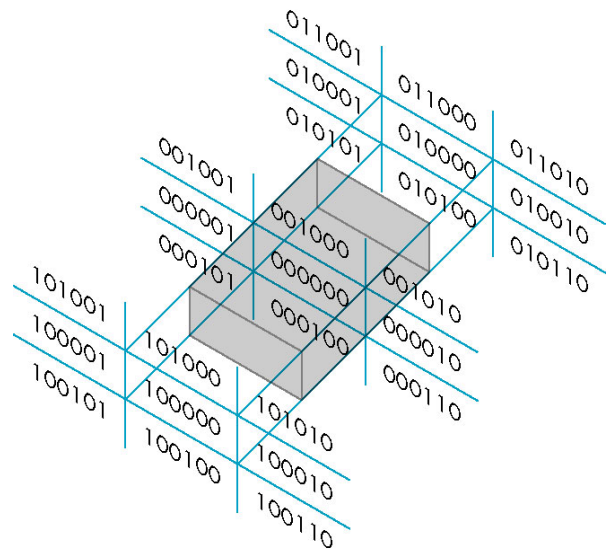




Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes

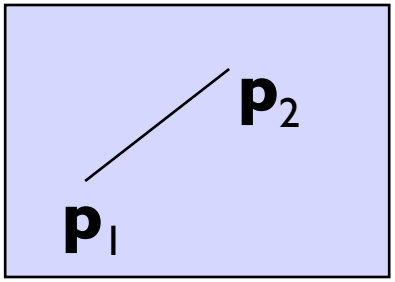




Liang-Barsky Clipping

Consider parametric form of a line segment

$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2 \quad 1 \geq \alpha \geq 0$$

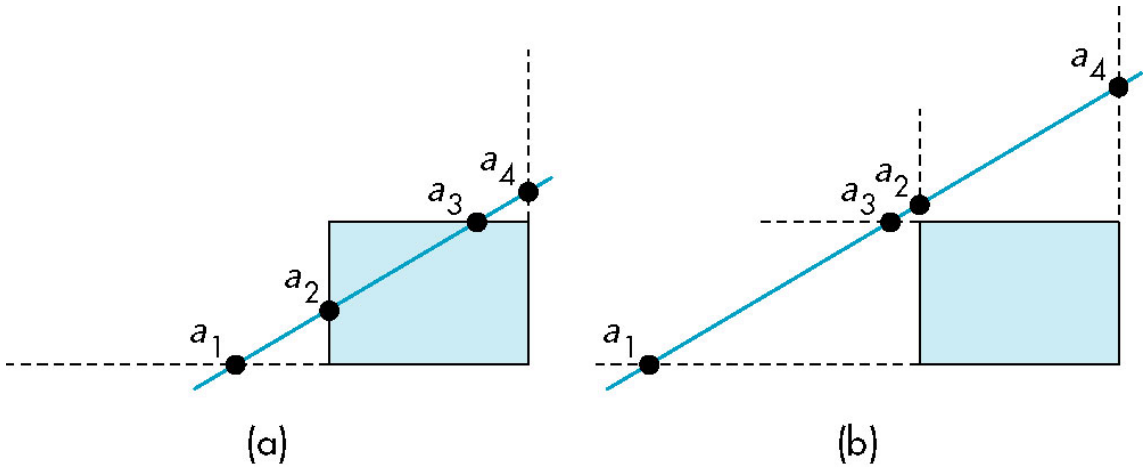


Intersect with parallel slabs –

- Pair for Y
- Pair for X
- Pair for Z

Liang-Barsky Clipping

- In (a): $a_4 > a_3 > a_2 > a_1$
  - Intersect right, top, left, bottom: shorten
- In (b): $a_4 > a_2 > a_3 > a_1$
  - Intersect right, left, top, bottom: reject



Advantages

- Can accept/reject as easily as with Cohen-Sutherland
- Using values of α, we do not have to use algorithm recursively as with C-S
- Extends to 3D

Polygon Clipping

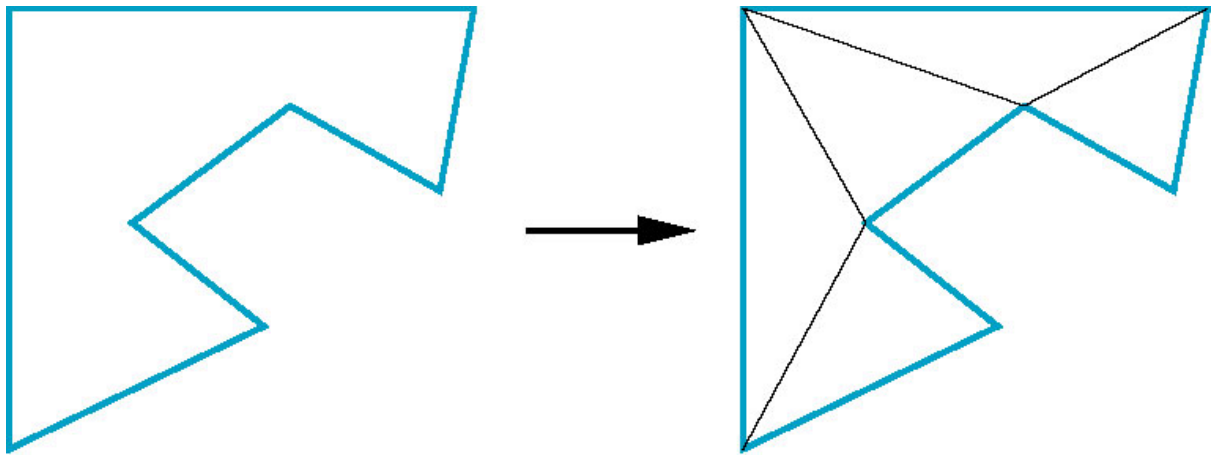
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons

- Convex polygon is cool ☺

Fixes

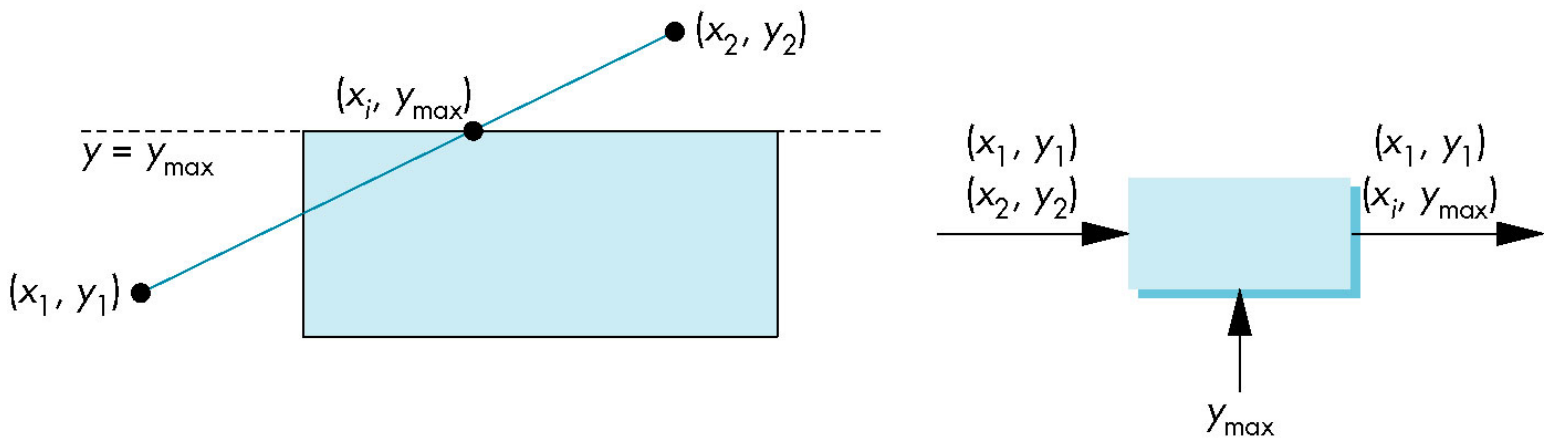
Tessellation and Convexity

Replace nonconvex (*concave*) polygons with triangular polygons (a *tessellation*)



