ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Surface Modeling

December 8th, 2015

Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Final Presentation

Thursday, Dec 10th, 2015, 2:10 - 3:30

- Show what you have to this points.
- Add sketches, concept drawings, etc. if cannot render anything at this point.

8 (min) -10 minutes presentation per team

- Introduce your team
- Introduce your application, especially the goal of the application. What is the user intend to do.
- Explain the objects in your scene and their behavior
- Explain the user interaction
- Highlight the most sophisticated techniques you used.
- Describe your experience.

- Test your presentations / computer in class next Tuesday.
- Online students: please record a video

Due date is still:

Due date: Friday, Dec. 18thsaf, 2015, 8 pm

COURSE ANNOUNCEMENT

ME 625
Surface Modeling

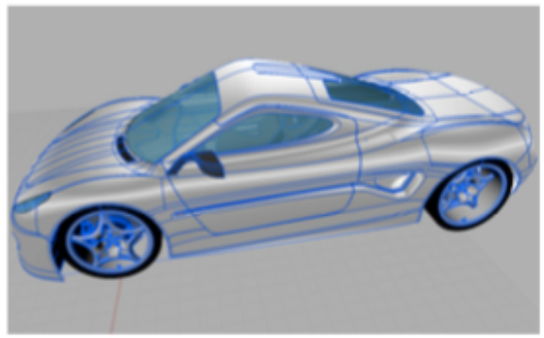
Spring 2016

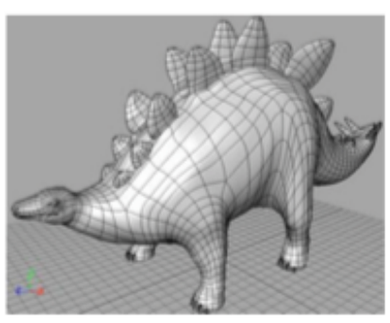
Instructor:
James Oliver

ARLAB

Surface modeling technologies are at the core of all contemporary computer shape modeling tools, spanning applications as diverse as mechanical, industrial, aerospace, naval, architectural, and even apparel design. This course explores the theory and practice of contemporary parametric sculptured surface design. Topics include:

- Curve and surface basics
- B-spline curves and surfaces
  - Basis functions
  - Derivatives
- Rational B-spline curves and surfaces
  - Conics
  - Re-parameterization
- Geometric tools
  - Knot insertion
  - Degree elevation
- Construction Techniques
- Trimmed surfaces
- Interpolation and Fitting
- Point Inversion and projection
- Shape modification
- Surface/surface intersection
- Applications – design, mesh generation, NC milling, integration with solid modeling systems, high-dimensional NURBS, others.





Text: *The NURBS Book*, by Les Piegl and Wayne Tiller, Springer-Verlag

Meeting Time: T, TH 11:00 – 12:15

Also available via Engineering/LAS Online Education

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

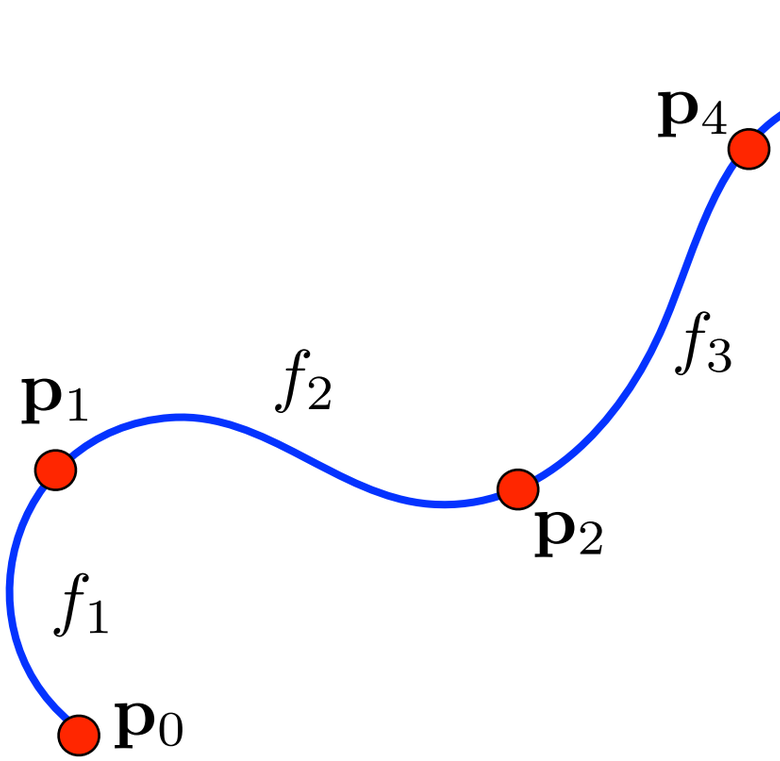
Content

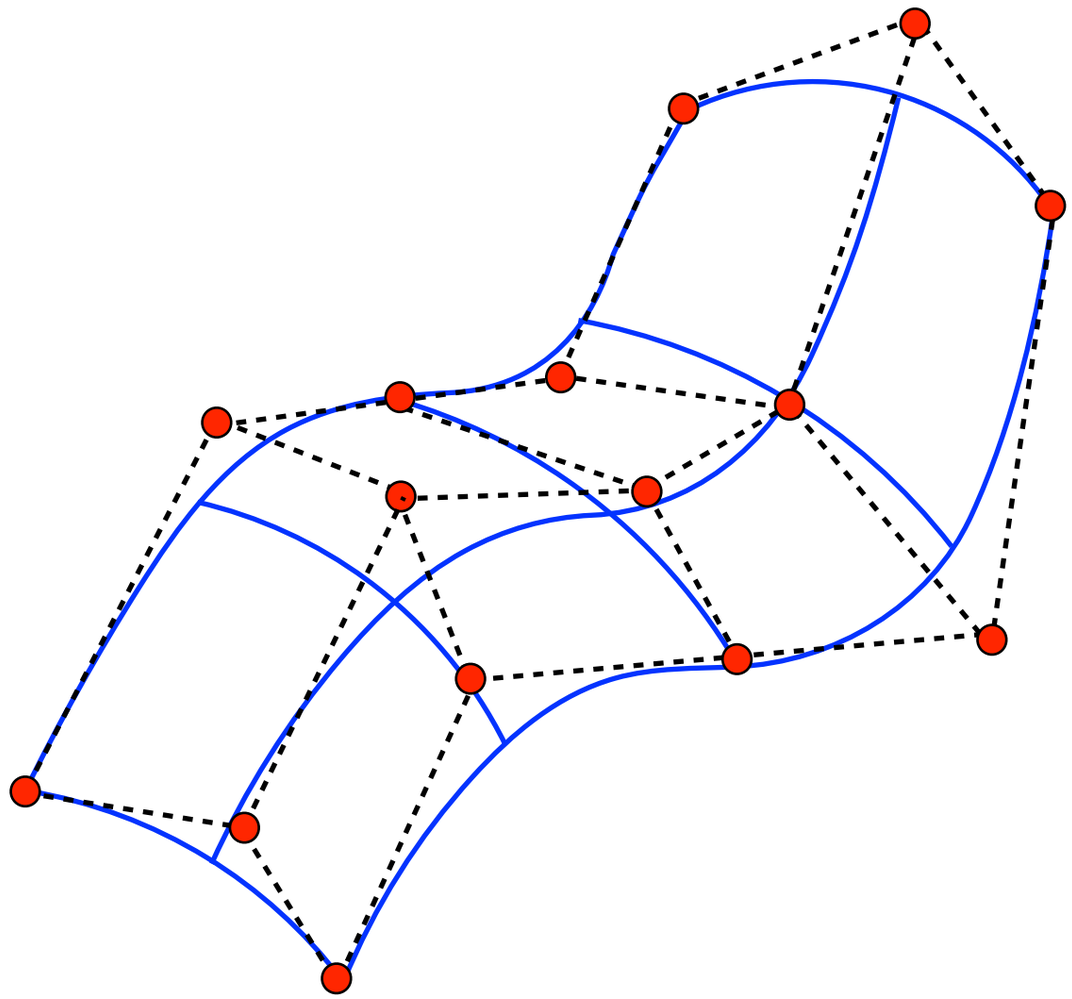
- Bezier Surface
- Surface Rendering

Spline

In modeling terminology, a spline curve is a flexible strip used to produce a smooth curve through a designated set of points. Mathematically, we can describe this curve with piecewise cubic polynomial functions.

A spline surface can be described with two sets of orthogonal spline curves.





Spline Specification

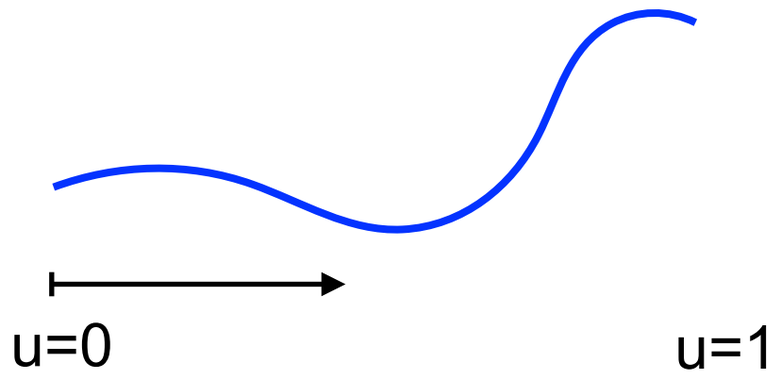
Let's start with a polynomial representation of a curve

$$x(u) = a_x\ u^3 + b_x\ u^2 + c_x\ u + d_x, \quad 0 \le u \le 1$$

Rewritten in matrix form.

$$x(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix}$$

$$= \mathbf{U} \cdot \mathbf{C}$$



We always run from u = [0,1]

Let's split C into

$$\mathbf{C} = \mathbf{M}_{spline} \cdot \mathbf{M}_{geom}$$

where $\mathbf{M}_{geom}$ is a four-element column matrix containing the geometric constraints (boundary conditions).

and $\mathbf{M}_{spline}$ is a 4x4 matrix that transforms the geometric constraints into the polynomial coefficients a_x, b_x, c_x, d_x.