Clipping as a Black Box

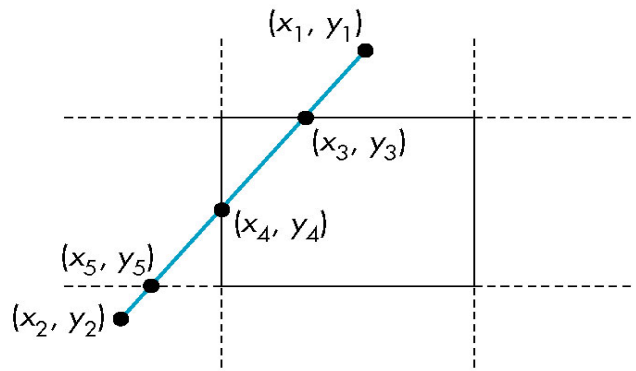
Line segment clipping - takes in two vertices and produces either no vertices or vertices of a clipped segment

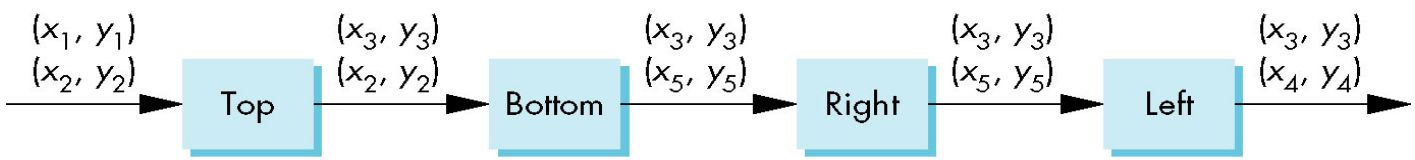


Pipeline Clipping - Line Segments

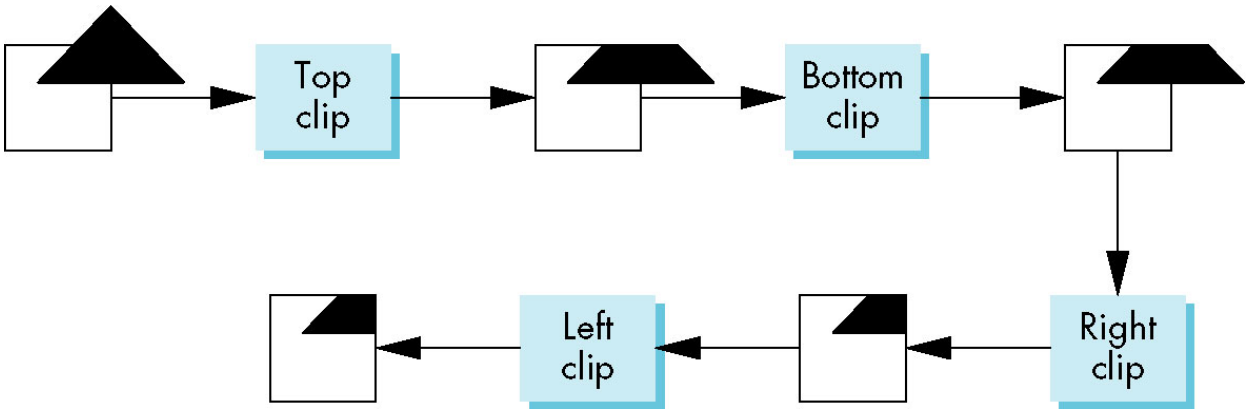
Clipping side of window is independent of other sides

– Can use four independent clippers in a pipeline





Pipeline Clipping of Polygons

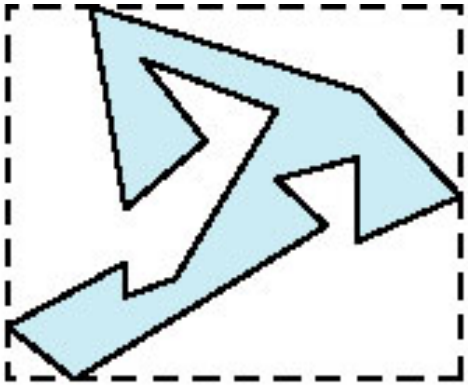


- Three dimensions: add front and back clippers
- Small increase in latency

Bounding Boxes

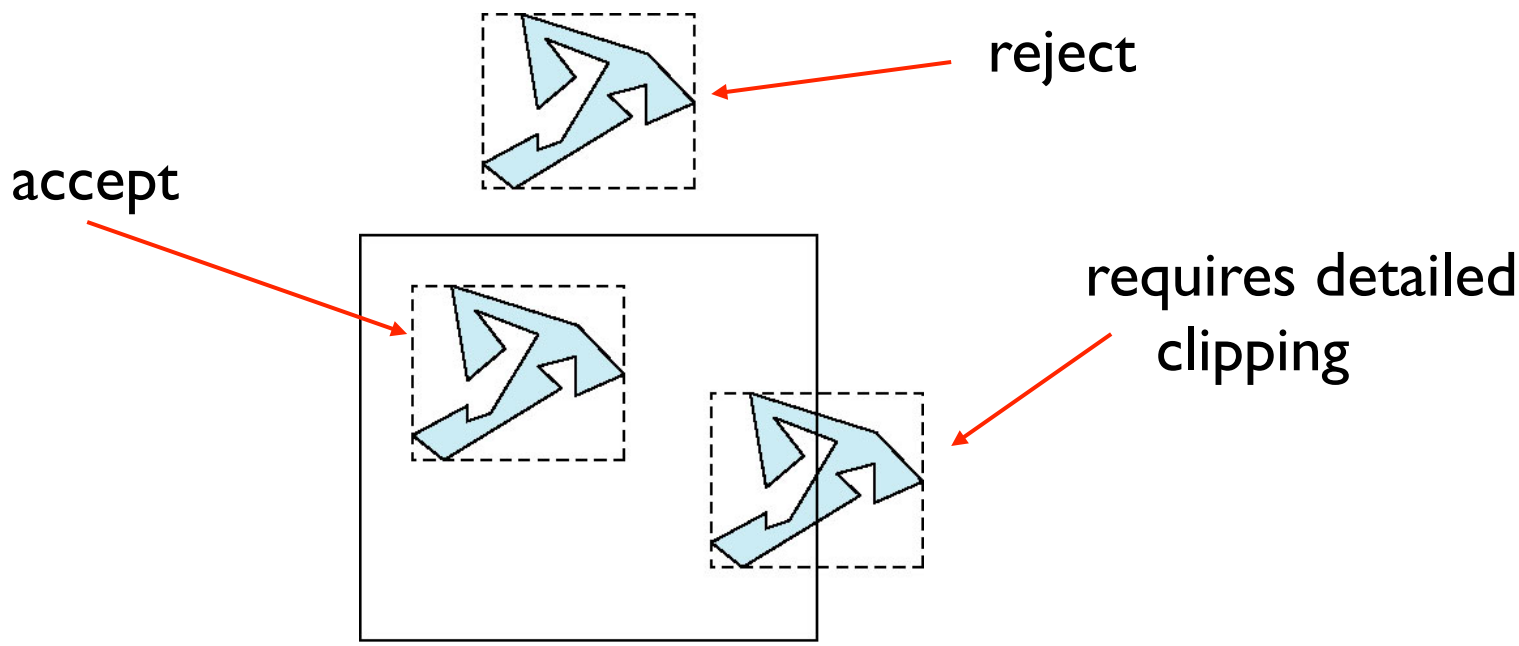
Ue an *axis-aligned bounding box* or *extent*

- Smallest rectangle aligned with axes that encloses the polygon
- Simple to compute: max and min of x and y



Bounding boxes

Can usually determine accept/reject based only on bounding box

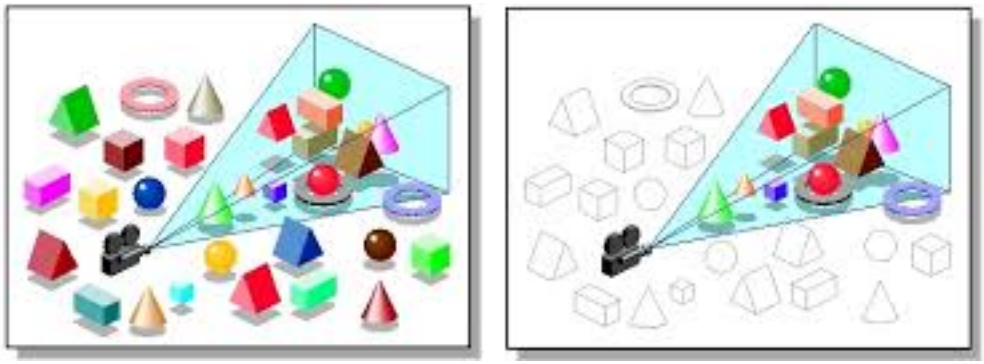


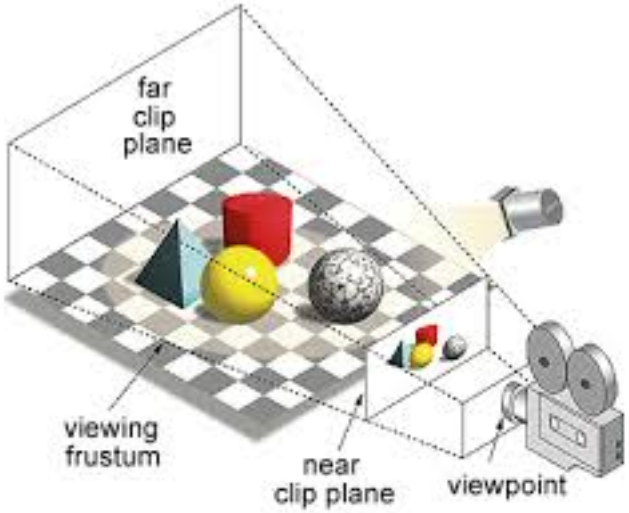
Clipping vs. Visibility

- Clipping similar to hidden-surface removal
- Remove objects that are not visible to the camera
- Use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

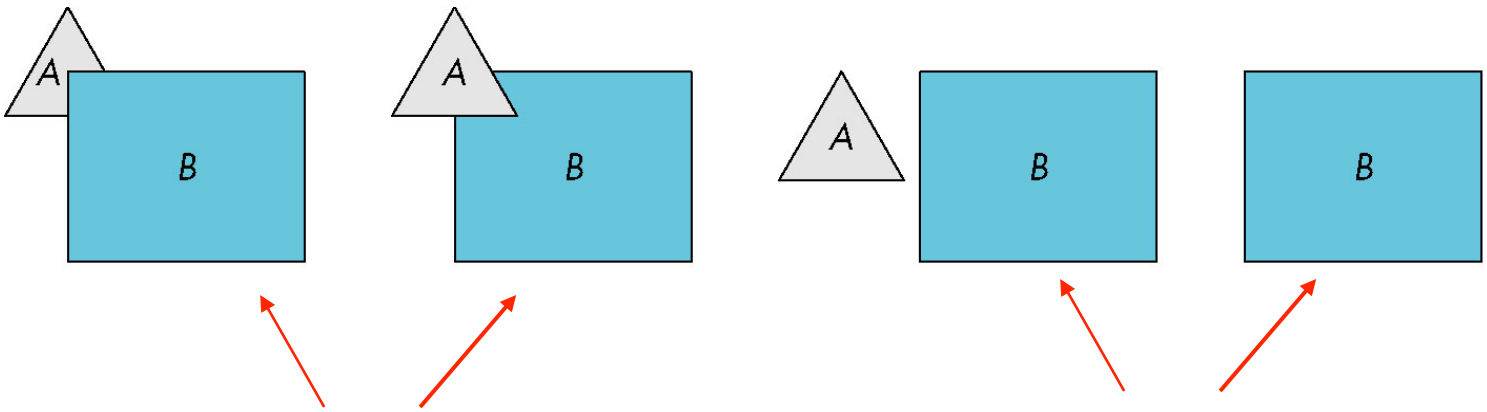
Clipping





Hidden Surface Removal

Object-space approach: use pairwise testing between polygons (objects)