Bezier Spline Equation

We start with the general equation for Bezier Splines of every order!

Given a set of control points for *n* curve sections, with *n*+1 control points

$$\mathbf{p}_k = \{x_k, y_k, z_k\}, \quad k = 0, 1, 2, ..., n$$

k, index of the point
n, number of points

We define a position vector to draw a line as

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k \mathrm{BEZ}_{k,n}(u), \quad 0 \le u \le 1$$

It describes the path between $\mathbf{p}_0$ and $\mathbf{p}_1$

The **Bezier blending function** is the *Bernstein polynomials*

$$\mathrm{BEZ}_{k,n}(u) = C(n,k)u^k(1-u)^{n-k}$$

with the parameter C:

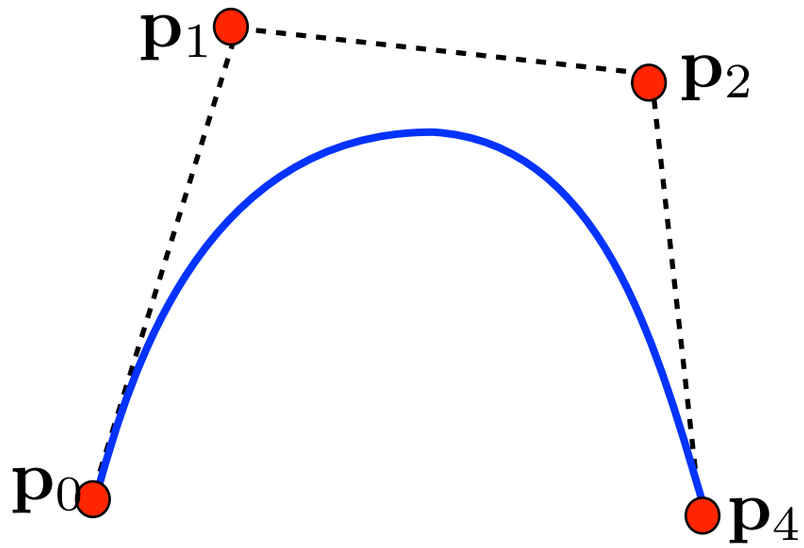
$$C(n,k) = \frac{n!}{k!(n-k)!}$$

0! = 1

Cubic Bezier Curve

For a Cubic Bezier Function, we use the following blending functions

$$\mathrm{BEZ}_{0,3}(u) = (1-u)^3$$
$$\mathrm{BEZ}_{1,3}(u) = 3u(1-u)^2$$
$$\mathrm{BEZ}_{2,3}(u) = 3u^2(1-u)$$
$$\mathrm{BEZ}_{3,3}(u) = u^3$$



The blending function determine how the control points influence the shape.

for u = 0: only the first blending function has an impact. Thus, we always begin with $\mathbf{p}_0$

for u = 1: only the last blending function has an impact, Thus, we will always end with $\mathbf{p}_3$

Each of this function is non zero in the range between 0 < u < 1

This also means, we have no local control points. The entire curve is always affected.

Cubic Bezier Curve

Back to the basic function:

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k \mathrm{BEZ}_{k,n}(u), \quad 0 \leq u \leq 1$$

We need to follow the boundary conditions:

$$\mathbf{P}(0) = \mathbf{p}_0$$

$$\mathbf{P}(1) = \mathbf{p}_n$$

Along with the hire order boundary conditions

$$\mathbf{P}'(0) = -n\,\mathbf{p}_0 + n\,\mathbf{p}_1$$

$$\mathbf{P}'(1) = -n\,\mathbf{p}_{n-1} + n\,\mathbf{p}_n$$

$$\mathbf{P}''(0) = n(n-1)[(\mathbf{p}_2 - \mathbf{p}_1) - (\mathbf{p}_1 - \mathbf{p}_0)]$$

$$\mathbf{P}''(1) = n(n-1)[(\mathbf{p}_{n-2} - \mathbf{p}_{n-1}) - (\mathbf{p}_{n-1} - \mathbf{p}_n)]$$

Cubic Bezier Curve

For our cubic curve at the endpoints

$$\begin{aligned}\mathbf{P}'(0) &= 3(\mathbf{p}_1 - \mathbf{p}_0)\\ \mathbf{P}'(1) &= 3(\mathbf{p}_3 + \mathbf{p}_2)\\ \mathbf{P}''(0) &= 6(\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2)\\ \mathbf{P}''(1) &= 6(\mathbf{p}_1 - 2\mathbf{p}_2 + \mathbf{p}_3)\end{aligned}$$

which we can transform into a matrix:

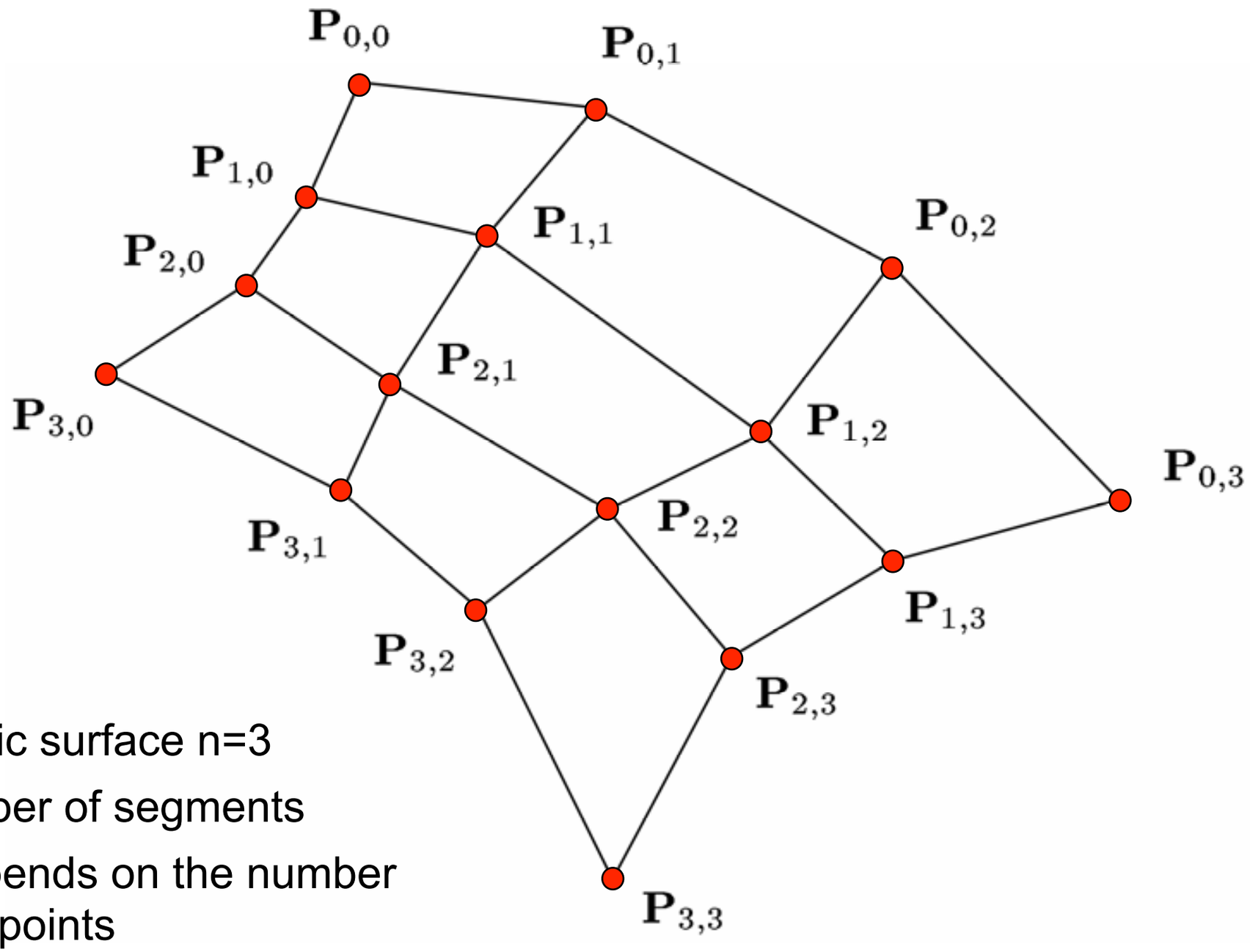
$$\mathbf{P}(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \cdot \mathbf{M}_{Bez} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix}$$

with the **Bezier matrix:**

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

Bezier-Surface

The surface is constructed from an n+1 × m+1 array of control points

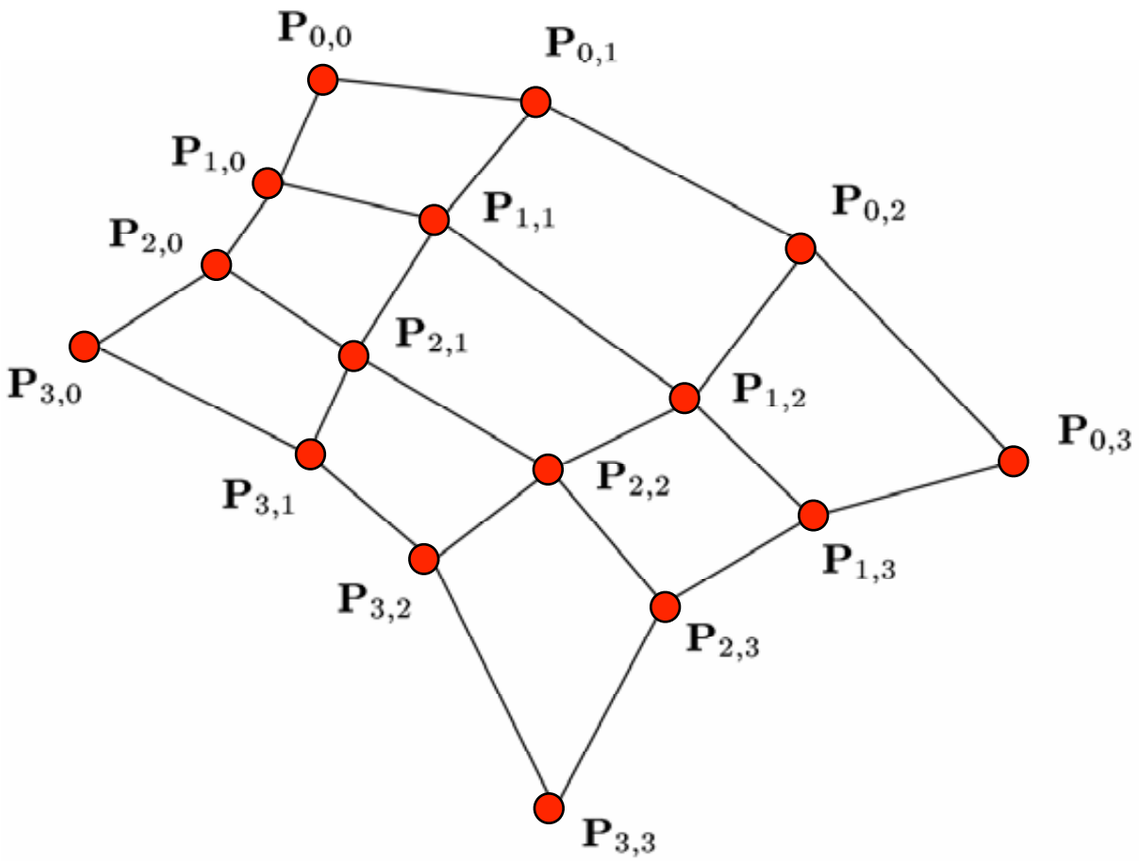


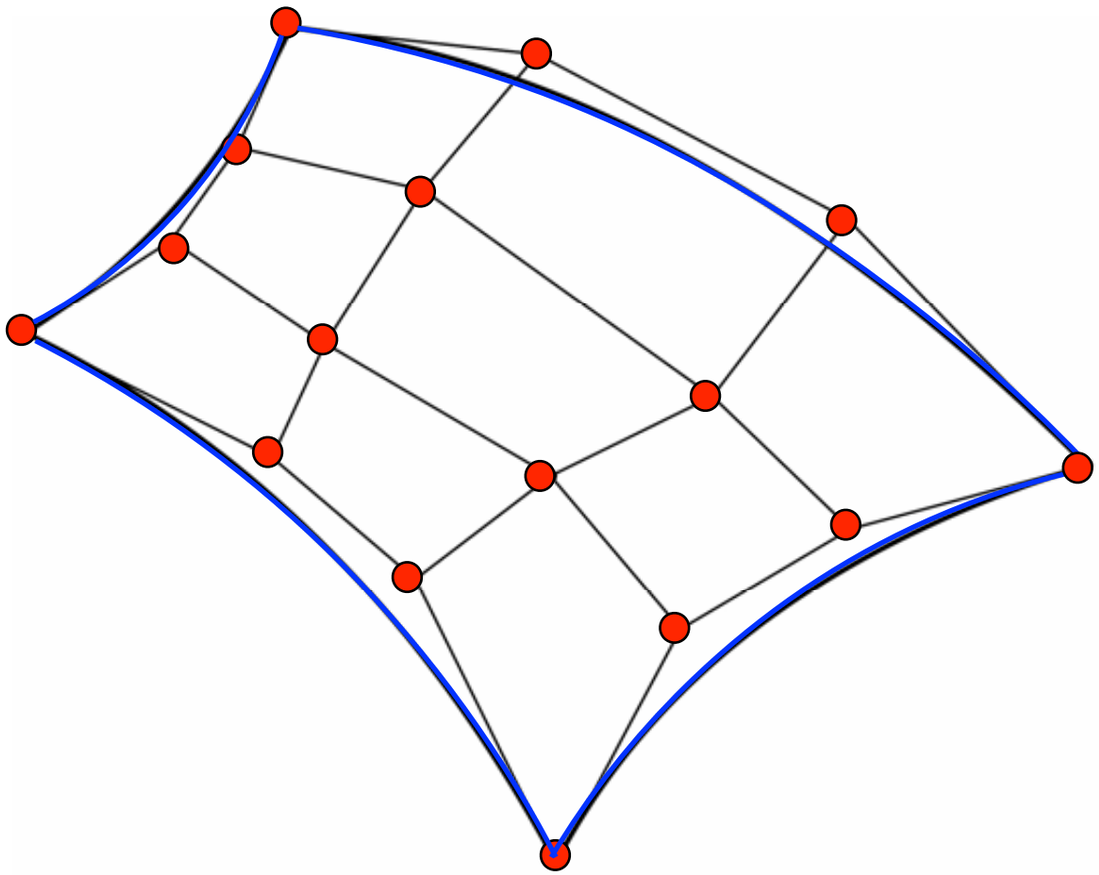
- Here: cubic surface n=3
- n,m: number of segments
- Order depends on the number of control points

Bezier-Surface

Consider n+1x m+1 control points, the Bézier surface is parameterized by two variables, is given by the equation

$$P(u,v) = \sum_{j=0}^{m} \sum_{k=o}^{n} \mathbf{p}_{j,k}(u)\ \mathrm{BEZ}_{j,m}(v)\ \mathrm{BEZ}_{k,m}(u)$$





VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Bezier-Surface

- It is summations running over all the control points
- The surface is formed as a product of the Bernstein functions:

$$\mathrm{BEZ}_{j,m}(v)\ \mathrm{BEZ}_{k,n}(u)$$

Multivariate Bernstein Polynomial

$$B_{f,n_1,n_2}(x_1, x_2) := \sum_{k_1=0}^{n_1} \sum_{k_2=0}^{n_2} f\left(\frac{k_1}{n_1}, \frac{k_2}{n_2}\right) \binom{n_1}{k_1} \binom{n_2}{k_2} x_1^{k_1} (1-x_1)^{n_1-k_1} x_2^{k_2} (1-x_2)^{n_2-k_2}$$