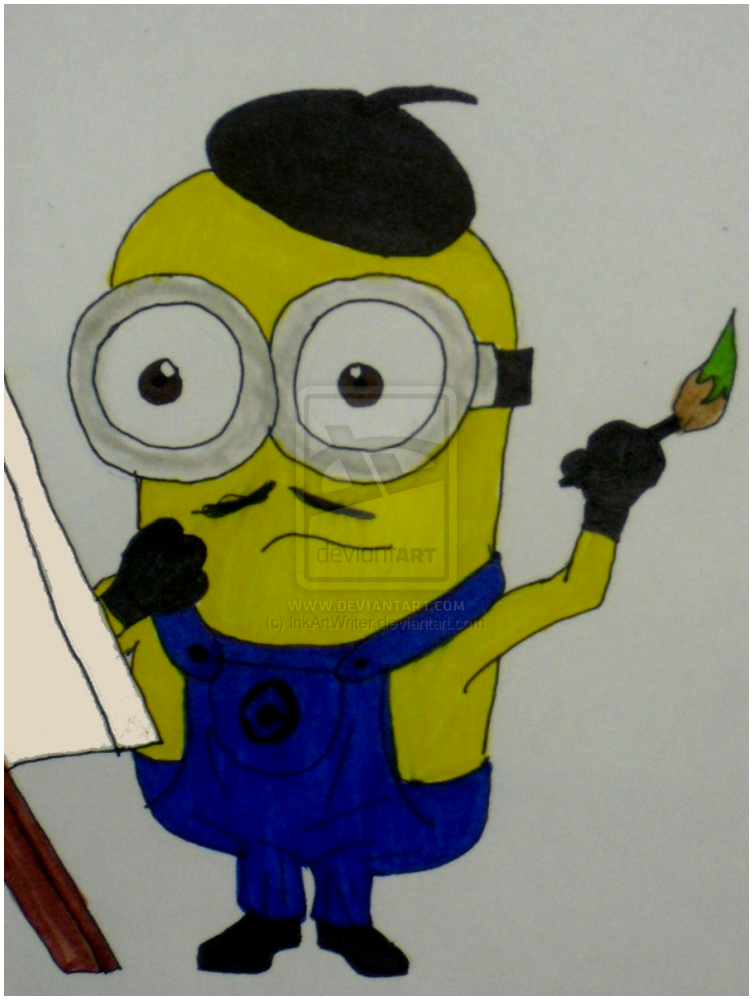


partially obscuring

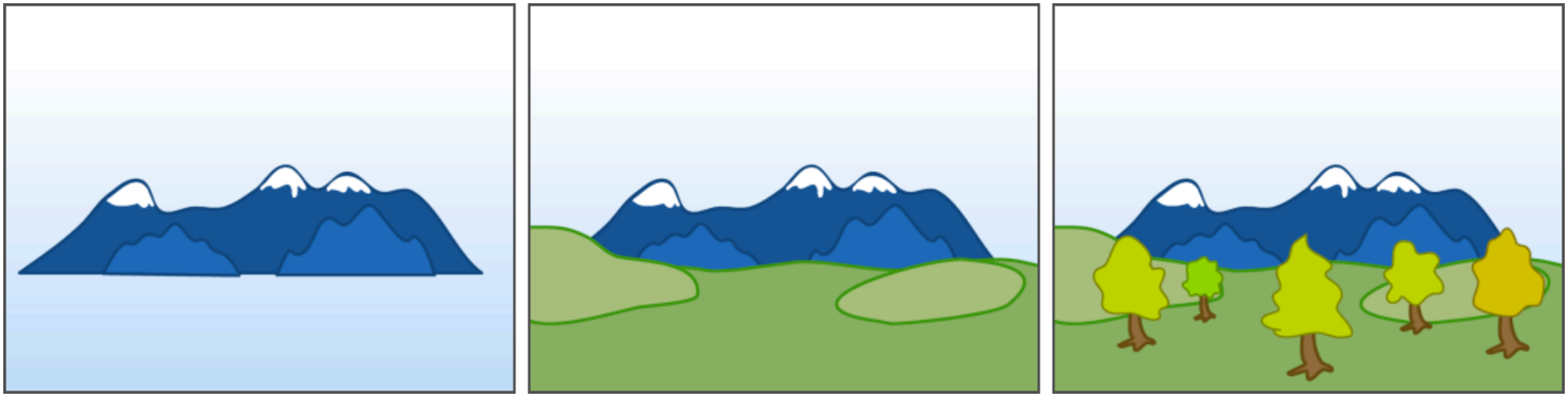
can draw independently

Worst case complexity $O(n^2)$ for n polygons

Better Still

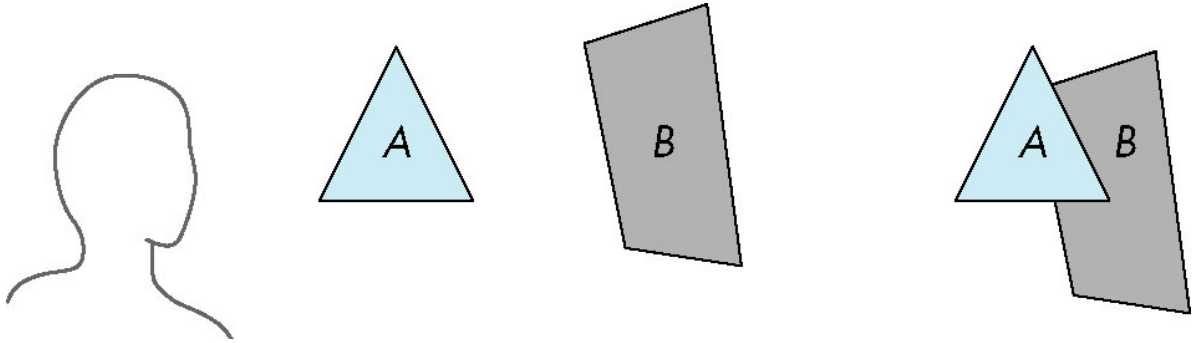


Better Still



Painter's Algorithm

Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

Fill B then A

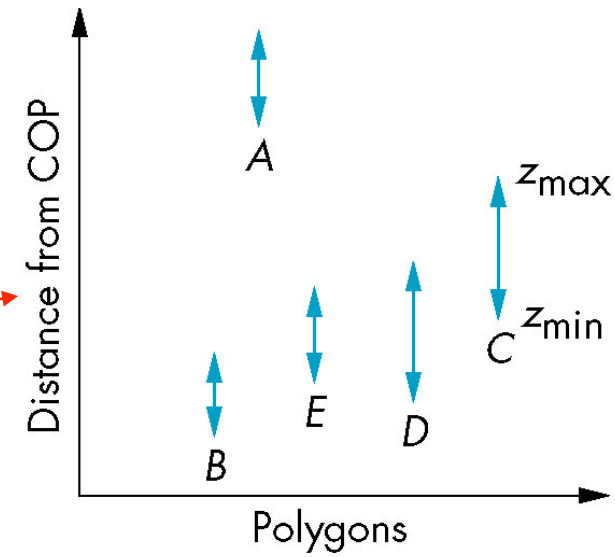
Depth Sort

Requires ordering of polygons first

- O(n log n) calculation for ordering
- Not all polygons front or behind all other polygons

Order polygons and deal with easy cases first, harder later

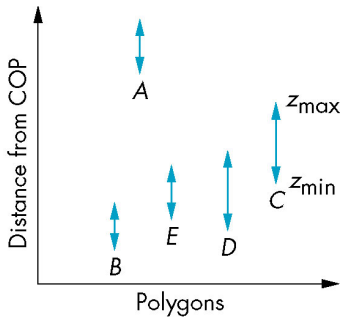
Polygons sorted by distance from COP



Easy Cases

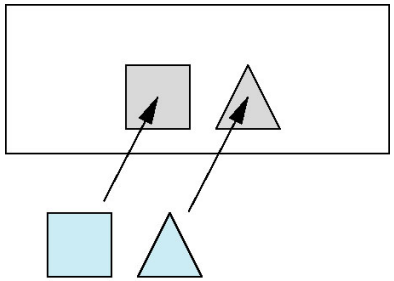
A lies behind all other polygons

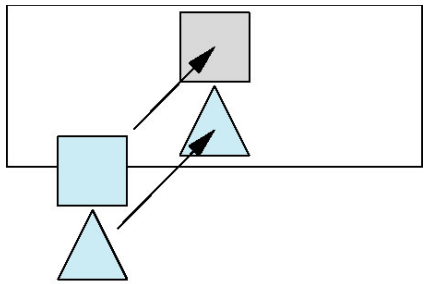
– Can render



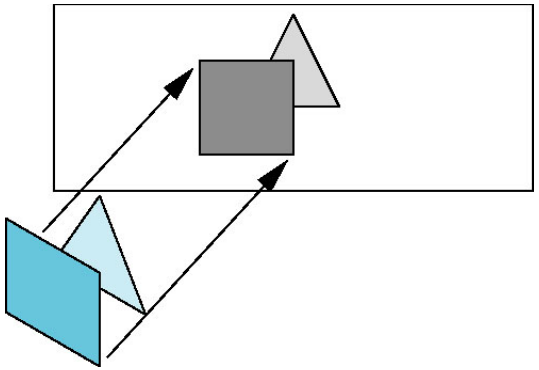
Polygons overlap in z but not in either x or y

– Can render independently

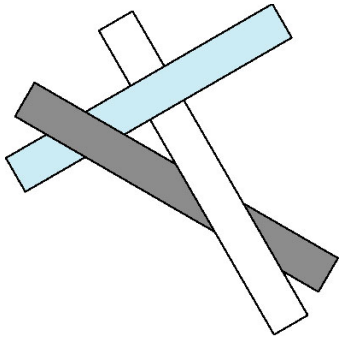




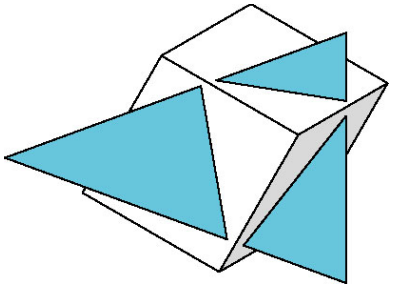
Hard Cases



Overlap in all directions but can one is fully on one side of the other



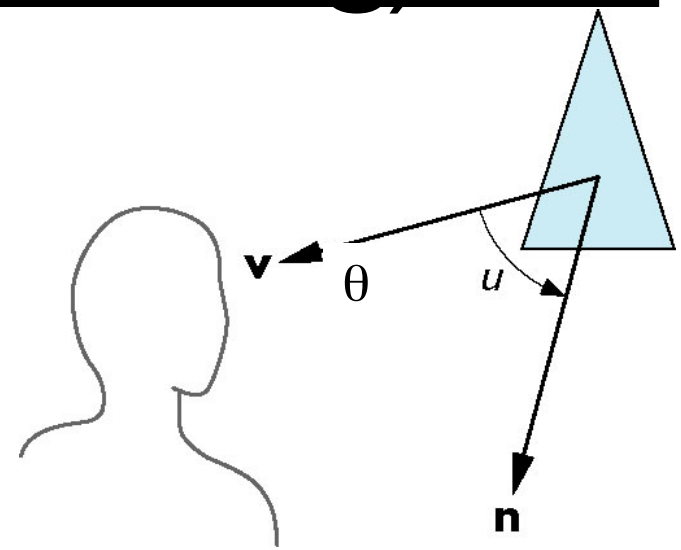
cyclic overlap



penetration

Back-Face Removal (Culling)

face is visible iff $90 \geq \theta \geq -90$
equivalently $\cos\theta \geq 0$
or $\mathbf{v} \cdot \mathbf{n} \geq 0$



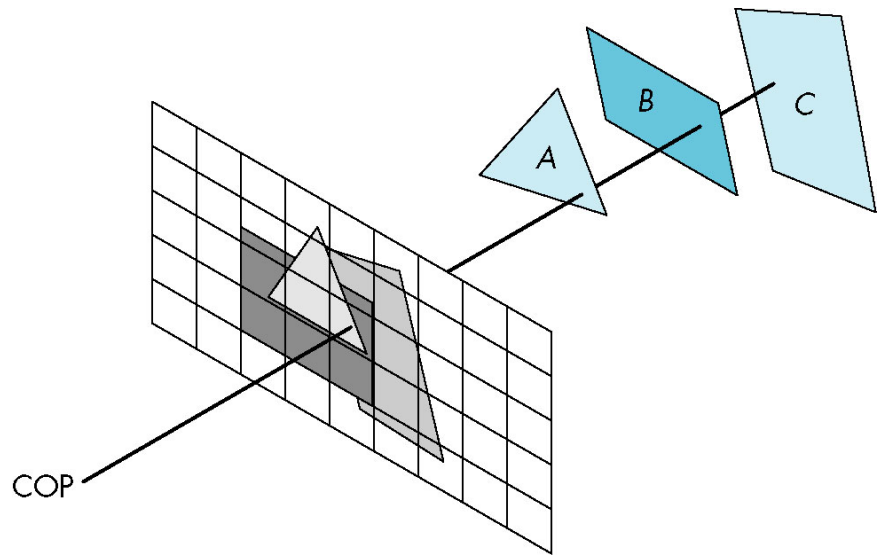
- plane of face has form ax + by +cz +d =0
- After normalization $\mathbf{n} = (0\ 0\ 1\ 0)^T$