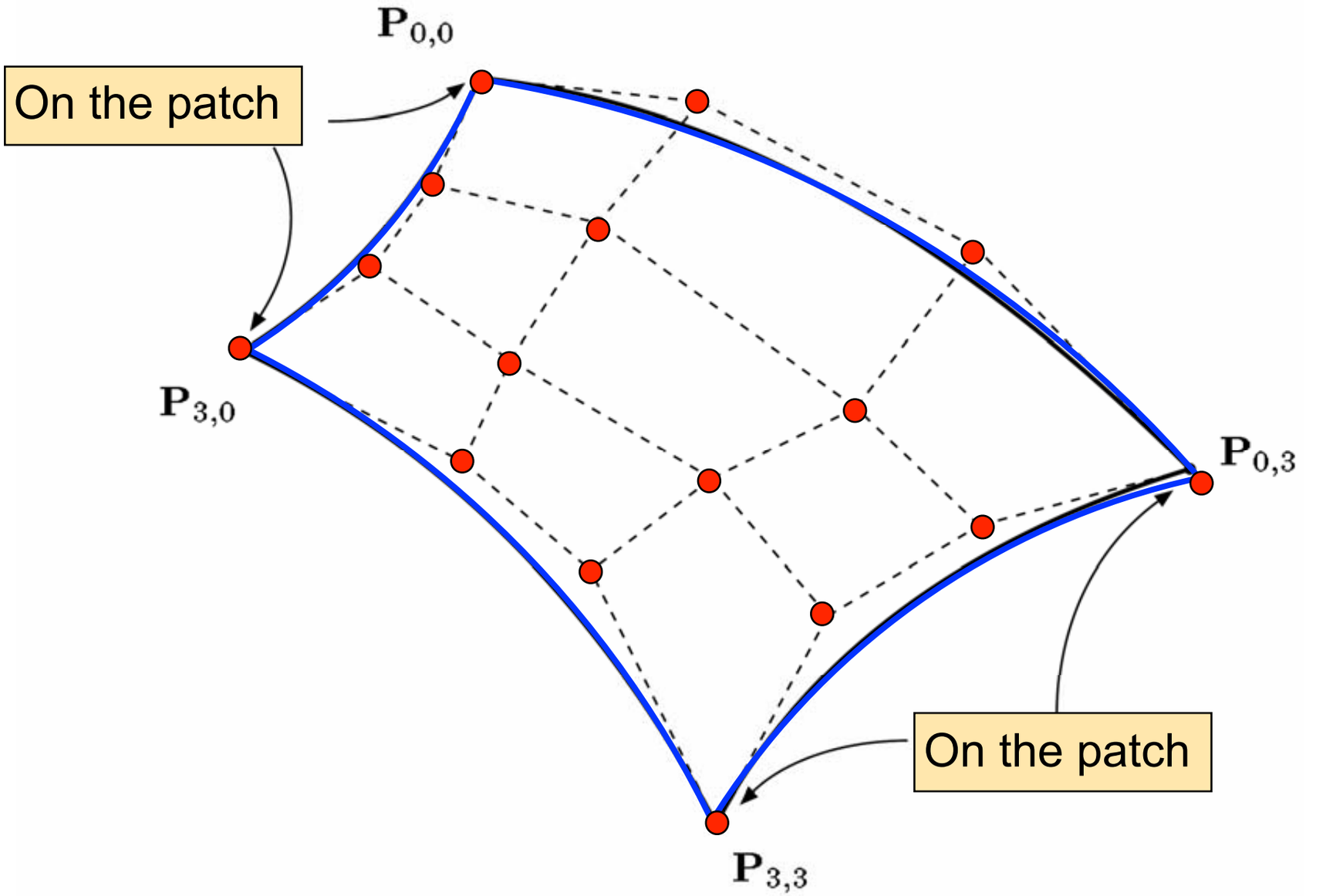
[wikipedia]

You can multiply the one-dimensional Bernstein functions

Properties

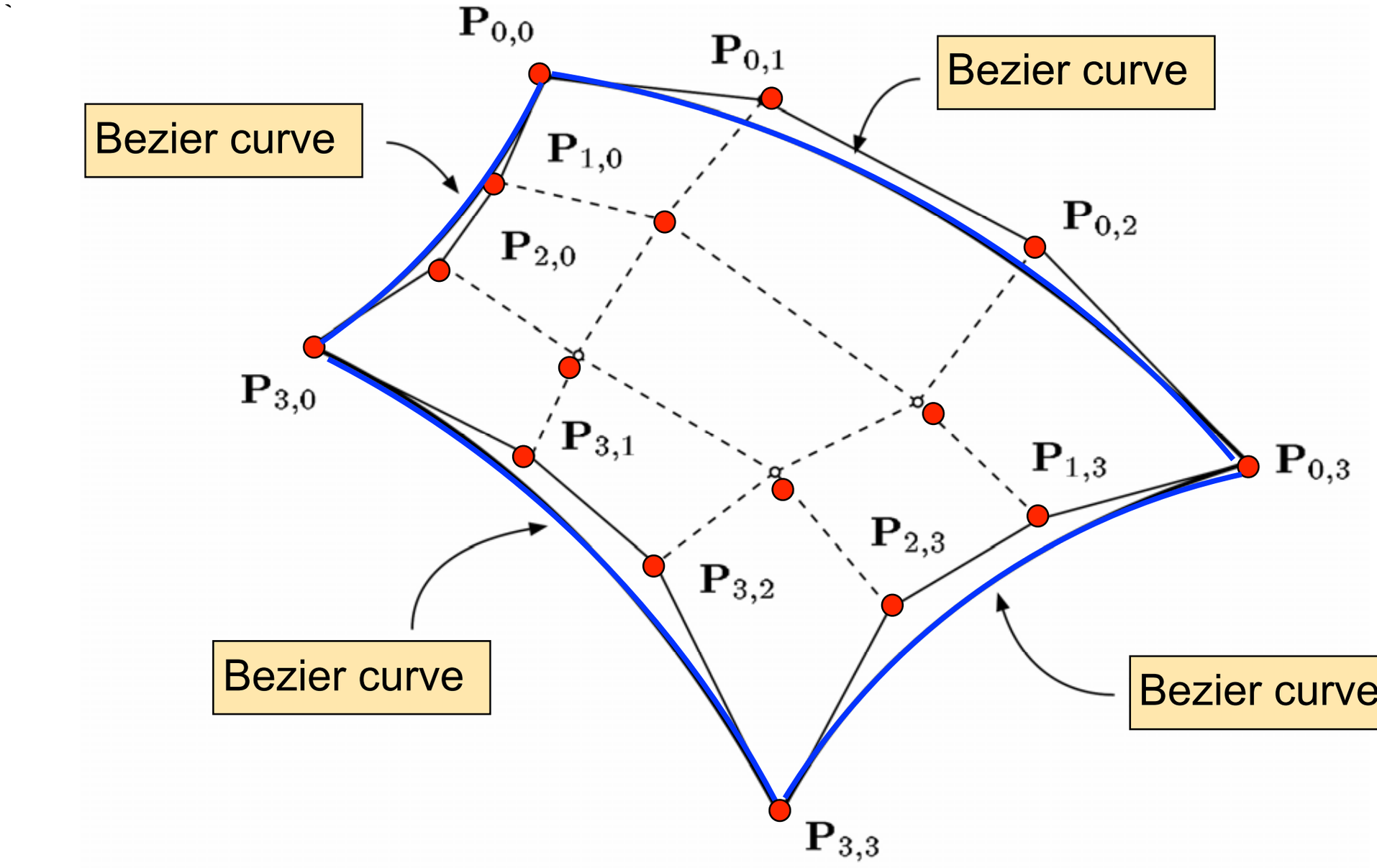
Do not expect that the entire surface intersect with the control points.

The corner of the control points align with the surface corners



Properties

The edges are Bezier curves. Do not expect that the entire surface intersect with the control points.



Properties

- The four points $P_{0,0}$, $P_{0,m}$, $P_{n,0}$ and $P_{n,m}$ are on the patch.
- The other control points are all on the patch only if the patch is planar.
- The patch is continuous and partial derivatives of all orders exist and are continuous.
- The patch lies within the convex hull of its control points.

ARLAB

Rendering

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Rendering

There are three main approaches to rendering an object defined by Bezier patches. In this summary, I will only consider the first two methods:

1. Calculate mesh of points on surface, and use mesh to generate vertex, polygon representation
2. Recursively subdivide patch into two smaller Bezier patches until patch projects to one pixel
3. Render directly from analytical description of curve.

Code Example

28_Bezier_Surface

- Two additional files:
- Bezier.h
- Bezier.cpp

Functionality:

- Bezier point computation from control points
- Triangulation
- Normal vector calculation

Generating Points from Control Points

```
/*!
Computes the points for a cubic spline
@param control_points - a vector of four points in x,y or x,y,z.
@param result - the output points
@param num - the number of points that should be generated along the spline
*/
bool ComputeCubicSpline(const int num, const vector< vector<float> >&
control_points, vector< vector<float> >& result);
```

```
/*!
Computes the points for a cubic spline
@param control_points - a vector of four points in x,y or x,y,z.
@param result - the output points
@param num - the number of points that should be generated along the spline
*/
bool ComputeCubicSplineC(const int num, const vector< vector<float> >&
control_points, vector< vector<float> >& result);

/*!
Computes a cubic surface patch.
@param points - a vector of four points in x,y or x,y,z.
@param result - the output points
@param num - the number of points that should be generated along the spline
*/
bool ComputeCubicPatch(const int num, const vector< vector<float> >& control_points,
vector< vector<float> >& result);
```

Generating Points from Control Points


```
/*!
Computes the points for a cubic spline
@param control_points - a vector of four points in x,y or x,y,z.
@param result - the output points
@param num - the number of points that should be generated along the spline
*/
bool ComputeCubicSpline(const int num, const vector< vector<float> >& control_points, vector< vector<float> >& result)
{
    result.clear();

    float increment = 1.0/float(num-1);


    vector<float> cp0 = control_points[0];
    vector<float> cp1 = control_points[1];
    vector<float> cp2 = control_points[2];
    vector<float> cp3 = control_points[3];
```

Increment

The control points

```
    for (int i=0; i<num; i++) {

        double t = increment * i;

        vector<float>p(3);
        p[0] = cp0[0] * ComputeBernsteinP(t,3,0) + cp1[0] * ComputeBernsteinP(t,3,1) + cp2[0] * ComputeBernsteinP(t,3,2) + cp3[0]
* ComputeBernsteinP(t,3,3);
        p[1] = cp0[1] * ComputeBernsteinP(t,3,0) + cp1[1] * ComputeBernsteinP(t,3,1) + cp2[1] * ComputeBernsteinP(t,3,2) + cp3[1]
* ComputeBernsteinP(t,3,3);
        p[2] = cp0[2] * ComputeBernsteinP(t,3,0) + cp1[2] * ComputeBernsteinP(t,3,1) + cp2[2] * ComputeBernsteinP(t,3,2) + cp3[2]
* ComputeBernsteinP(t,3,3);

        cout << t << " :\t" << p[0] << "\t" << p[1] << "\t" << p[2] << endl;

        result.push_back(p);
    }

    return true;
```

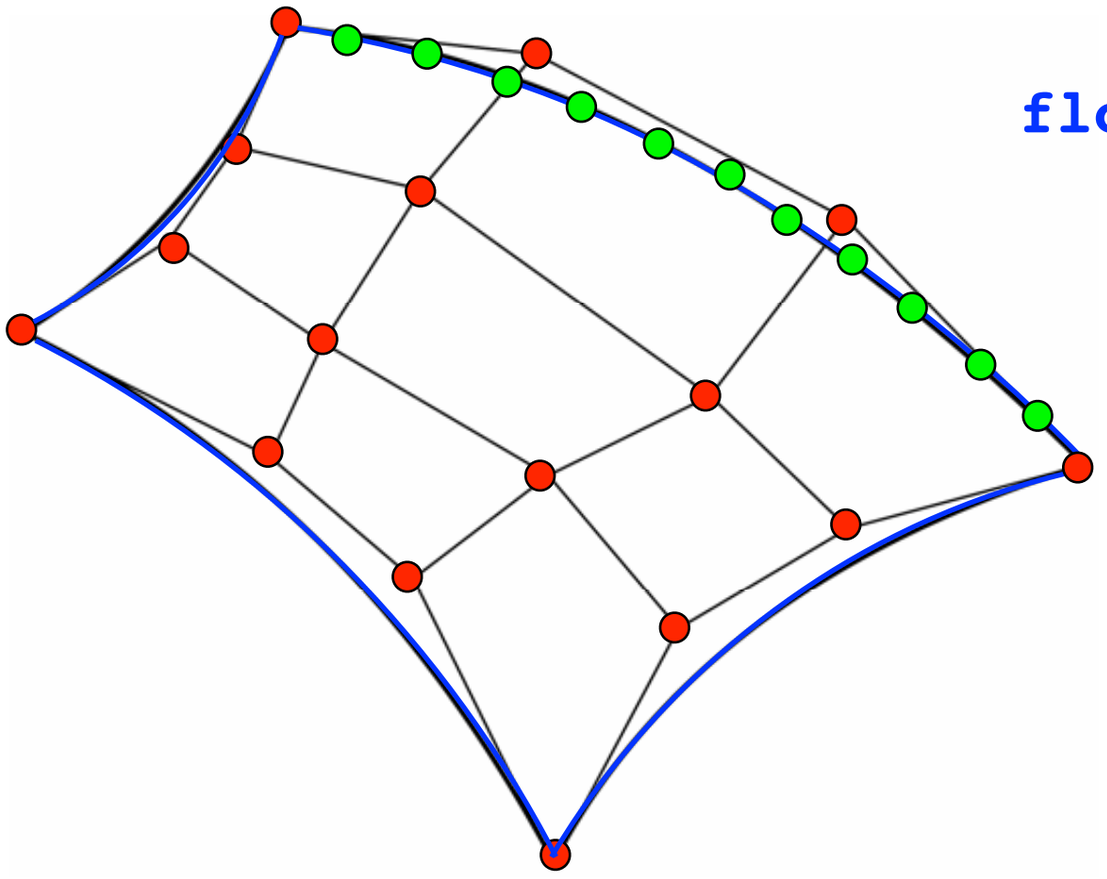
The cubic spline

Try not to read the code

Generating Points from Control Points

To render a line, we need points between the control points, along the Bezier curve.

Therefore, we discretize the parameter *u* and *v* = [0,1]

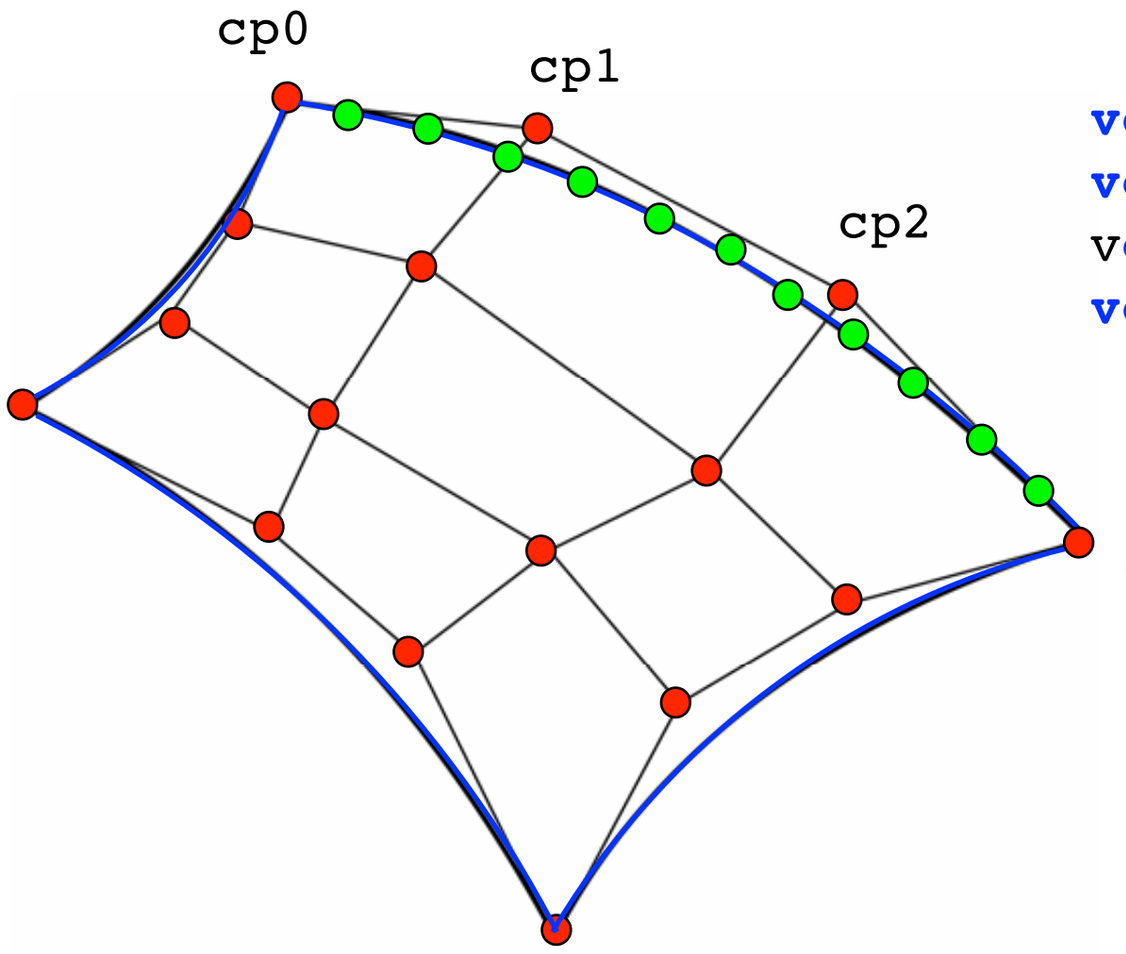


```
float increment = 1.0/float(num-1);
```

A Bezier surface. Note, we discuss Bezier line code.
A Bezier line is part of the surface.

Generating Points from Control Points

Each control point is stored as a vector with three components, x, y, and z




```
vector<float> cp0 = control_points[0];
vector<float> cp1 = control_points[1];
vector<float> cp2 = control_points[2];
vector<float> cp3 = control_points[3];
```

x	y	z
0.2	0.34	0.3

A Bezier surface. Note, we discuss Bezier line code.
A Bezier line is part of the surface.

Generating Points from Control Points

```
for (int i=0; i<num; i++) {

        double t = increment * i;

        vector<float>p(3);
x:      p[0] = cp0[0] * BEZ(t,3,0) + cp1[0] * BEZ(t,3,1) + cp2[0] *
              BEZ(t,3,2) + cp3[0] * BEZ(t,3,3);
y:      p[1] = cp0[1] * BEZ(t,3,0) + cp1[1] * BEZ(t,3,1) + cp2[1] *
              BEZ(t,3,2) + cp3[1] * BEZ(t,3,3);
z:      p[2] = cp0[2] * BEZ(t,3,0) + cp1[2] * BEZ(t,3,1) + cp2[2] *
              BEZ(t,3,2) + cp3[2] * BEZ(t,3,3);

        result.push_back(p);
}
```

which is this function: for n = 3, k = {0, 1, 2, 3}

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k \mathrm{BEZ}_{k,n}(u), \quad 0 \le u \le 1$$