+ Need only test the sign of c

- Will not work correctly if we have nonconvex objects

Image Space Approach

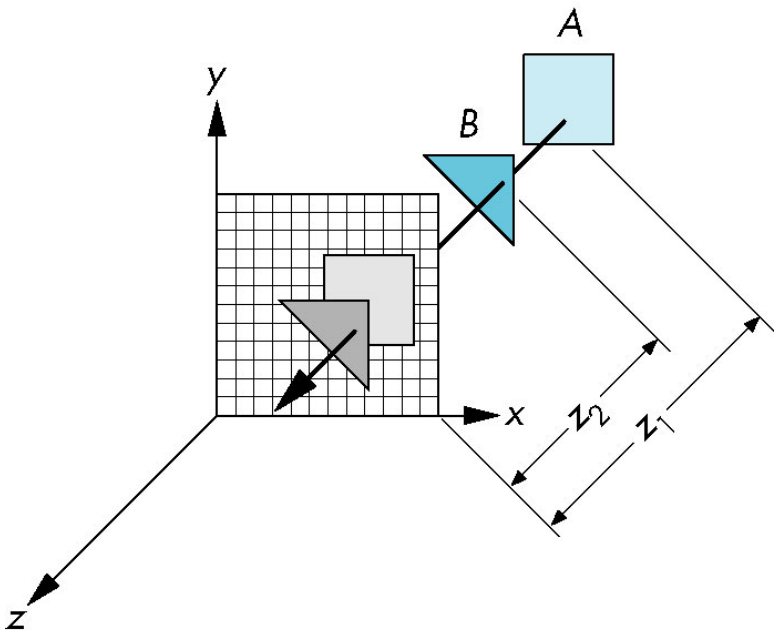
- Look at each ray (nm for an n x m frame buffer)
- Find closest of k polygons
- Complexity O(nmk)
- Ray tracing
- z-buffer



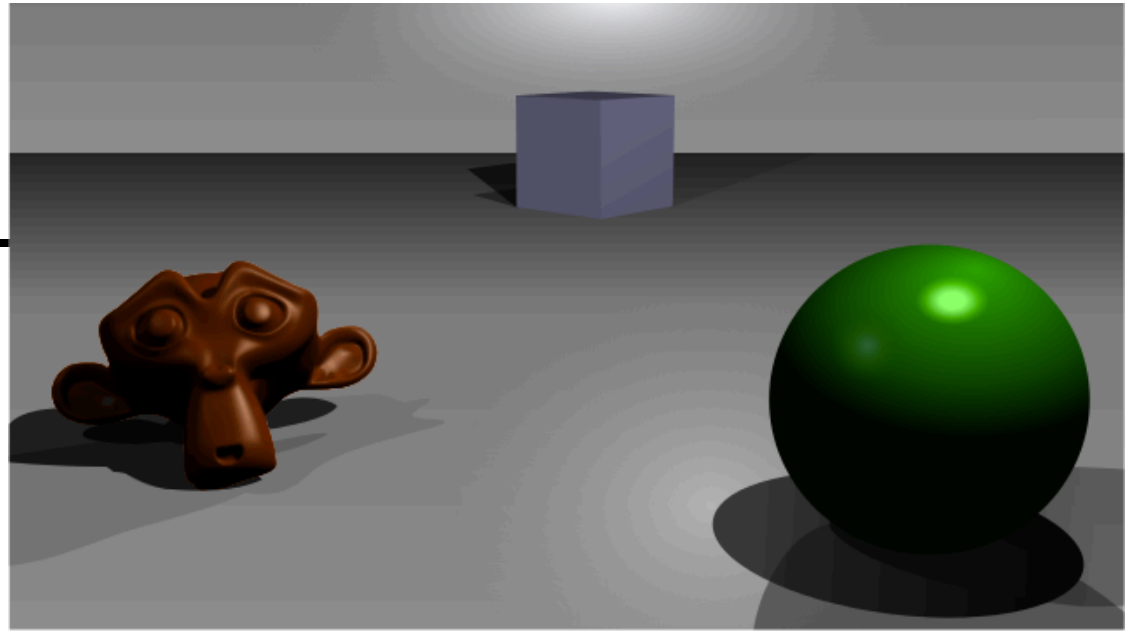
z-Buffer Algorithm

- Use a buffer called z or depth buffer to store depth of closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer

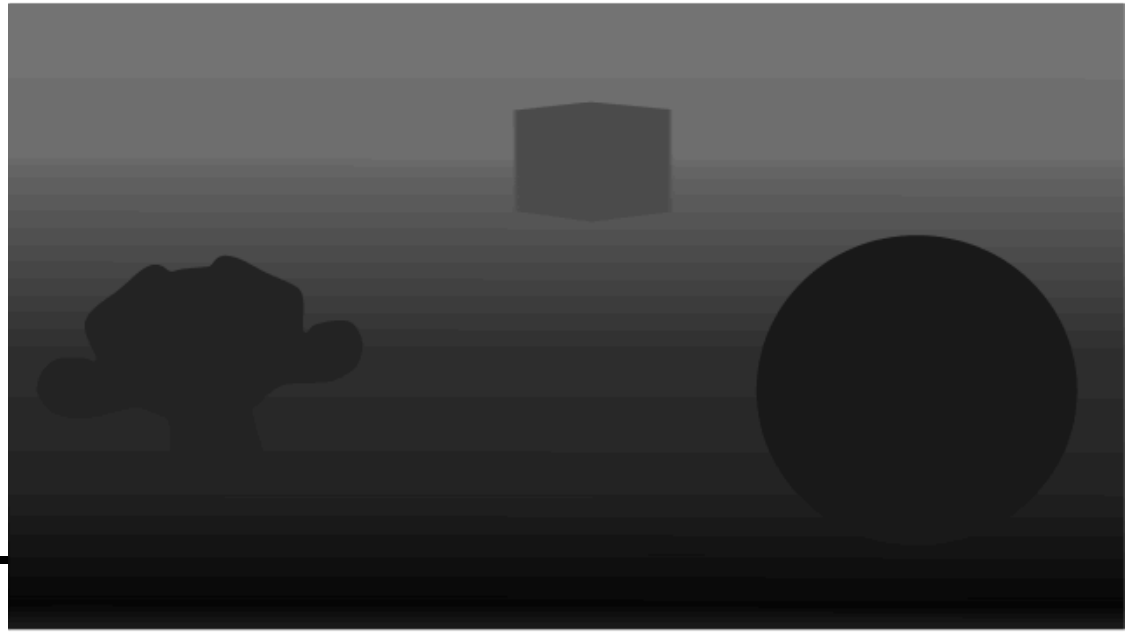
z-Buffer




```
for(each polygon P in the polygon list)
    do{
        for(each pixel(x,y) that intersects P)
        do{
            Calculate z-depth of P at (x,y)
            If (z-depth < z-buffer[x,y])
            then{
                z-buffer[x,y]=z-depth;
                COLOR(x,y)=Intensity of P at(x,y);
            }
            #If-programming-for alpha compositing:
            Else if (COLOR(x,y).opacity < 100%)
            then{
                COLOR(x,y)=Superimpose
COLOR(x,y) in front of Intensity of P at(x,y);
            }
            #Endif-programming-for
        }
    }
 display COLOR array.
```



A simple three-dimensional scene



Z-buffer representation

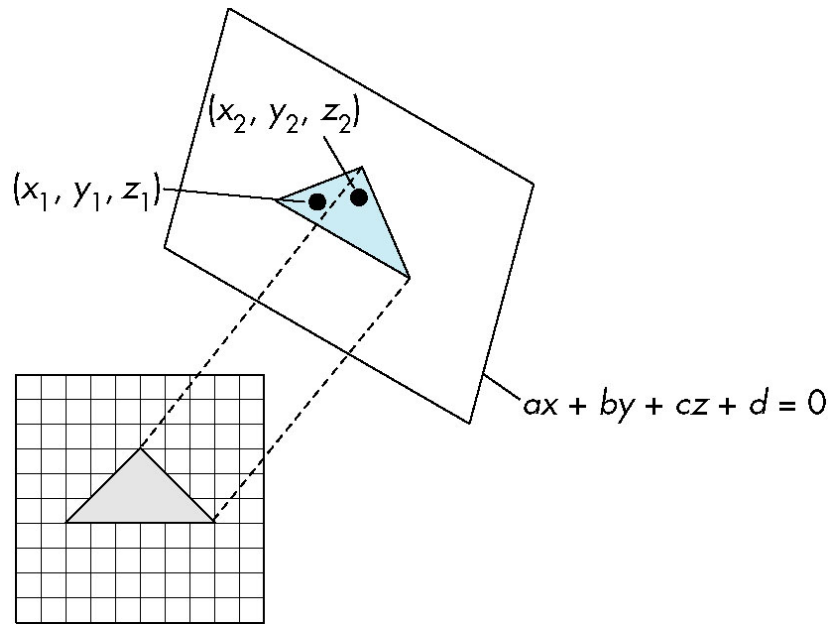
Efficiency - Scanline

As we move across a scan line, the depth changes satisfy $a\Delta x+b\Delta y+c\Delta z=0$