ComputeBernsteinP = BEZ

Bernstein Function

ARLAB

```
/*!
Calculate the bernstein polynom for P(x) = BC(n,k)  x^k (1-x)^{n-k}
@param x - the interpolation variable x = [0,1]
@param n - the number of line segments where the number of supporting
points is n+1
@param k - the current point
*/
float ComputeBernsteinP(float x, int n, int k)
{
    int c = BinomialCoeff(n,k);

    float p_x = float(c) * pow(x, k) *  pow((1.0-x), n-k);

    return p_x;
}
```

$$\mathrm{BEZ}_{k,n}(u) = C(n,k)u^k(1-u)^{n-k}$$

VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

Binomial Coefficient

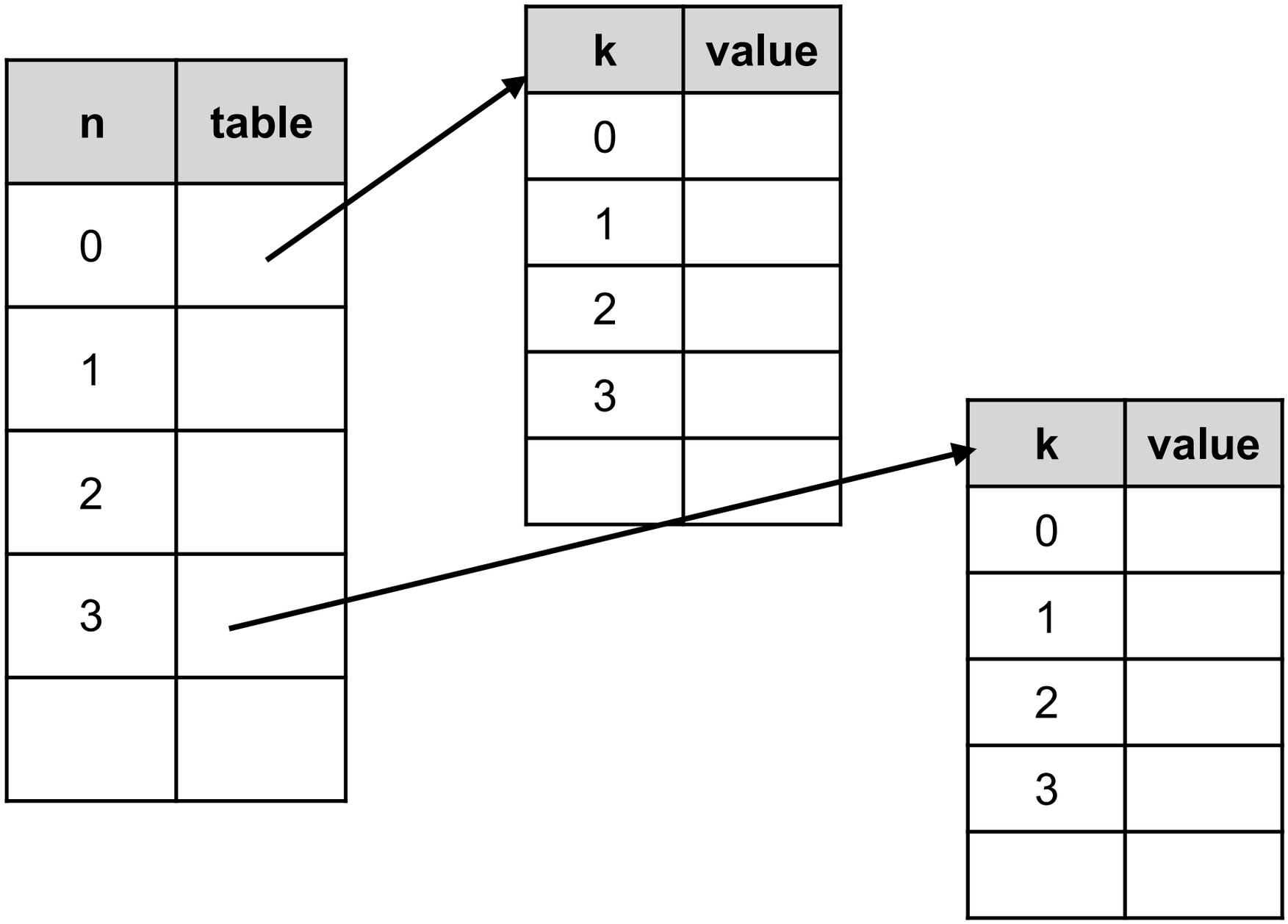
ARLAB

```
// [  n , [k, coeff] ]
map<unsigned int, map< unsigned int,unsigned int> > binominal_coefficients;


/*!
Calculate the binomial coefficient for n and k
@param n - the number of line segments where the number of supporting points
is n+1
@param k - the current point
*/
int BinomialCoeff(int n, int k)
{
    int c = binominal_coefficients[n][k];

    if(c==0) // binominal coefficient is min = 1, 0! = 1
    {
        c = factorial(n)/ ( factorial(k) * factorial(n-k) );
        binominal_coefficients[n][k] = c;
        return c;
    }
    return c;
    //return factorial(n)/ ( factorial(k) * factorial(n-k) );
}
```

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

Binomial Coefficient

```
// [  n , [k, coeff] ]
map<unsigned int, map< unsigned int,unsigned int> > binominal_coefficients;
```



Factorial x!

ARLAB

```
/*!
Calculates the factorial of a number
@param x - the value that should be factorized.
@return - the factor.
*/
int factorial(int x, int result = 0) {
  if (x == 1  || x == 0) return result;
  else return factorial(x - 1, x * result);
}
```

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

main

```
vector< vector<float> > control_points;
vector< vector<float> > result;

vector<float> p0 = vector<float>(3,0);
p0[0] = 0.0; p0[1] = 1.0; p0[2] = 0.0;

vector<float> p1 = vector<float>(3,0);
p1[0] = 0.0; p1[1] = 2.0; p1[2] = 1.0;

vector<float> p2 = vector<float>(3,0);
p2[0] = 0.0; p2[1] = 3.0; p2[2] = 3.0;

vector<float> p3 = vector<float>(3,0);
p3[0] = 0.0; p3[1] = 2.0; p3[2] = 4.0;

control_points.push_back(p0);
control_points.push_back(p1);
control_points.push_back(p2);
control_points.push_back(p3);

ComputeCubicSplineC(10, control_points, result);
```

Result

u	x	y	z
0:00	0	1	0
0.111111	0	1.33059	0.367627
0.222222	0	1.64472	0.792867
0.333333	0	1.92593	1.25926
0.444444	0	2.15775	1.75034
0.555556	0	2.32373	2.24966
0.666667	0	2.40741	2.74074
0.777778	0	2.39232	3.20713
0.888889	0	2.262	3.63237
1:00	0	2	4

ComputeCubicSplineC

The spline function:

$$\mathbf{P}(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \cdot \mathbf{M}_{Bez} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix}$$

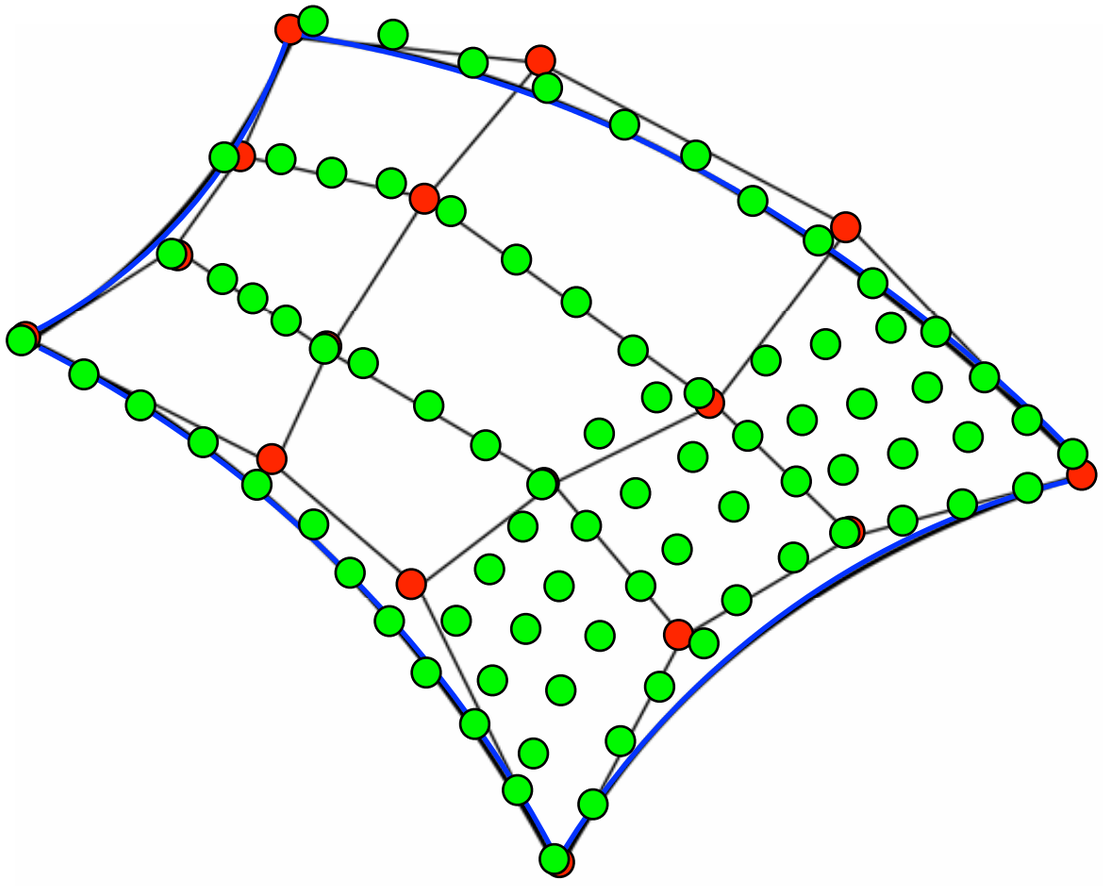
with the **Bezier matrix:**

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

```
// the bezier matrix
   static vector< vector<float> > bm( 4, vector<float>(4, 0.0));
   bm[0][0] = -1; bm[0][1] = 3;  bm[0][2] = -3; bm[0][3] = 1;
   bm[1][0] = 3;  bm[1][1] = -6; bm[1][2] = 3;  bm[1][3] = 0;
   bm[2][0] = -3; bm[2][1] = 3;  bm[2][2] = 0;  bm[2][3] = 0;
   bm[3][0] = 1; bm[3][1] = 0;  bm[3][2] = 0;  bm[3][3] = 0;
```

Cubic Surface in 3D

$$P(u,v) = \sum_{j=0}^{m} \sum_{k=o}^{n} \mathbf{p}_{j,k}(u)\ \mathrm{BEZ}_{j,m}(v)\ \mathrm{BEZ}_{k,m}(u)$$



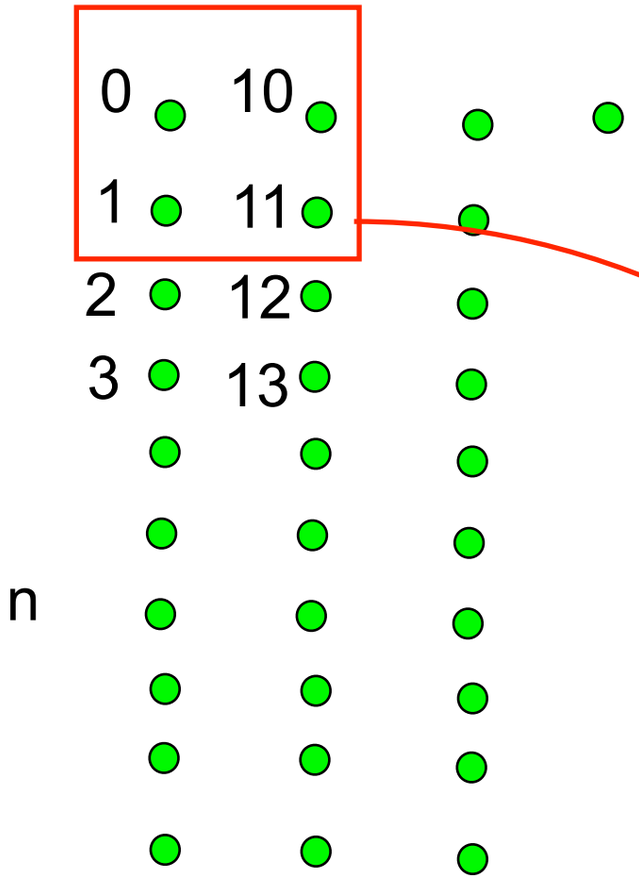
Bezier.cpp / this code does not fit on any slide

Rendering

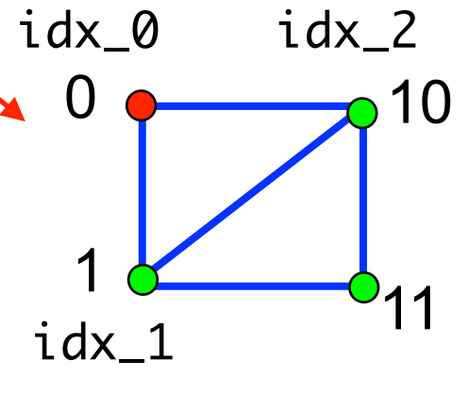
Subdivide into triangles:

```
/*
@param vertices, the vertices in an vector. Three vertices in
a row are one triangle
@param normals, a normal vector for each point
*/
bool TriangulateCubicPatch(const int n, const int m, const
vector< vector<float> >& points, vector< vector<float> >&
vertices, vector< vector<float> >& normals )
```

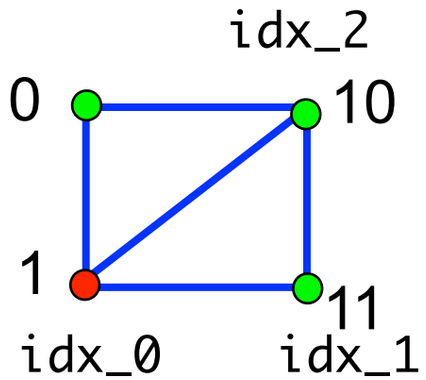
Find the points



i = 0



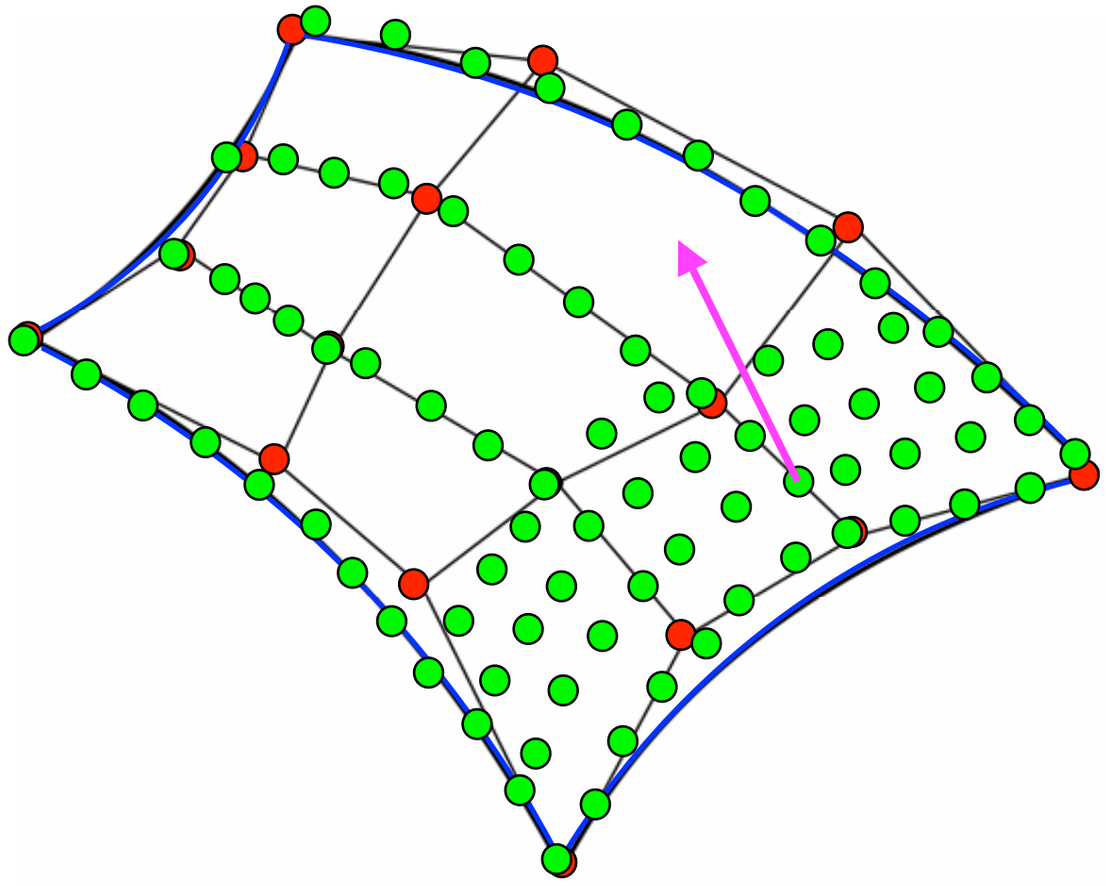
$$idx_0 = i$$
$$idx_1 = i + 1$$
$$idx_2 = 1 + n$$



$$idx_0 = i + 1$$
$$idx_1 = i + 1 + n$$
$$idx_2 = i + n$$

Normal vectors

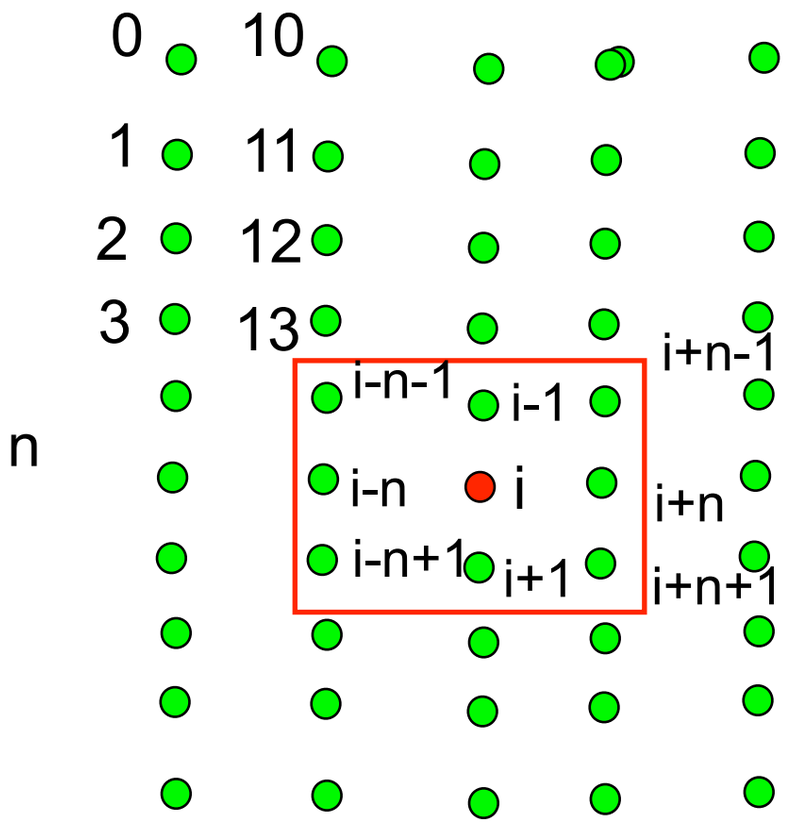
For rendering, we need a normal vector for each point