Along scan line

$\Delta y = 0$

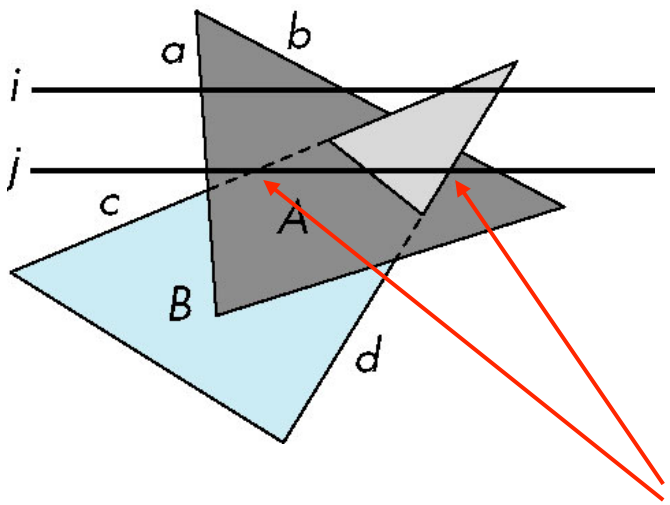
$\Delta z = -\frac{a}{c}\ \Delta x$



In screen space $\Delta x = 1$

Scan-Line Algorithm

Combine shading and hsr through scan line algorithm



scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Implementation

Need a data structure to store

- Flag for each polygon (inside/outside)
- Incremental structure for scan lines that stores which edges are encountered
- Parameters for planes

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Rasterization

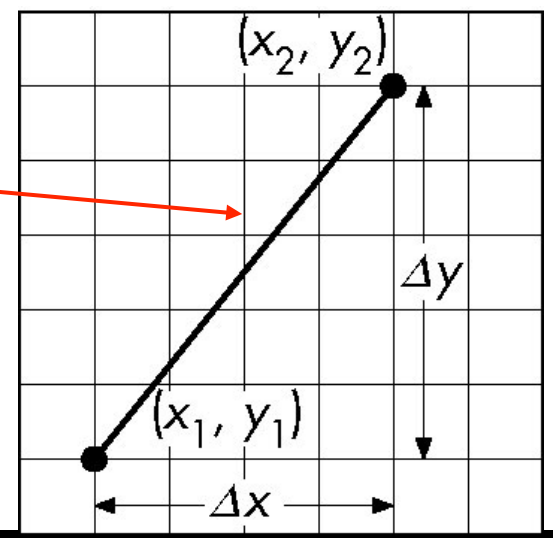
- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Scan-Line Rasterization

ScanConversion -Line Segments

- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **write_pixel** function

y = mx + h



$$m = \frac{\Delta y}{\Delta x}$$

DDA Algorithm

- Digital Differential Analyzer
  - Line y=mx+ h satisfies differential equation

    $dy/dx = m = Dy/Dx = y_2-y_1/x_2-x_1$

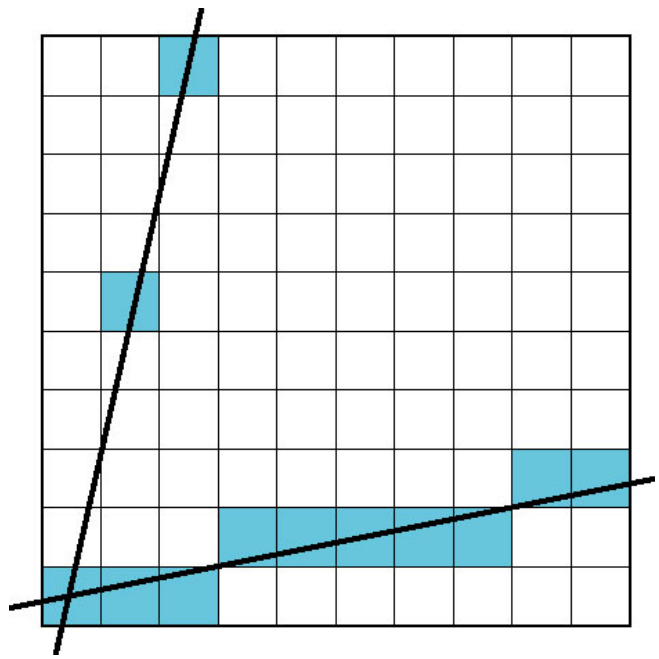
- Along scan line Dx = 1

```
For(x=x1; x<=x2,ix++) {
   y+=m;
 display (x, round(y), line_color)
}
```

Problem

DDA = for each x plot pixel at closest y

– Problems for steep lines

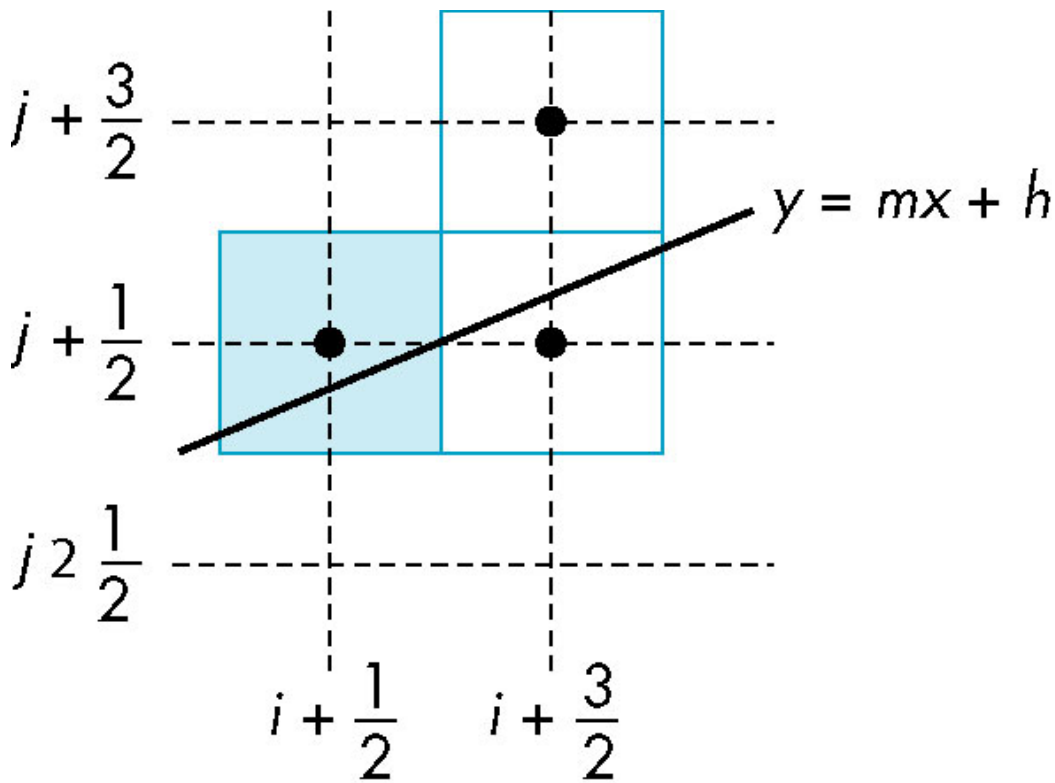


Bresenham's Algorithm

- DDA requires one floating point addition per step
- Eliminate computations through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers

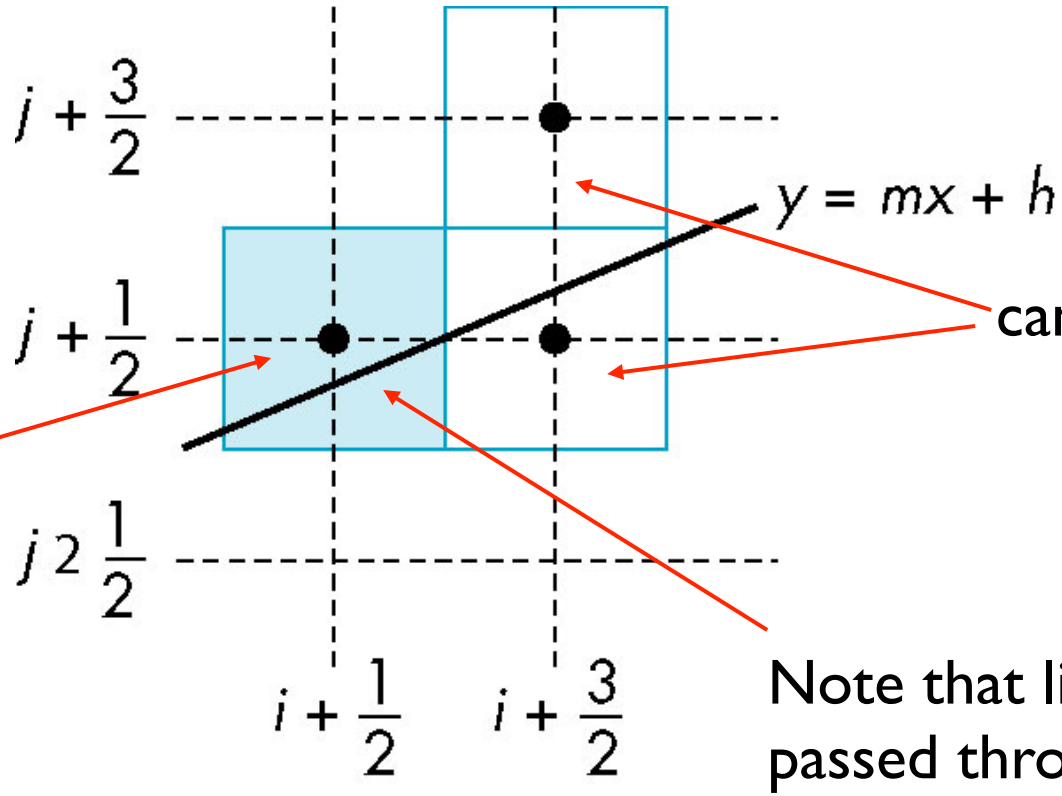
Main Premise

If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer



Candidate Pixels

$1 \geq m \geq 0$



candidates

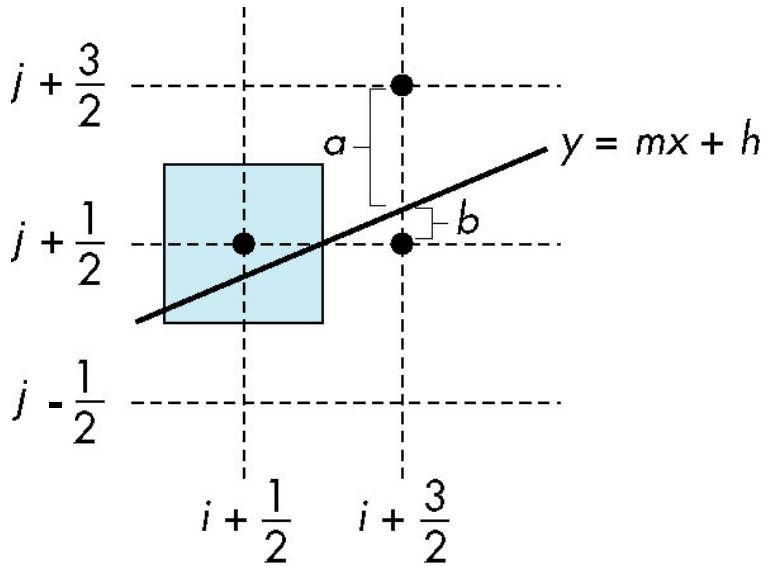
last pixel

Note that line could have passed through any part of this pixel

Decision Variable

$d = \Delta x(b-a)$

d is an integer
d > 0 use upper pixel
d < 0 use lower pixel



Incremental Form

Inspect d_k at $x = k$

$$d_{k+1} = d_k - 2Dy, \quad \text{if } d_k < 0$$
$$d_{k+1} = d_k - 2(Dy - Dx), \quad \text{otherwise}$$

For each x, we need do only an integer addition and test

Single instruction on graphics chips

Polygon Scan Conversion

- Scan Conversion = Fill
- How to tell inside from outside
  - Convex easy
  - Nonsimple difficult
  - Odd even test
    - Count edge crossings

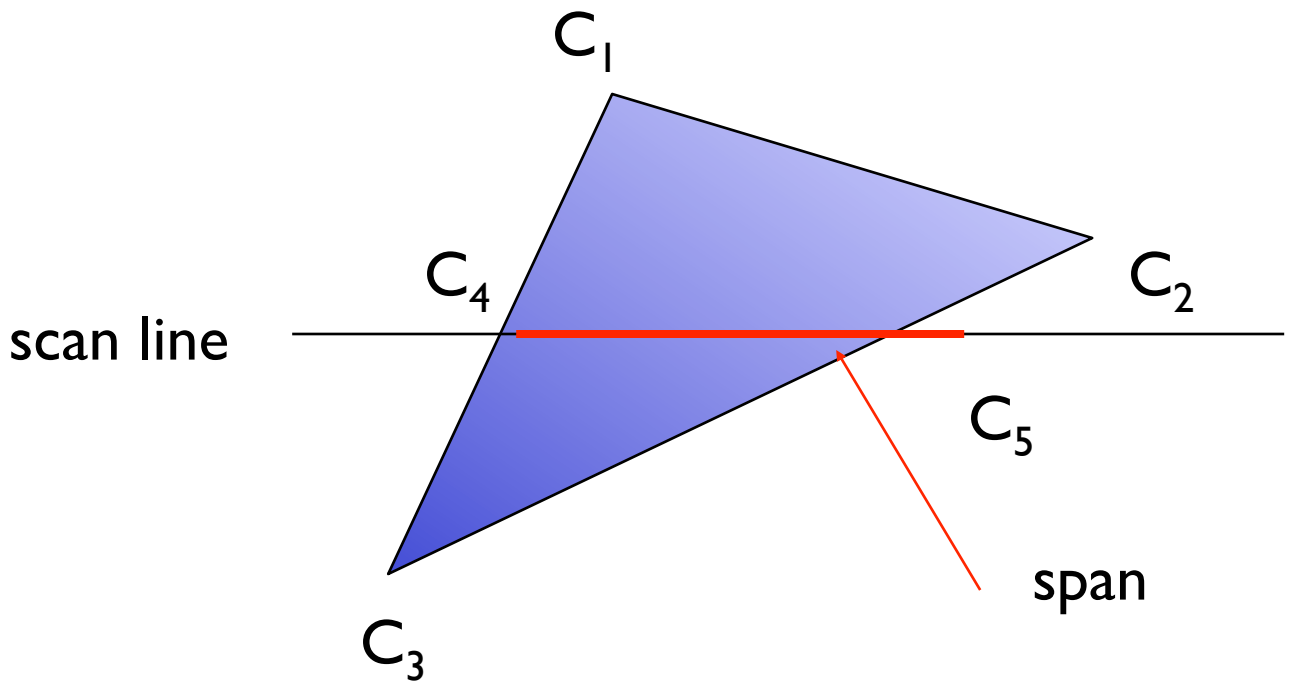
Filling in the Frame Buffer

Fill at end of pipeline

- Convex Polygons only
- Nonconvex polygons assumed to have been tessellated
- Shades (colors) have been computed for vertices (Gouraud shading)
- Combine with z-buffer algorithm
  - March across scan lines interpolating shades
  - Incremental work small

Using Interpolation

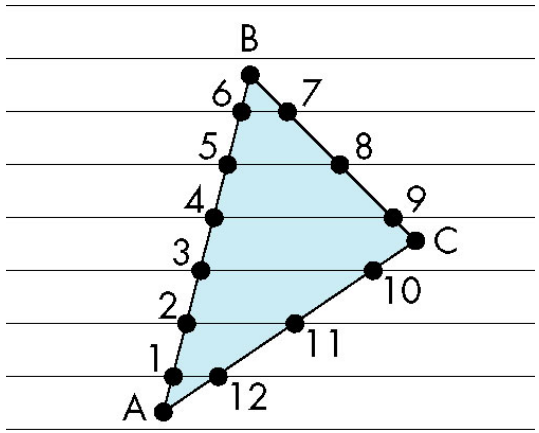
C_1 C_2 C_3 specified by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



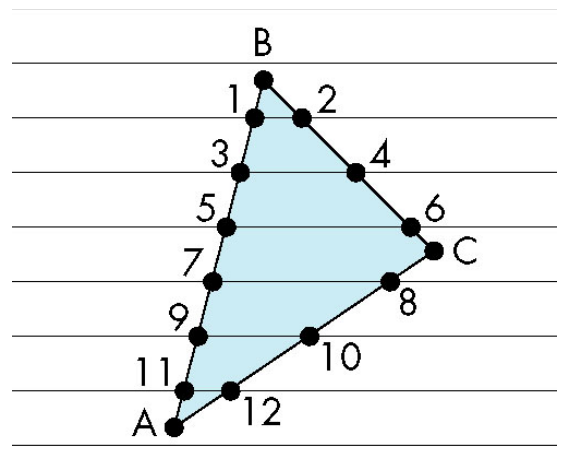
Scan Line Fill

Can also fill by maintaining a data structure of all intersections of polygons with scan lines

- Sort by scan line
- Fill each span

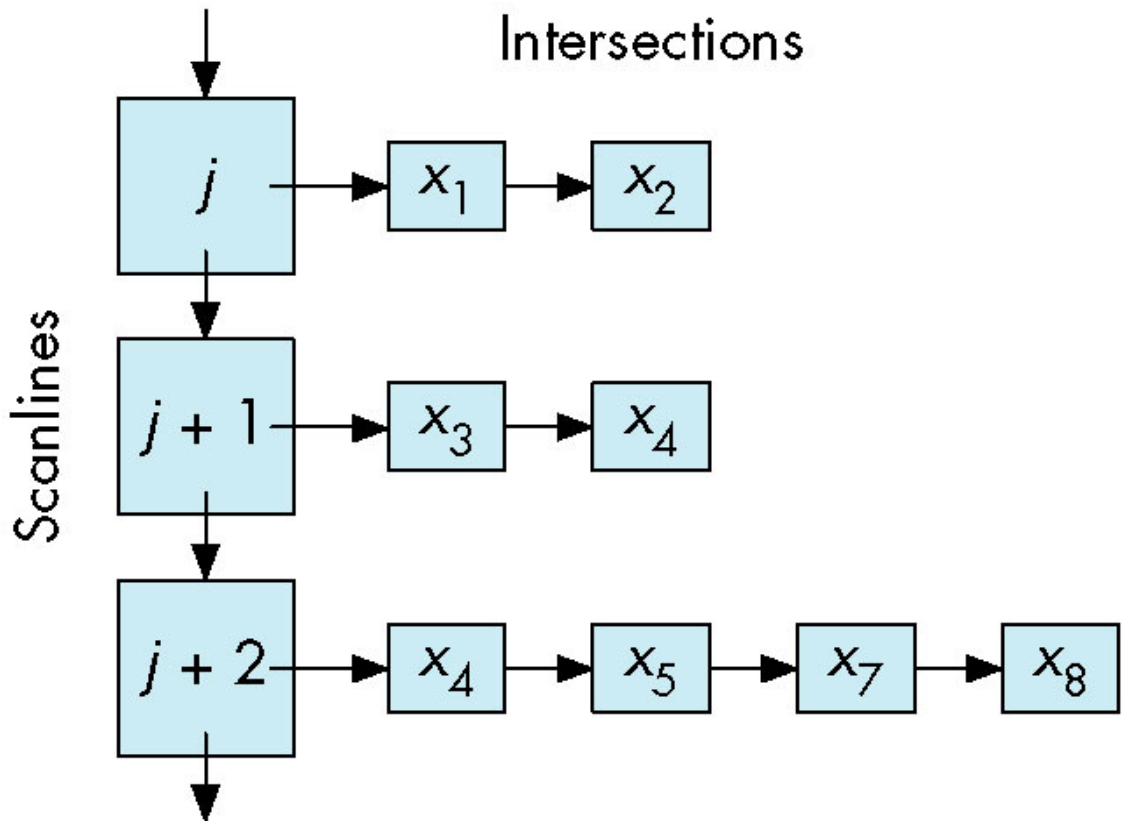


vertex order generated by vertex list



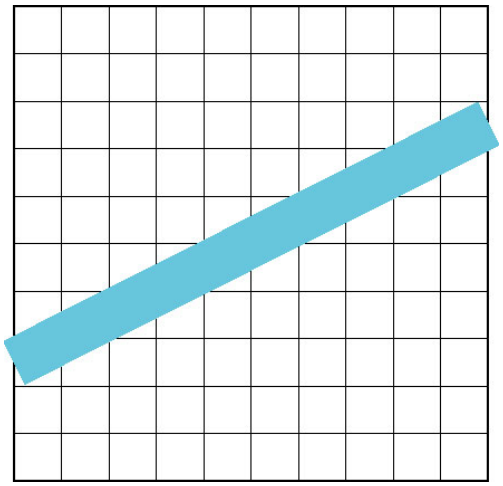
desired order

Data Structure



Aliasing

- Ideal rasterized line should be 1 pixel wide

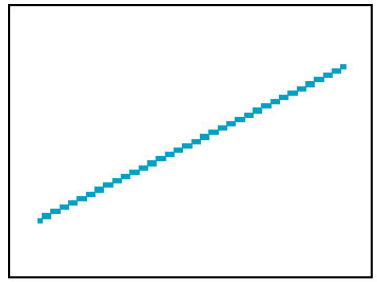


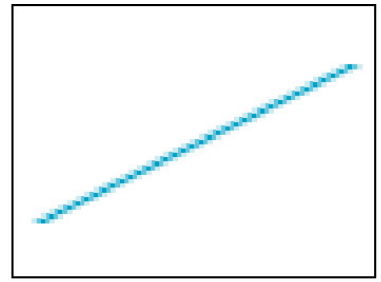
- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

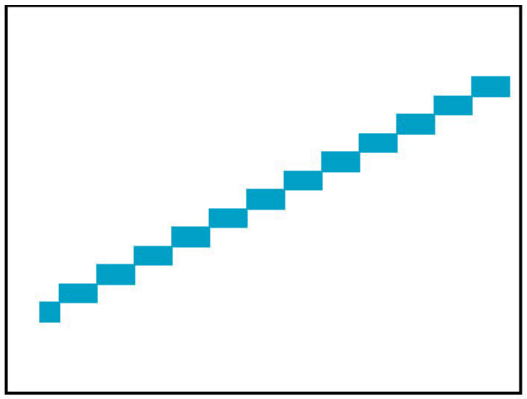
- Color multiple pixels for each x depending on coverage by ideal line

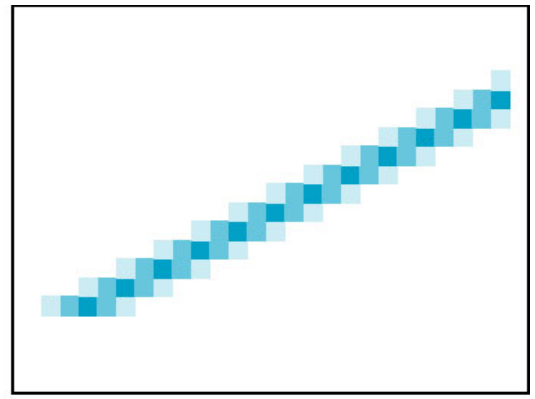
original





antialiased

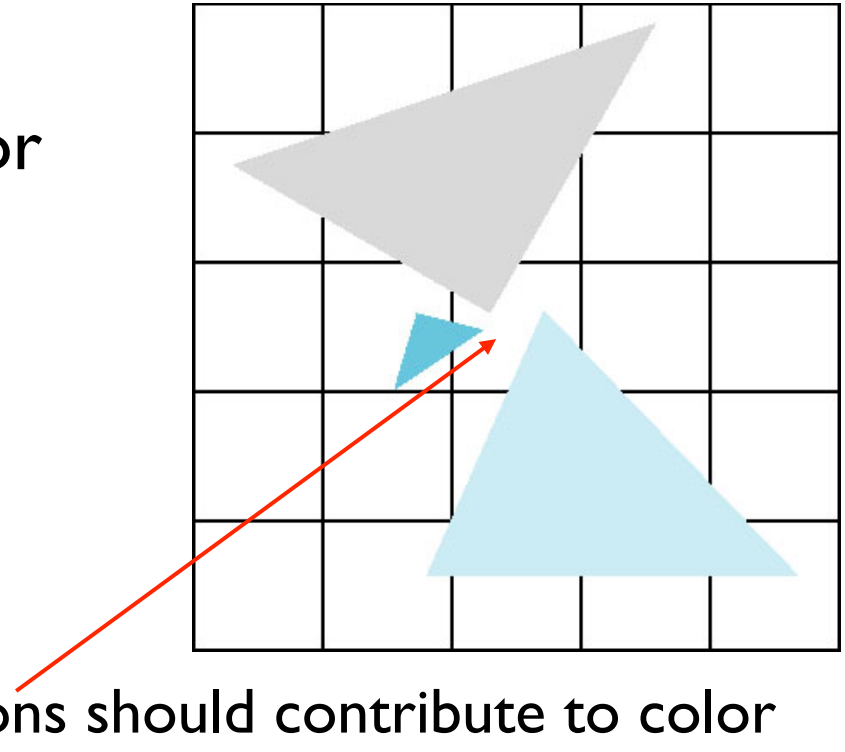




magnified

Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color