For each point:

- find the neighbors
- compute the covariance matrix
- calculate the eigenvectors and eigenvalues
- the eigenvector, which is associated with the smallest eigenvalue is the normal vector.
- But we calculate it from the two other vectors

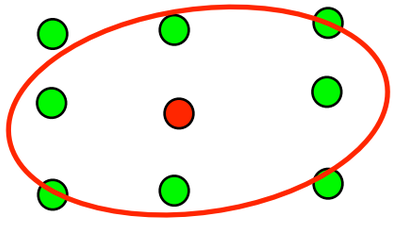
Find the neighbors



Take advantage of the regular grid that we have:

```
const int box[8] = { -n-1, -1 ,n-1,
                     -n ,      n,
                     -n+1, +1, n+1};
```

Calculate the covariance matrix



in general:

$$Cov(X,Y) = \sum (X_i - \bar{X}_i)(Y_i - \bar{Y}_i)/N$$

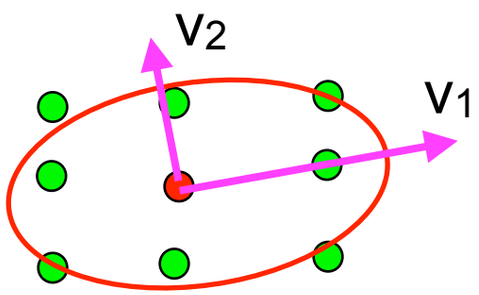
for points in 3D

$$Cov(\mathbf{p}) =$$

$$\frac{1}{N}\begin{bmatrix} \sum_i (x_i - \mu_x)^2 & \sum_i (x_i - \mu_x)(y_i - \mu_y) & \sum_i (x_i - \mu_x)(z_i - \mu_z) \\ \sum_i (y_i - \mu_y)(x - \mu_x) & \sum_i (y_i - \mu_y)^2 & \sum_i (y_i - \mu_y)(z_i - \mu_z) \\ \sum_i (z_i - \mu_z)(x - \mu_x) & \sum_i (z_i - \mu_z)(y_i - \mu_y) & \sum_i (z_i - \mu_z)^2 \end{bmatrix}$$

You will get the directions in which the points co-vary

Eigenvalues and eigenvectors



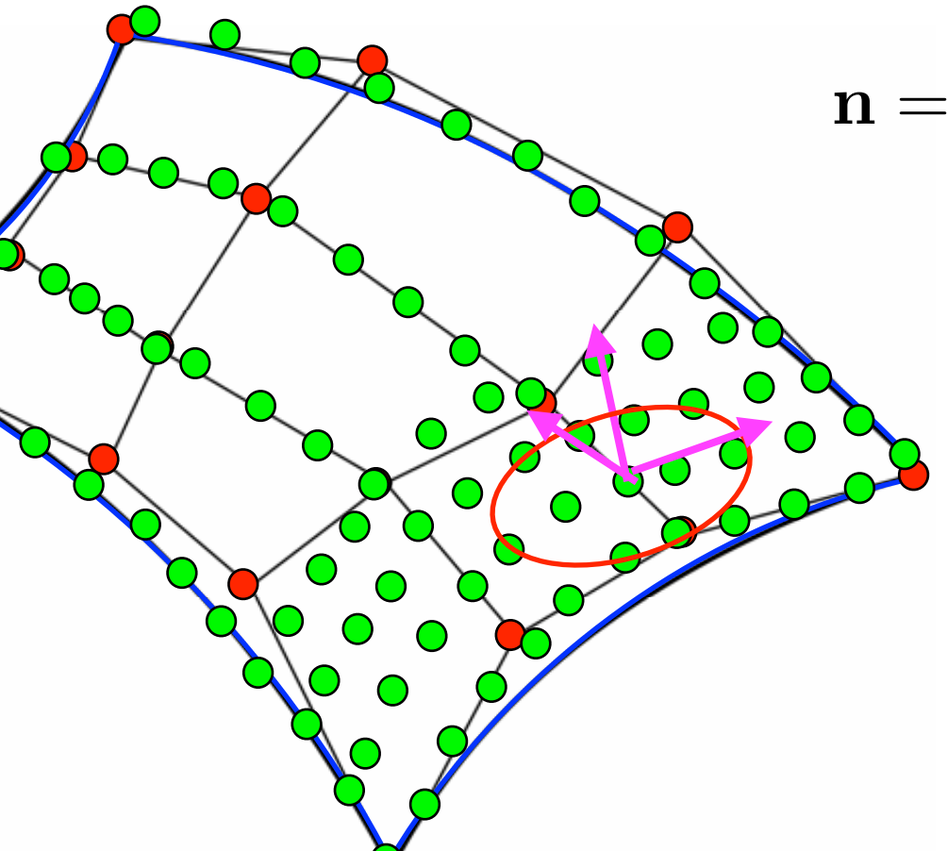
Solve the characteristic equation to get the eigenvalues and eigenvectors.

$$Cov(\mathbf{p})\mathbf{v} = \lambda\mathbf{v}$$

Select the vectors which are associated with the largest and second largest eigenvalue

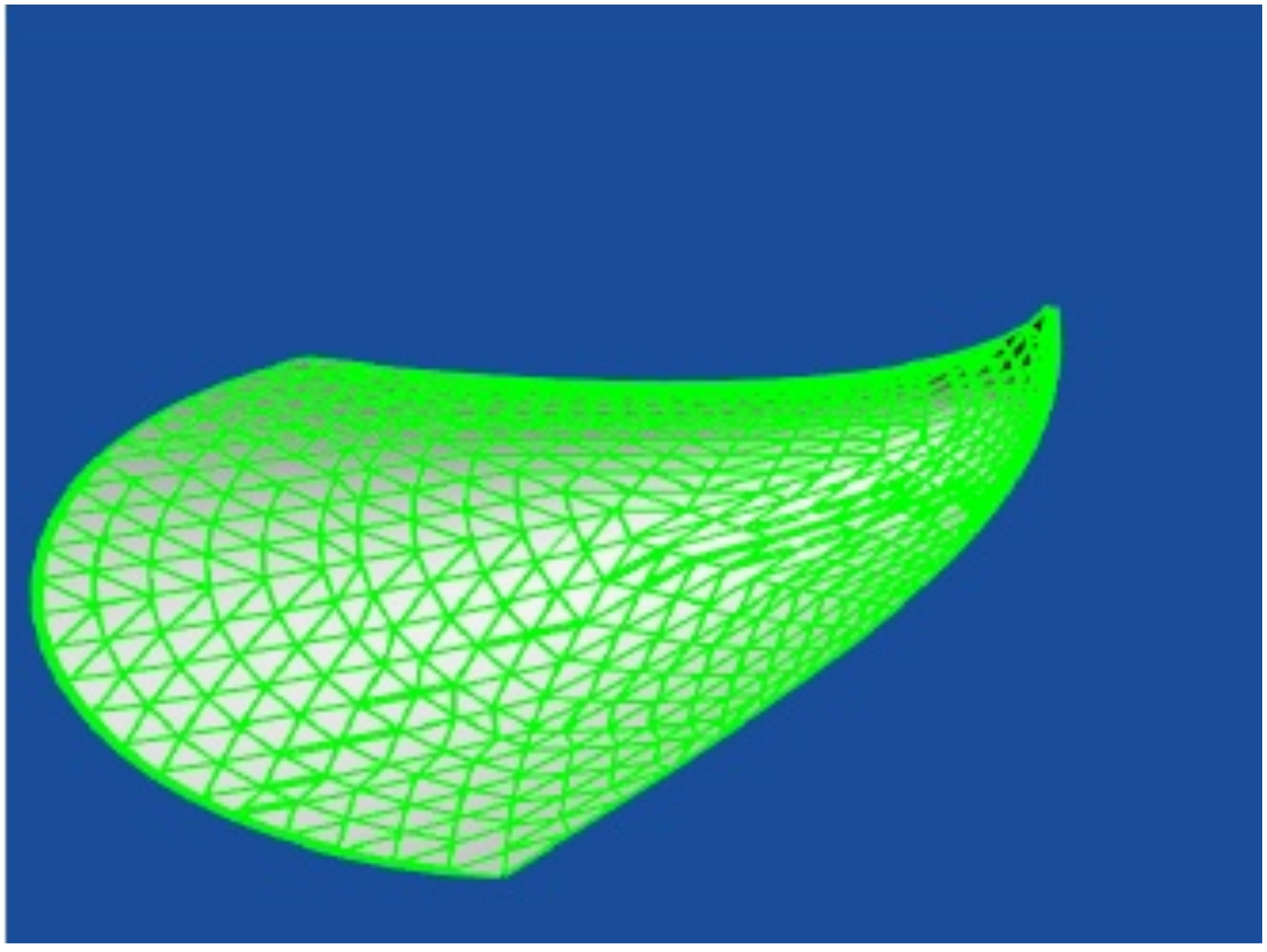
Calculate the normal vector as crossproduct of those:

$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$$



Result

ARLAB



VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 50011, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu
http://arlab.me.iastate.edu

www.linkedin.com/in/rradkowski

ARLAB

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY