Geometric Modeling

Summer Semester 2012

Spline Surfaces

Tensor Product Surfaces · Total Degree Surfaces

UNIVERSITÄT DES SAARLANDES

M²CI CLUSTER OF EXCELLENCE

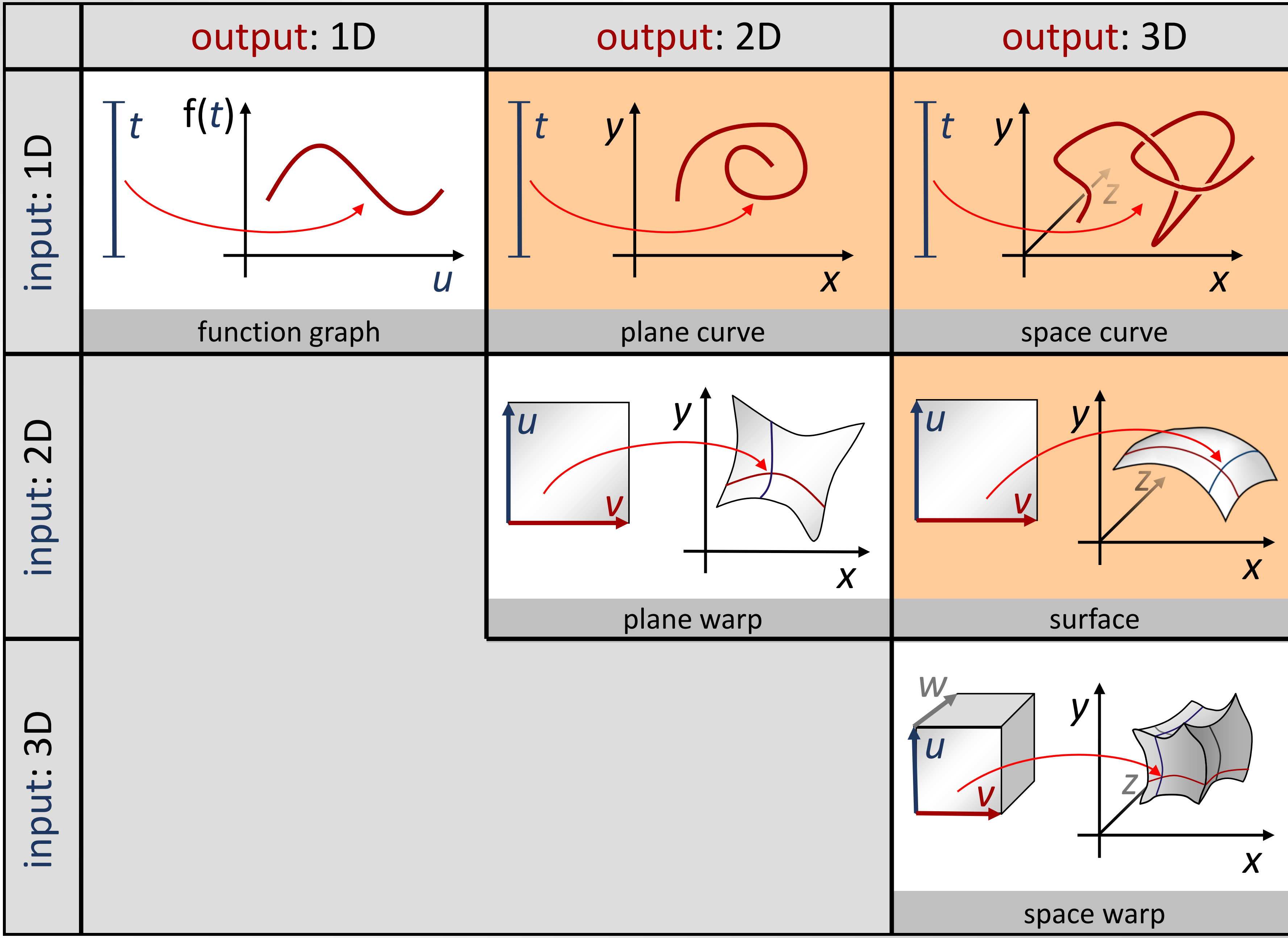
max planck institut informatik

Overview...

Topics:

- Polynomial Spline Curves
- Blossoming and Polars
- Rational Spline Curves
- Spline Surfaces
  - Introduction
  - Tensor Product Surfaces
  - Total Degree Surfaces

Introduction:
Spline Surfaces

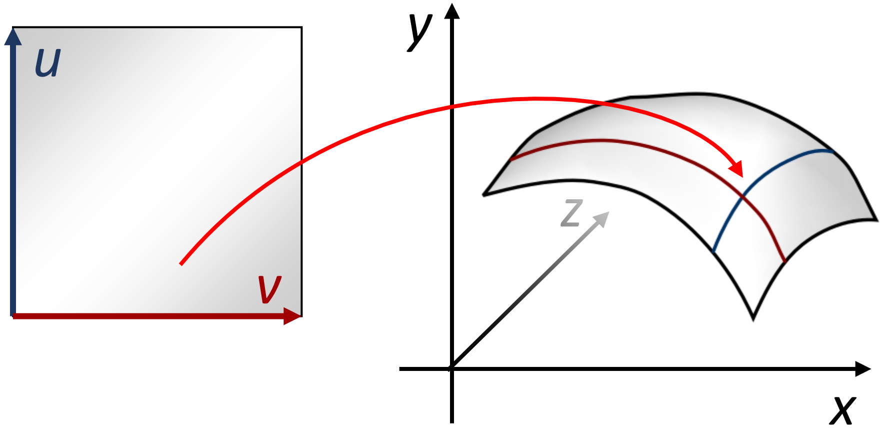


output: 1D
output: 2D
output: 3D
input: 1D
input: 2D
input: 3D
t
f(t)
u
function graph
t
y
x
plane curve
t
y
z
x
space curve
u
v
y
x
plane warp
u
v
y
z
x
surface
w
u
v
y
z
x
space warp

Spline Surfaces

Parametric spline surfaces:

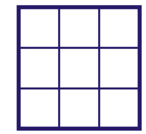
- Two parameter coordinates (u,v)
- Piecewise bivariate polynomials (rational surfaces → homogeneous coords)
- Assemble multiple pieces to form a surface with continuity
- Each piece is called *spline patch*

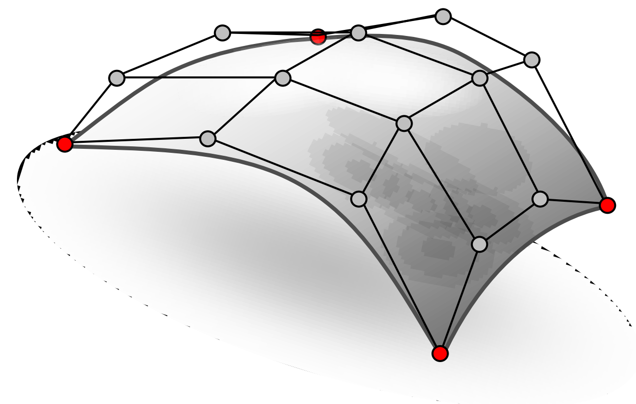


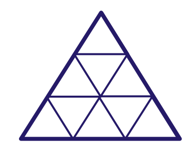
Spline Surfaces

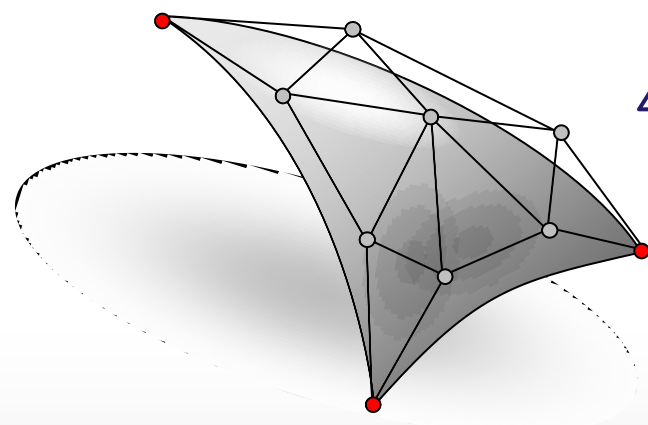
Two different approaches

- Tensor product surfaces
  - Simple construction
  - Everything carries over from curve case
  - Quad patches
  - Degree anisotropy
- Total degree surfaces
  - Not as straightforward (blossoming will help)
  - Isotropic degree
  - Triangle patches









Tensor Product Surfaces

Tensor Product Surfaces

Simple Idea:

- Given a basis for a one dimensional function space on the interval $t \in [t_0, t_1] \to \mathbb{R}^d$:

  $\mathbf{B}^{(\text{curv})} := \{b_1(t), \ldots, b_n(t)\}$

- Build a new basis with two parameters by taking all possible products:

  $\mathbf{B}^{(\text{surf})} := \{b_1(u)b_1(v), b_1(u)b_2(v), \ldots, b_n(u)b_n(v)\}$

Tensor Product Surfaces

Tensor product basis

	$b_1(u)$	$b_2(u)$	$b_3(u)$	$b_4(u)$
$b_1(v)$	$b_1(v)b_1(u)$	$b_1(v)b_2(u)$	$b_1(v)b_3(u)$	$b_1(v)b_4(u)$
$b_2(v)$	$b_2(v)b_1(u)$	$b_2(v)b_2(u)$	$b_2(v)b_3(u)$	$b_2(v)b_4(u)$
$b_3(v)$	$b_3(v)b_1(u)$	$b_3(v)b_2(u)$	$b_3(v)b_3(u)$	$b_3(v)b_4(u)$
$b_4(v)$	$b_4(v)b_1(u)$	$b_4(v)b_2(u)$	$b_4(v)b_3(u)$	$b_4(v)b_4(u)$

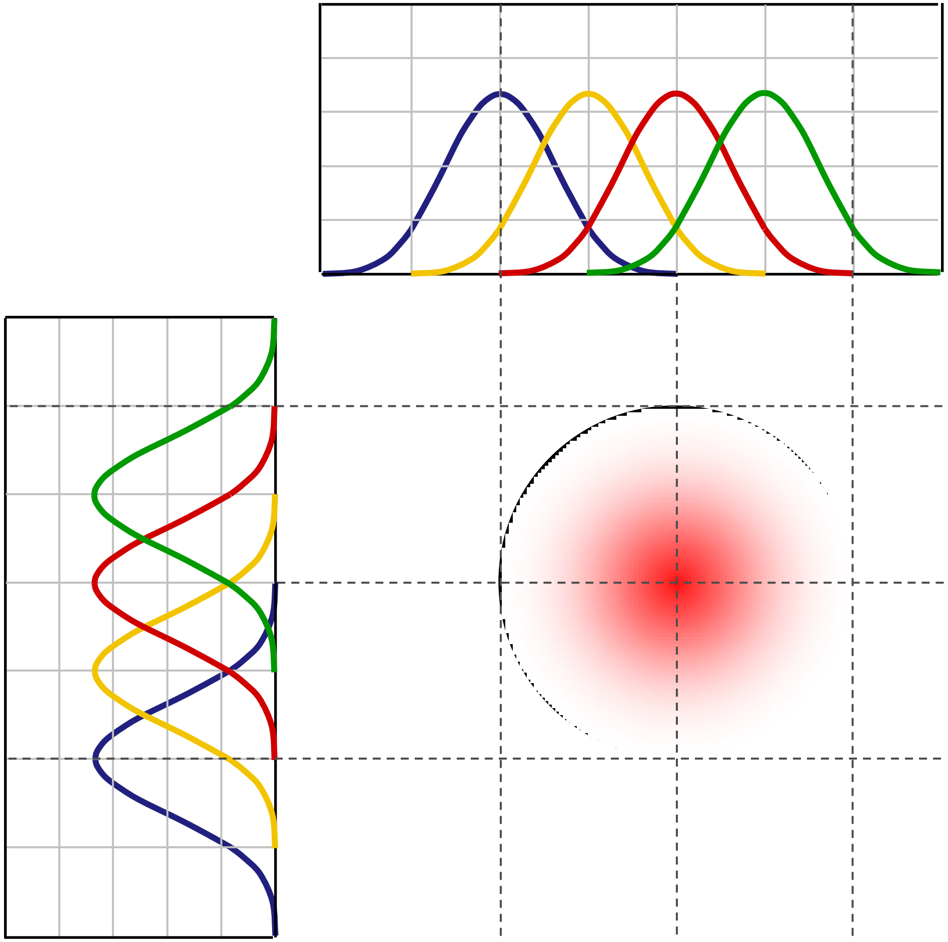
Monomial Example

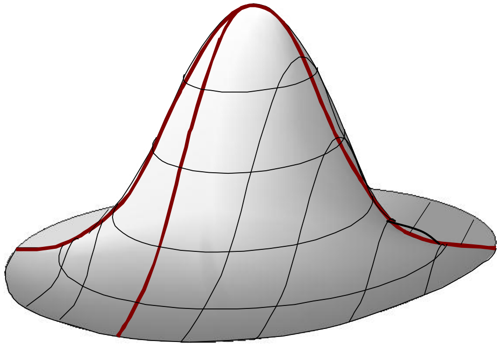
Tensor product basis of cubic monomials

	1	u	u^2	u^3
1	1	u	u^2	u^3
v	v	uv	u^2v	u^3v
v^2	v^2	uv^2	u^2v^2	u^3v^3
v^3	v^3	uv^3	u^2v^3	u^3v^3

Degree Anisotropy: $b_{33}(t,t)$ is of degree 6 in t

Example





Tensor Product Surfaces

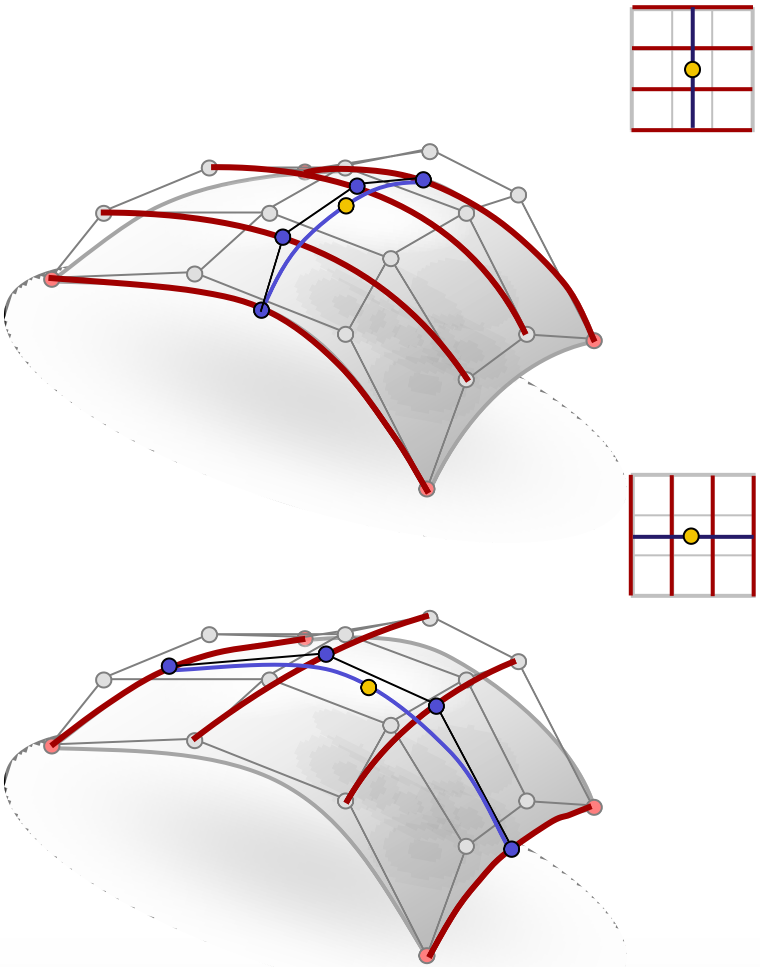
Tensor Product Surfaces:

$$\mathbf{f}(u,v) = \sum_{i=1}^{n}\sum_{j=1}^{n} b_i(u) b_j(v) \mathbf{p}_{i,j}$$

$$= \sum_{i=1}^{n} b_i(u) \sum_{j=1}^{n} b_j(v) \mathbf{p}_{i,j}$$

$$= \sum_{j=1}^{n} b_j(u) \sum_{i=1}^{n} b_i(v) \mathbf{p}_{i,j}$$

- “Curves of Curves”
- Order does not matter



Properties

Properties of tensor product surfaces:

- Linear invariance: Obvious
- Affine invariance?
  - Needs partition of unity property
  - Assume basis $\mathbf{B}^{(\text{curv})} := \{b_1(t), \ldots, b_n(t)\}$ forms a partition of unity, i.e.: $\sum_{i=1}^{n} b_i(v) = 1$
  - Then we get:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} b_i(u)b_j(v) = \sum_{i=1}^{n} b_i(u)\sum_{j=1}^{n} b_j(v) = \sum_{j=1}^{n} b_j(u)\cdot 1 = 1$$

- Affine invariance for tensor product surfaces is induced by the corresponding property of the employed curve basis

Properties

Properties of tensor product surfaces:

- Convex Hull?
  - Assume basis $\mathbf{B}^{(curv)} := \{b_1(t), \ldots, b_n(t)\}$ forms a partition of unity and it is positive (≥ 0) on $t \in [t_0, t_1]$
  - Obviously, we then have:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\underbrace{b_i(u)}_{\geq 0}\underbrace{b_j(v)}_{\geq 0} \geq 0$$

  - So we have the convex hull property on $[t_0, t_1]^2$
- The convex hull property for tensor product surfaces is induced by the property of the employed curve basis.

Partial Derivatives

Computing partial derivatives:

- First derivatives:

$$\frac{\partial}{\partial u}\sum_{i=1}^{n}\sum_{j=1}^{n} b_i(u) b_j(v) \mathbf{p}_{i,j} = \sum_{j=1}^{n} b_j(v) \sum_{i=1}^{n} \left(\frac{d}{du} b_i \right)(u) \mathbf{p}_{i,j}$$

$$\frac{\partial}{\partial v}\sum_{i=1}^{n}\sum_{j=1}^{n} b_i(u) b_j(v) \mathbf{p}_{i,j} = \sum_{i=1}^{n} b_i(u) \sum_{j=1}^{n} \left(\frac{d}{dv} b_j \right)(v) \mathbf{p}_{i,j}$$

- Just spline-curve combinations of curve derivatives

Partial Derivatives

Computing partial derivatives:

- Second derivatives:

$$\frac{\partial^2}{\partial u^2}\sum_{i=1}^{n}\sum_{j=1}^{n} b_i(u)b_j(v)\mathbf{p}_{i,j} = \sum_{j=1}^{n} b_j(v)\sum_{i=1}^{n}\left(\frac{d}{du^2}b_i\right)(u)\mathbf{p}_{i,j}$$

$$\frac{\partial^2}{\partial u \partial v}\sum_{i=1}^{n}\sum_{j=1}^{n} b_i(u)b_j(v)\mathbf{p}_{i,j} = \frac{\partial}{\partial v}\sum_{j=1}^{n} b_j(v)\sum_{i=1}^{n}\left(\frac{d}{du}b_i\right)(u)\mathbf{p}_{i,j}$$

$$= \sum_{j=1}^{n}\left(\frac{d}{dv}b_j\right)(v)\sum_{i=1}^{n}\left(\frac{d}{du}b_i\right)(u)\mathbf{p}_{i,j}$$

Partial Derivatives

Computing partial derivatives:

- General derivatives:

$$\frac{\partial^{r+s}}{\partial u^r \partial v^s} \sum_{i=1}^{n} \sum_{j=1}^{n} b_i(u) b_j(v) \mathbf{p}_{i,j} = \sum_{j=1}^{n} \left(\frac{d^s}{dv^s} b_j \right)(v) \sum_{i=1}^{n} \left(\frac{d^r}{du^r} b_i \right)(u) \mathbf{p}_{i,j}$$

$$= \sum_{i=1}^{n} \left(\frac{d^r}{du^r} b_i \right)(u) \sum_{j=1}^{n} \left(\frac{d^s}{dv^s} b_j \right)(v) \mathbf{p}_{i,j}$$

Normal Vectors

We can compute normal vectors from partial derivatives:

- $$\mathbf{n}(u,v) = \frac{\left(\sum_{j=1}^{n} b_j(v) \sum_{i=1}^{n} \frac{d}{du} b_i(u)\mathbf{p}_{i,j}\right) \times \left(\sum_{j=1}^{n} \frac{d}{dv} b_j(v) \sum_{i=1}^{n} b_i(u)\mathbf{p}_{i,j}\right)}{\left\| \left(\sum_{j=1}^{n} b_j(v) \sum_{i=1}^{n} \frac{d}{du} b_i(u)\mathbf{p}_{i,j}\right) \times \left(\sum_{j=1}^{n} \frac{d}{dv} b_j(v) \sum_{i=1}^{n} b_i(u)\mathbf{p}_{i,j}\right) \right\|}$$
- Problem: degenerate cases
  - Colinear tangents
  - Irregular parametrization
- Need extra code to handle special cases

Bezier Patches

Bezier Patches:

- Use tensor product Bernstein basis

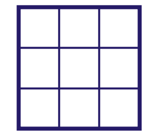
$$\mathbf{f}(u,v)=\sum_{i=0}^{d}\sum_{j=0}^{d}B_i^{(d)}(u)B_j^{(d)}(v)\mathbf{p}_{i,j}$$

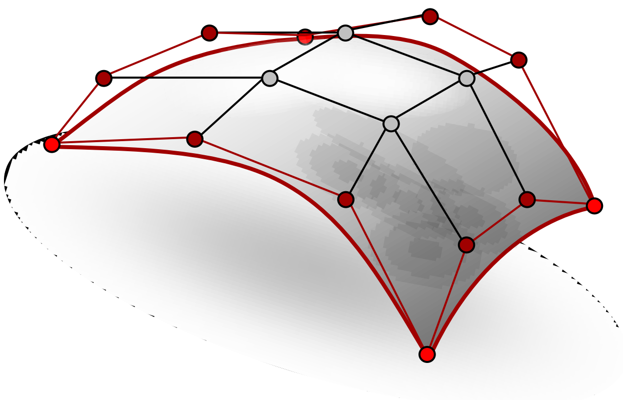
- We get automatically:
  - Affine invariance
  - Convex hull property

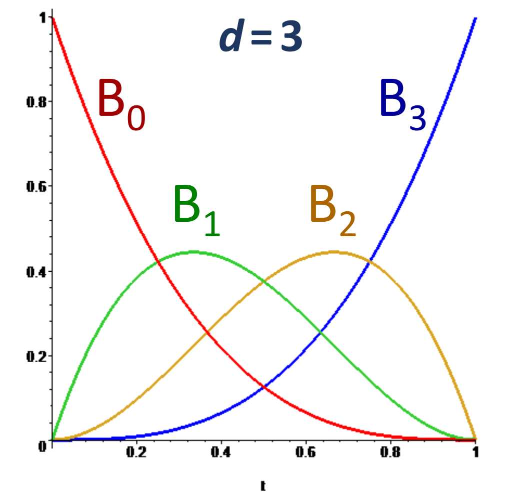
Bezier Patches

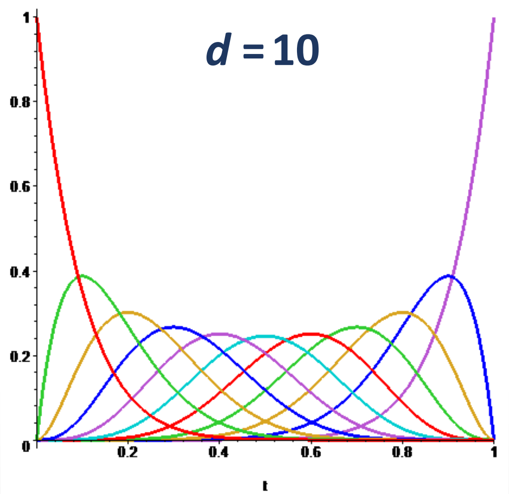
Bezier Patches:

- Interpolation:
  - Boundary curves are Bezier curves of the boundary control points









Bezier Patches

Bezier Patches

- Tangent vectors:
  - First derivatives at boundary points are proportional to differences of control points:

$$\left.\frac{\partial}{\partial u}\mathbf{f}(u,v)\right|_{u=0} = \sum_{i=0}^{d}\sum_{j=0}^{d} B_j^{(d)}(v) B'^{(d)}_i(0)\mathbf{p}_{i,j}$$

$$= d\sum_{j=0}^{d} B_j^{(d)}(v)\left(\mathbf{p}_{1,j} - \mathbf{p}_{0,j}\right)$$

$$\left.\frac{\partial}{\partial u}\mathbf{f}(u,v)\right|_{u=1} = d\sum_{j=0}^{d} B_j^{(d)}(v)\left(\mathbf{p}_{d,j} - \mathbf{p}_{d-1,j}\right)$$

Continuity Conditions

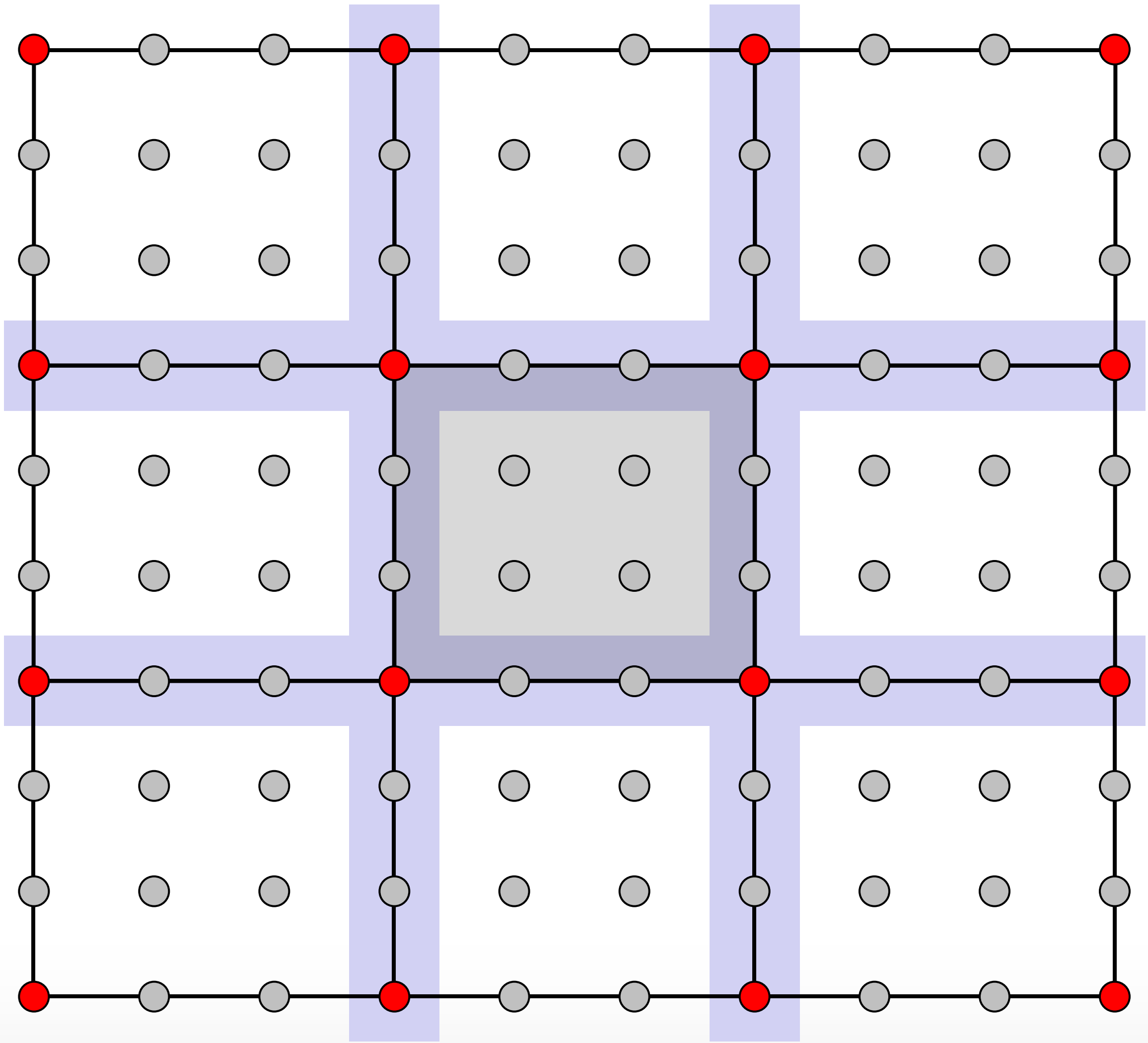
For C^0 continuity:

- Boundary control points must match

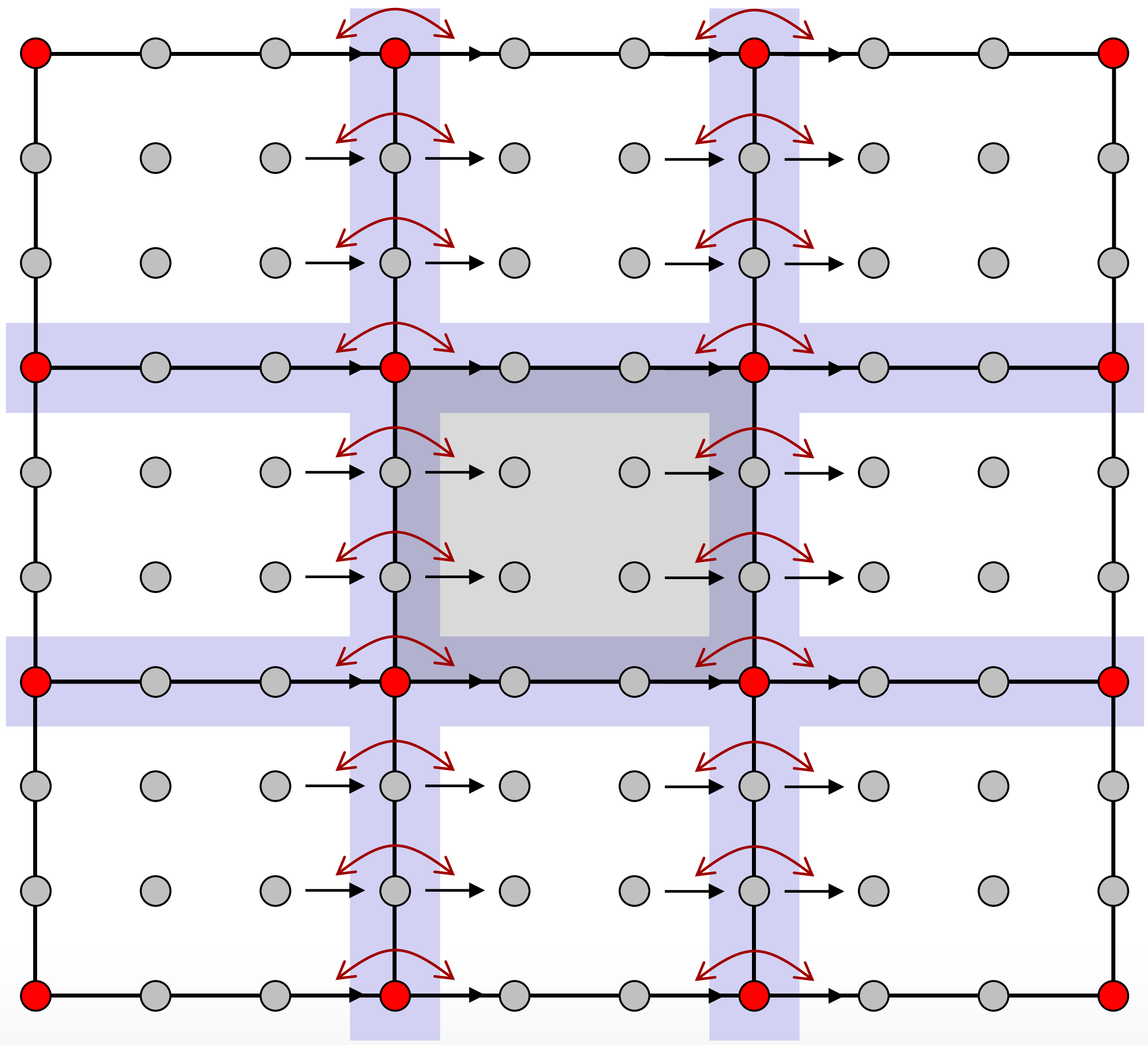
For C^1 continuity:

- Difference vectors must match at the boundary

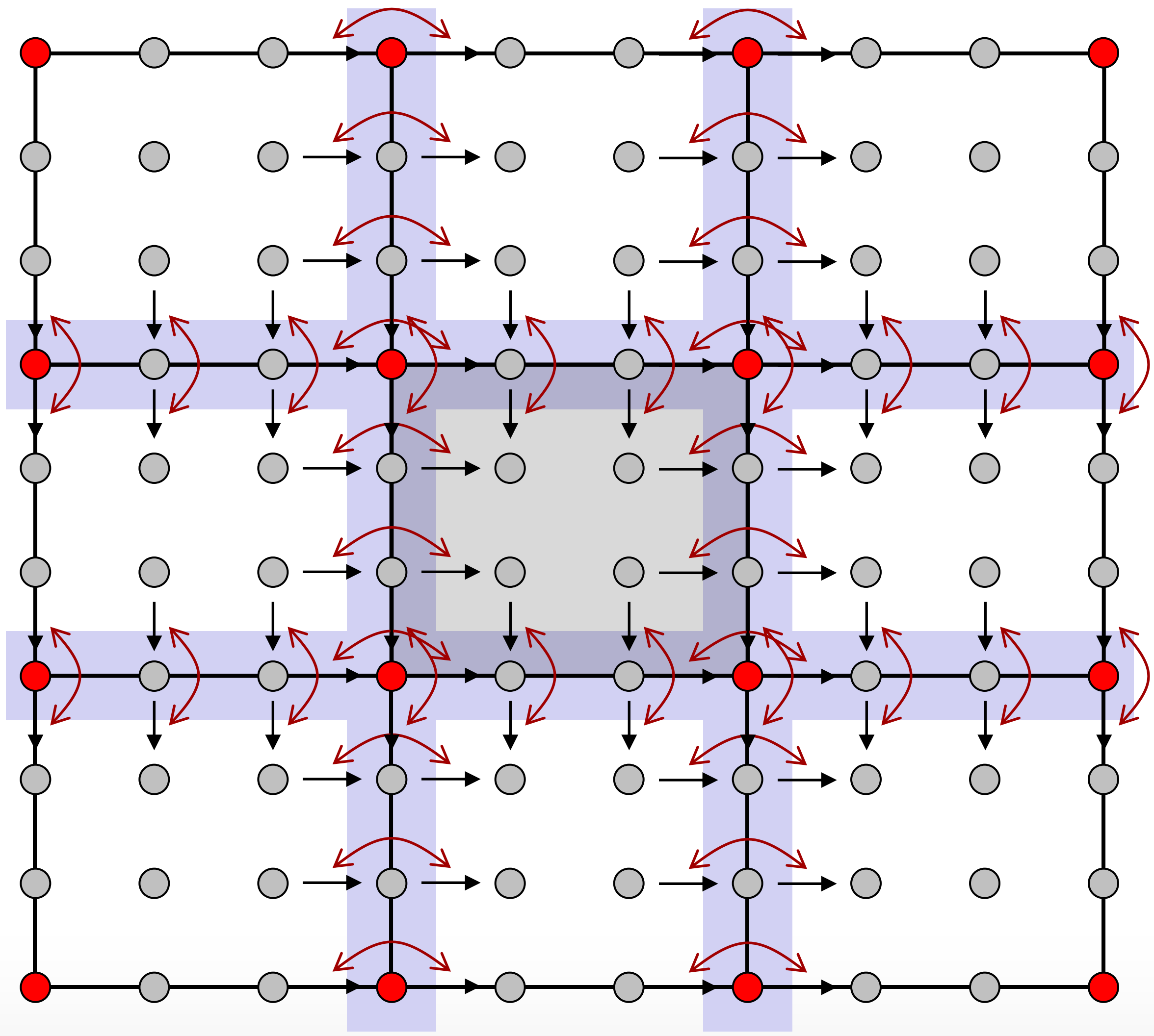
C^0 Continuity



C^1 Continuity



C^1 Continuity



Polars & Blossoms

Blossoms for tensor product surfaces:

- Polar form of a polynomial tensor product surface of degree d:

  $\mathbf{F}: \mathbb{R} \times \mathbb{R} \to \mathbb{R}^n \quad \mathbf{F}(u, v)$

  $\mathbf{f}: \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^n \quad \mathbf{f}(u_1,\dots,u_d; v_1,\dots, v_d)$

- Required Properties:
  - **Diagonality:** $\mathbf{f}(u,\dots,u; v,\dots, v) = \mathbf{F}(u, v)$
  - **Symmetry:** $\mathbf{f}(u_1,\dots,u_d; v_1,\dots, v_d) = \mathbf{f}(u_{\pi(1)},\dots, u_{\pi(d)}; v_{\mu(1)},\dots, v_{\mu(d)})$
    for all permutations of indices π, μ.
  - **Multi-affine:** $\Sigma\alpha_k = 1$
    $\Rightarrow \mathbf{f}(u_1,\dots, \Sigma\alpha_k u_i^{(k)},\dots, u_d; v_1,\dots, v_d)$
    $= \alpha_1 \mathbf{f}(u_1,\dots, u_i^{(1)},\dots, u_d; v_1,\dots, v_d) + \dots + \alpha_n \mathbf{f}(u_1,\dots, u_i^{(n)},\dots, u_d; v_1,\dots, v_d)$
    and $\mathbf{f}(u_1,\dots, u_d; v_1,\dots, \Sigma\alpha_k v_i^{(k)},\dots, v_d)$
    $= \alpha_1 \mathbf{f}(u_1,\dots, u_d; v_1,\dots, v_i^{(1)},\dots, v_d) + \dots + \alpha_n \mathbf{f}(u_1,\dots, u_d; v_1,\dots, v_i^{(n)},\dots, v_d)$

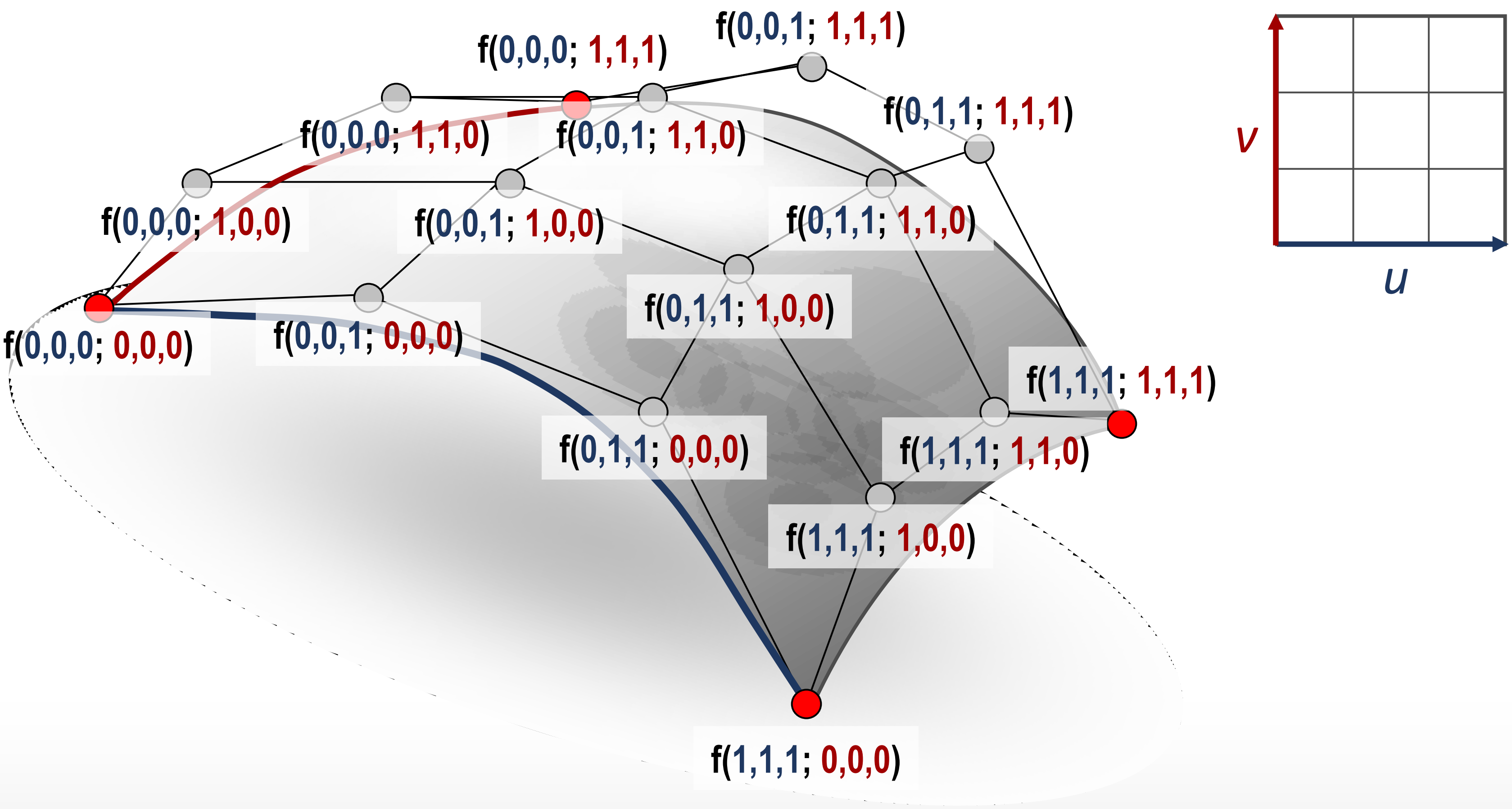
Short Summary

Polar forms for tensor product surfaces:

- Polarize separately in u and v.
- Notation: $\mathbf{f}(\underbrace{u_1,\ldots,u_d}_{u\text{-parameters}};\ \underbrace{v_1,\ldots,v_d}_{v\text{-parameters}})$
- Can be used to derive properties/algorithms similar to the curve case
- More interesting: Polar forms for total degree surfaces (we will see this later)

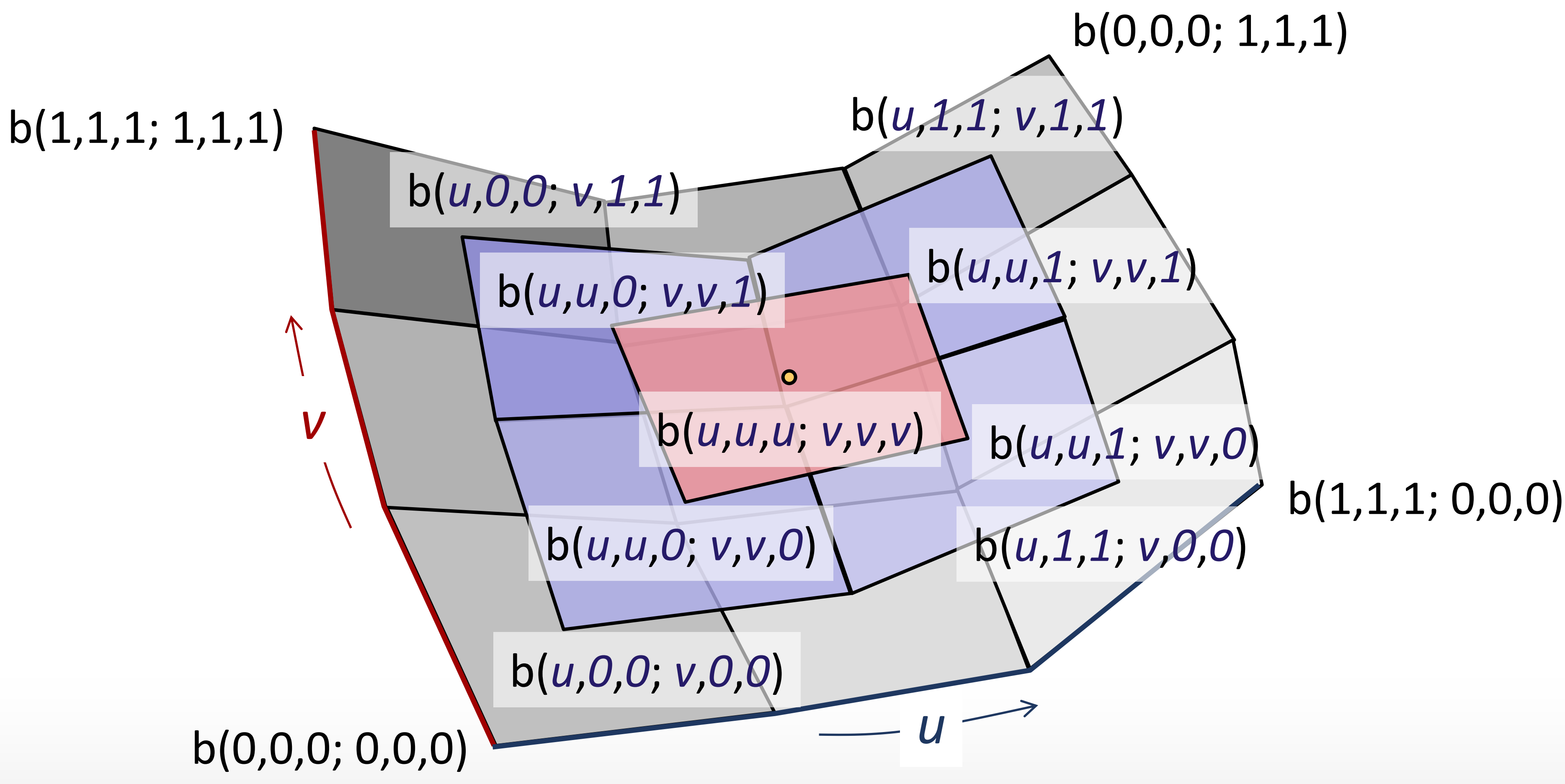
Bezier Control Points

Bezier control points in blossom notation:



De Casteljau Algorithm

De Casteljau algorithm for tensor product surfaces:



B-Spline Patches

B-Spline Patches

- More general than Bezier patches
  (we get Bezier patches as a special case)
- First, we fix a degree d.
- Then, we need knot sequences in u and v direction:
  $(u_1,\ldots,u_n)$, $(v_1,\ldots,v_m)$
- And a corresponding array of control points:

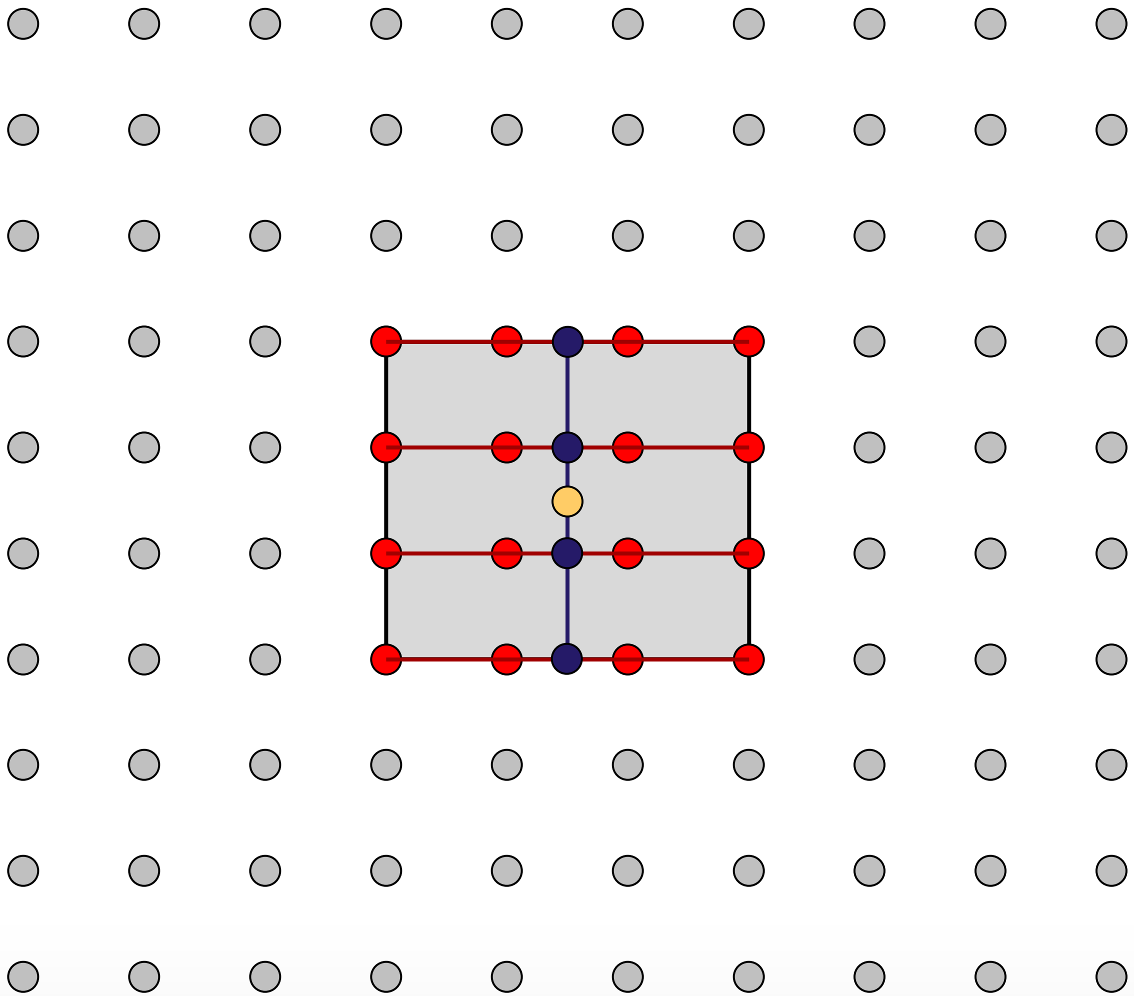
$$
\begin{matrix}
\mathbf{d}_{0,0} & \cdots & \mathbf{d}_{n-d+1,0} \\
\vdots & & \vdots \\
\mathbf{d}_{0,m-d+1} & \cdots & \mathbf{d}_{n-d+1,m-d+1}
\end{matrix}
$$

B-Spline Patches

Then, obtain a parametric B-spline patch as:

- $\mathbf{f}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^{(d)}(u) N_j^{(d)}(v) \mathbf{p}_{i,j}$
- We can evaluate the patches using the de Boor Algorithm:
  - "Curves of curves" idea
  - Determine the knots/control points influencing (u,v).
    These will be no more than (d+1) × (d+1) points.
  - Compute (d+1) v-direction control points along u direction, performing (d+1) curve evaluations.
  - Then evaluate the curve in v-direction.
  - (or the other way round, interchanging u,v-directions)

Illustration



B-Spline Patches

Alternative:

- 2D de Boor algorithm
- Works similar to the 2D de Casteljau algorithm
  but with different weights
  (we can use tensor-product blossoming to derive the weights)

Rational Patches

Rational Patches:

- We can use rational Bezier/B-splines to create the patches (“rational Bezier patches” / “NURBS-patches”)
- Idea:
  - Form a parametric surface in 4D, homogenous space
  - Then project to $\omega = 1$ to obtain the surface in Euclidian 3D space
- In short: Just use homogeneous coordinates everywhere

Rational Patch

Rational Bezier Patch:

$$\mathbf{f}^{(\mathrm{hom})}(u,v) = \sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u) B_j^{(d)}(v) \begin{pmatrix} \omega_{i,j}\mathbf{p}_{i,j} \\ \omega_{i,j} \end{pmatrix}$$

$$\mathbf{f}^{(Eucl)}(u,v) = \frac{\sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u) B_j^{(d)}(v) \mathbf{p}_{i,j}}{\sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u) B_j^{(d)}(v) \omega_{i,j}}$$

Rational Patch

Rational B-Spline Patch:

$$\mathbf{f}^{(\mathrm{hom})}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^{(d)}(u) N_j^{(d)}(v) \begin{pmatrix} \omega_{i,j}\mathbf{p}_{i,j} \\ \omega_{i,j} \end{pmatrix}$$

$$\mathbf{f}^{(Eucl)}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^{(d)}(u) N_j^{(d)}(v)\mathbf{p}_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^{(d)}(u) N_j^{(d)}(v)\omega_{i,j}}$$

Remark: Rational Patches

Observation:

- Euclidian surface is not a tensor product surface
  - denominator depends on both u and v
- Homogeneous space: 4D surface is a tensor product surface.

$$\mathbf{f}^{(Eucl)}(u,v) = \frac{\sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u) B_j^{(d)}(v)\mathbf{p}_{i,j}}{\sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u) B_j^{(d)}(v)\omega_{i,j}}$$

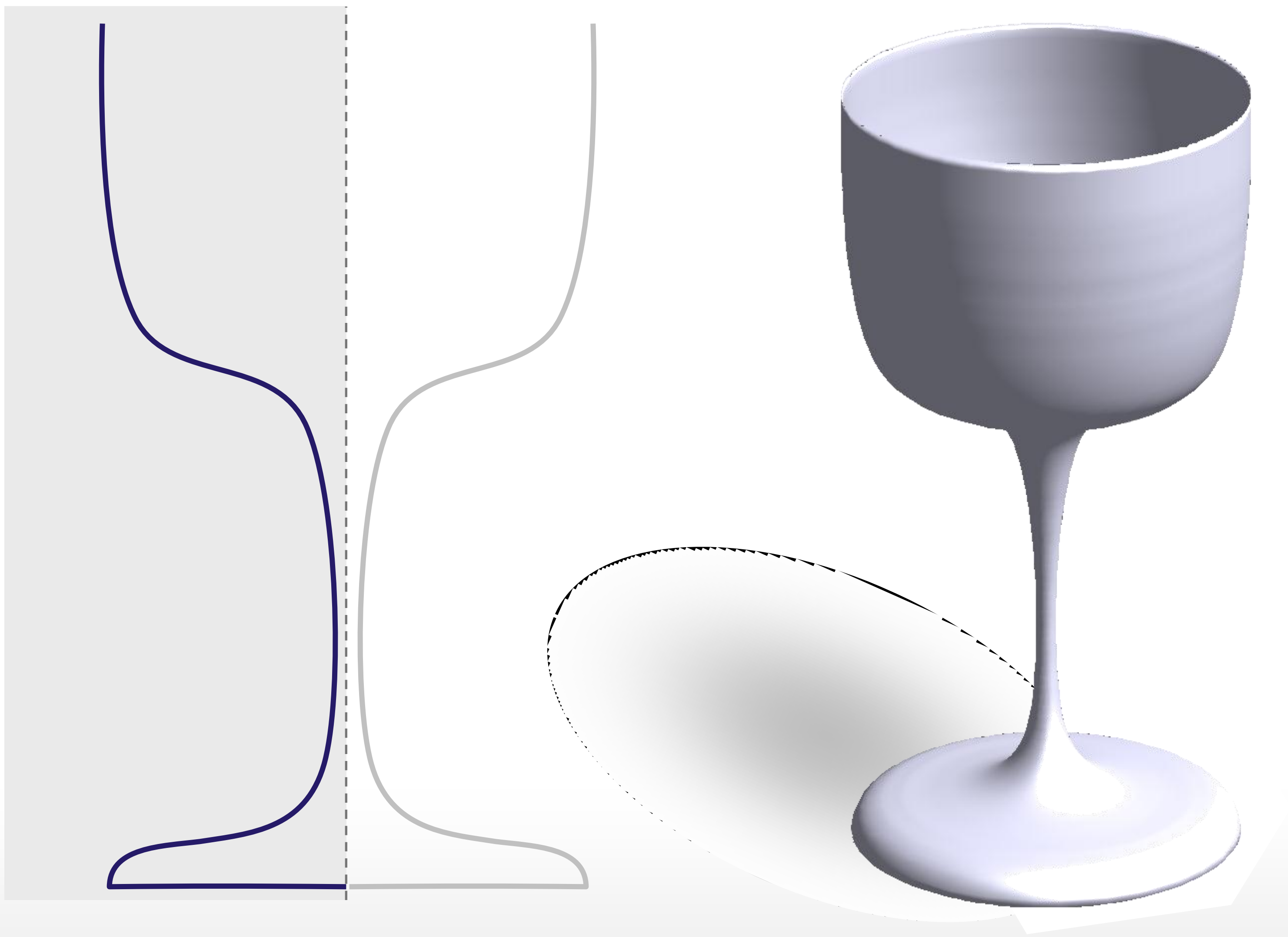
$$\mathbf{f}^{(Eucl)}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^{(d)}(u) N_j^{(d)}(v)\mathbf{p}_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^{(d)}(u) N_j^{(d)}(v)\omega_{i,j}}$$

Surfaces of Revolution

Advantages of rational patches:

- Rational patches can represent surfaces of revolution exactly.
- Examples:
  - Cylinders
  - Cones
  - Spheres
  - Ellipsoids
  - Tori
- Question: given a cross section curve, how do we get the control points for the 3D surface?

Surfaces of Revolution



Surfaces of Revolution

Given:

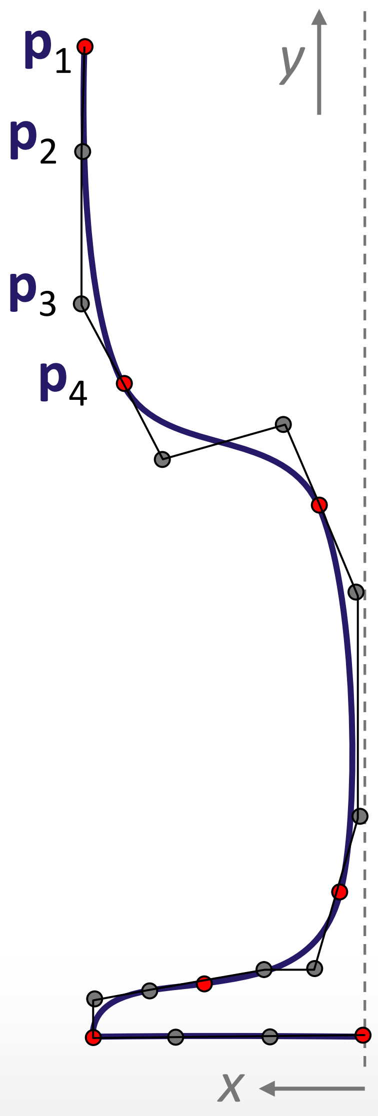
- Control points $\mathbf{p}_1,\ldots,\mathbf{p}_n$ of curve ("generatrix")

We want to compute:

- Control points $\mathbf{p}_{i,j}$ of a rational surface

Such that:

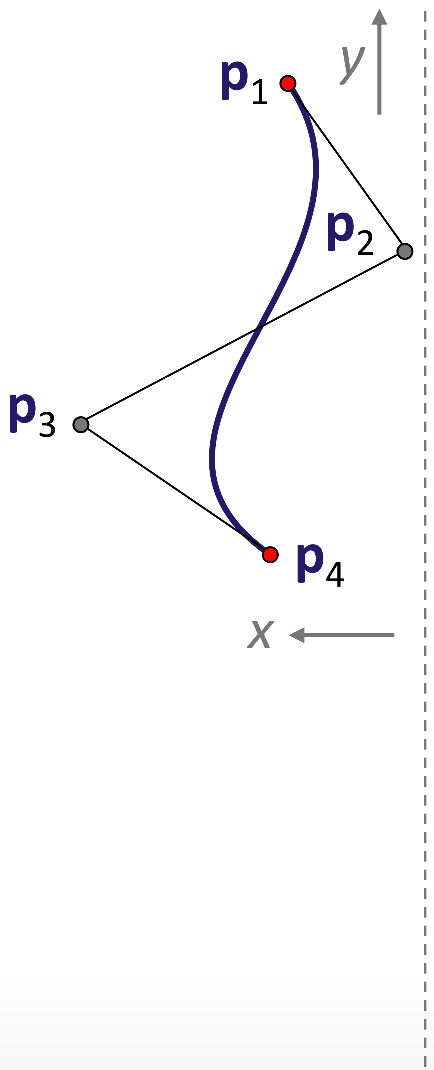
- The surface describes the surface of revolution that we obtain by rotating the curve around the y axis (w.l.o.g.)



Surfaces of Revolution

Simplification:

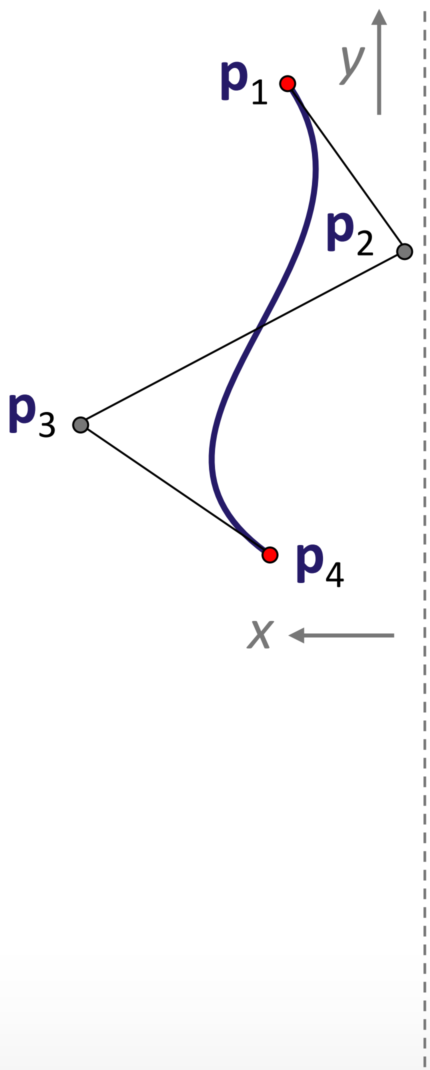
- We look only at a single rational Bezier segment.
- Applying the scheme to multiple segments together is straightforward.
- The same idea also works for B-splines.



Surfaces of Revolution

Construction:

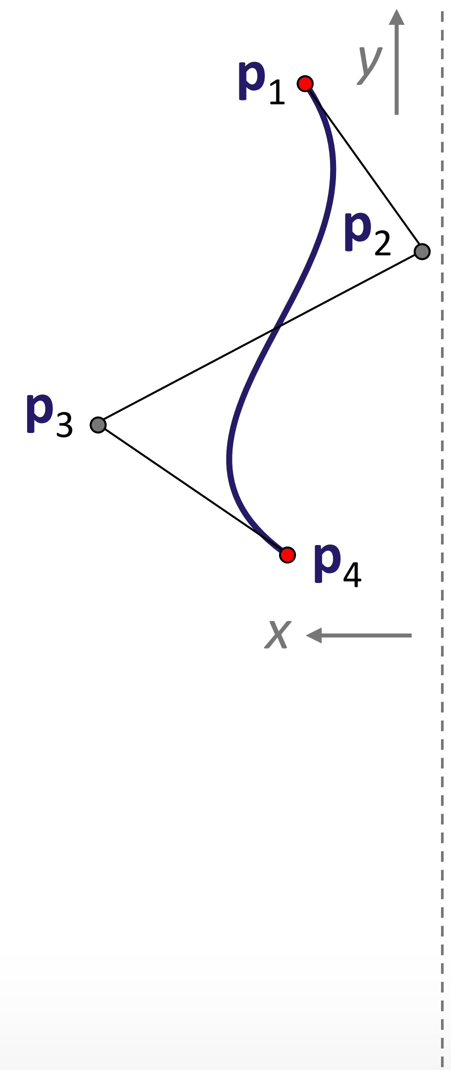
- We are given control points $\mathbf{p}_1, \ldots, \mathbf{p}_{d+1}$ (d is the degree in u direction)
- We introduce a new parameter v.
- In v direction, we use quadratic Bezier curves (2nd degree basis in v-direction)



Surfaces of Revolution

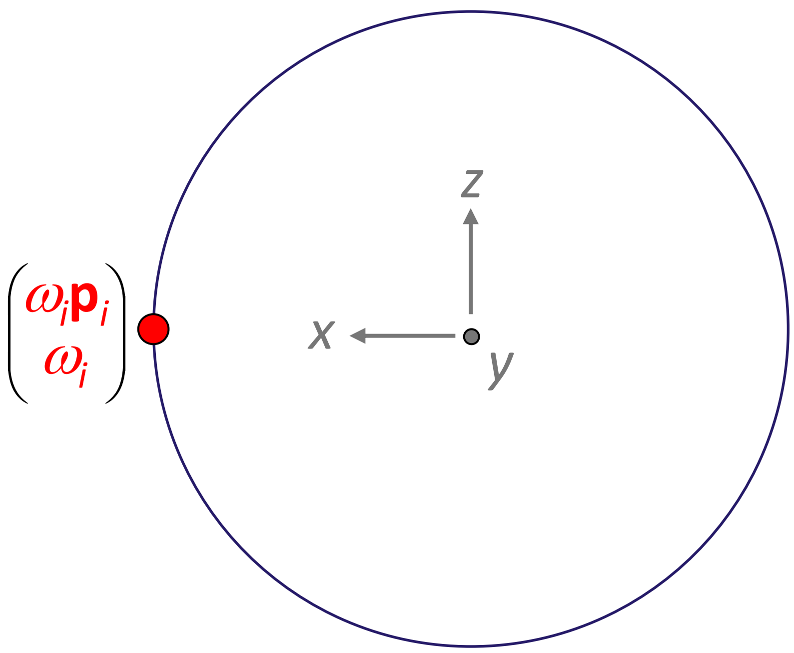
Key Idea:

- For *u*-direction curves: Control points (and thus the curves) must move on circles around the *y*-axis.
- Circles must have the same parametrization (this is easy)
- This means, the control points rotate around the *y*-axis.
- Affine invariance will make the whole curve rotate, we get the desired surface of revolution.



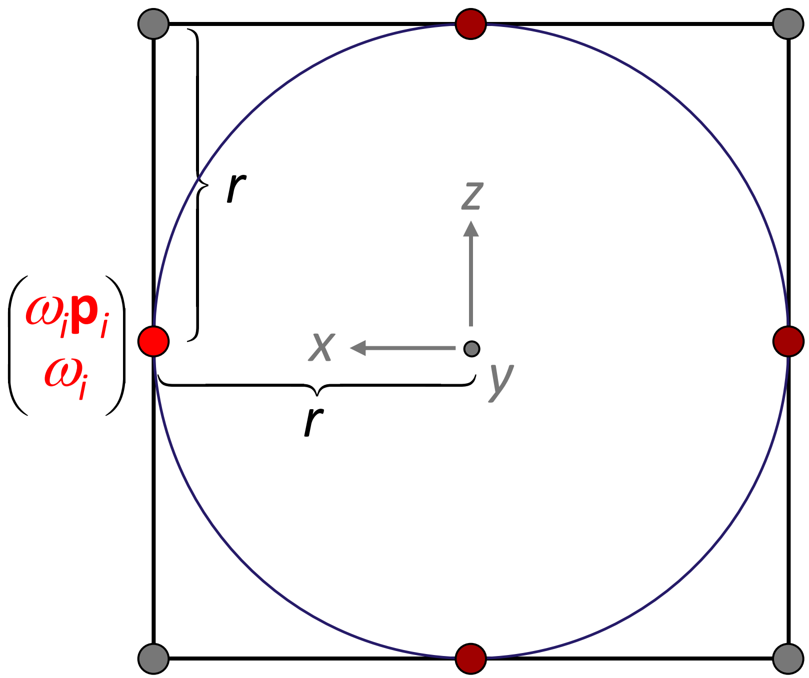
Surfaces of Revolution

Making one point rotate around the y-axis:



Surfaces of Revolution

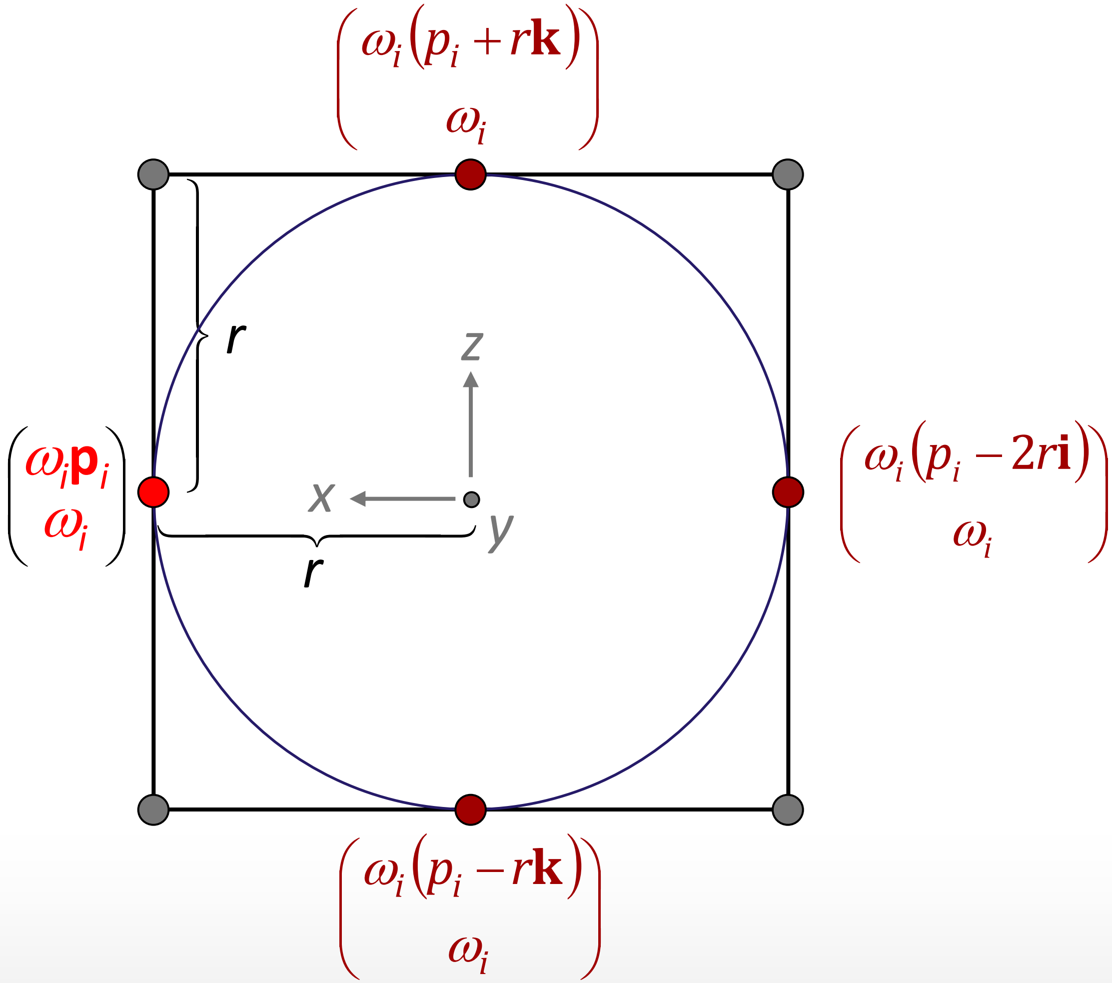
Making one point rotate around the y-axis:



Surfaces of Revolution

Making one point rotate around the y-axis:

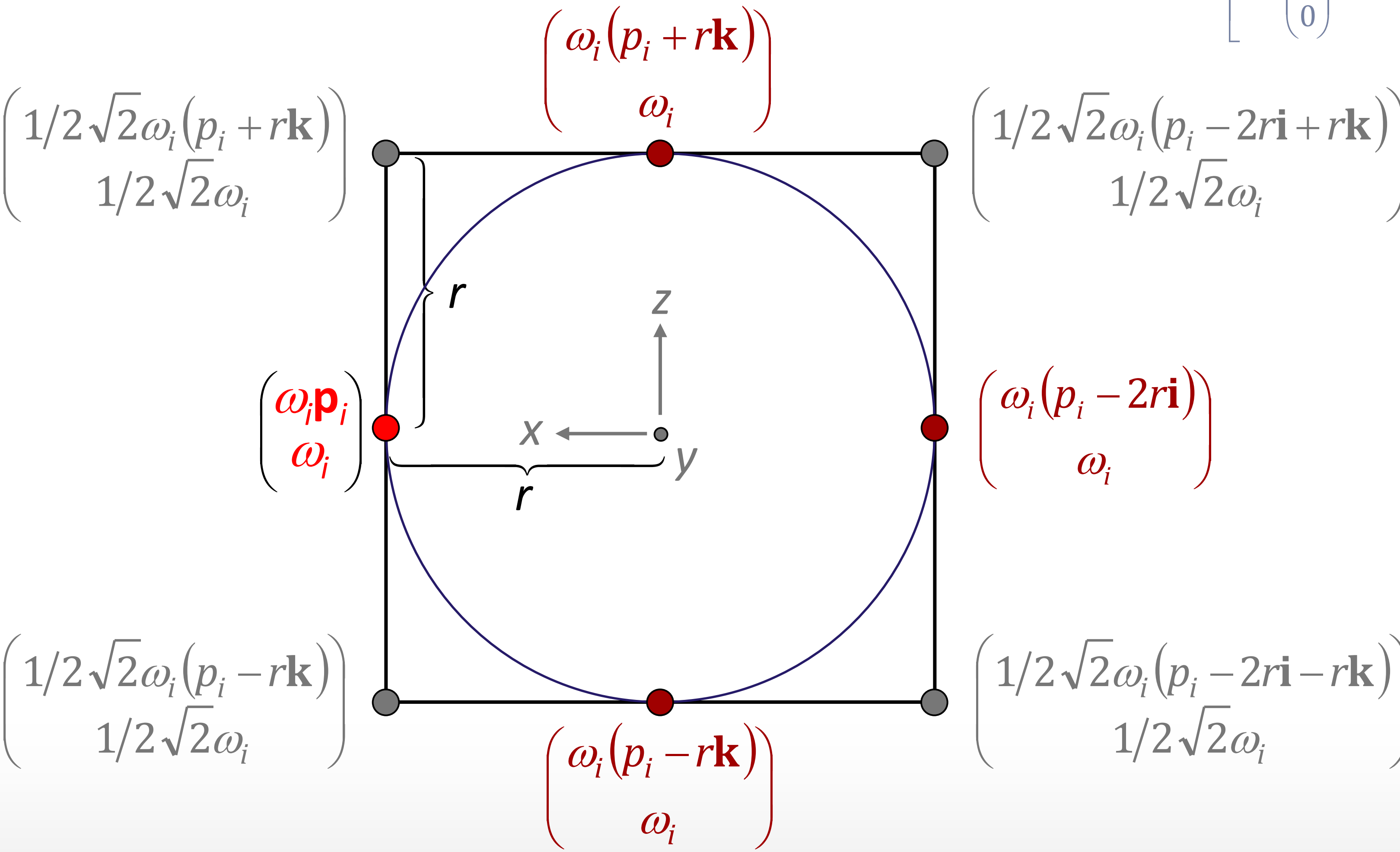
$$\left[\mathbf{i} := \begin{pmatrix}1\\0\\0\end{pmatrix}, \mathbf{k} := \begin{pmatrix}0\\0\\1\end{pmatrix}\right]$$



Surfaces of Revolution

Making one point rotate around the y-axis:

$$\left[\mathbf{i} := \begin{pmatrix}1\\0\\0\end{pmatrix},\ \mathbf{k} := \begin{pmatrix}0\\0\\1\end{pmatrix}\right]$$



Remark

What we get:

- We obtain 4 segments, i.e. 4 patches for each Bezier segment
- A similar construction with 3 segments exists as well

Does the scheme yield a circle for weights $\neq 1$ in the generatrix curve?

- Common factors in weights cancel out
- Therefore, we still obtain a circle at these points
- Parametrization does not change either

Benefit

With this construction, it is straightforward to create:

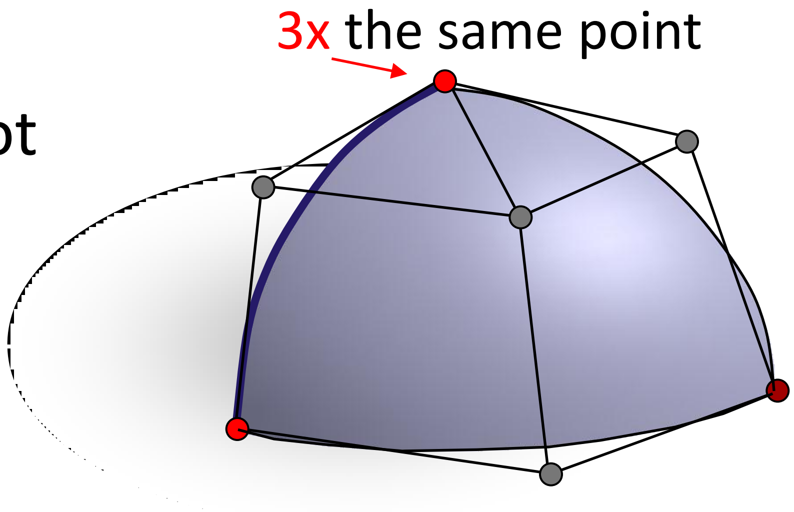
- Spheres
- Tori
- Cylinders
- Cones

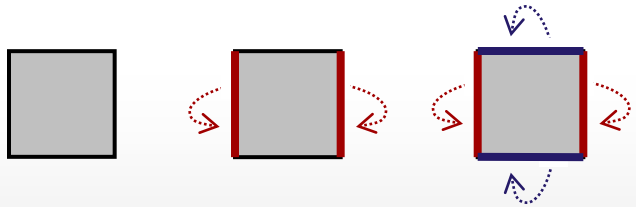
And affine transformations of these (e.g. ellipsoids)

Parametrization Restrictions

Remaining problem:

- The sphere and the cone are not regularly parametrized (double control points)
- Might cause trouble (normals computation, tessellation)
- In general: no spheres, or n-tori (n > 1) can be parametrized without degeneracies
- What works: open surfaces, cylinders, tori





Curves on Surfaces, trimmed NURBS

Quad patch problem:

- All of our shapes are parameterized over rectangular regions
- General boundary curves are hard to create
- Topology fixed to a disc (or cylinder, torus)
- No holes in the middle
- Assembling complicated shapes is painful
  - Lots of pieces
  - Continuity conditions for assembling pieces become complicated
  - Cannot use C^2 B-Splines continuity along boundaries when using multiple pieces

Curves on Surfaces, trimmed NURBS

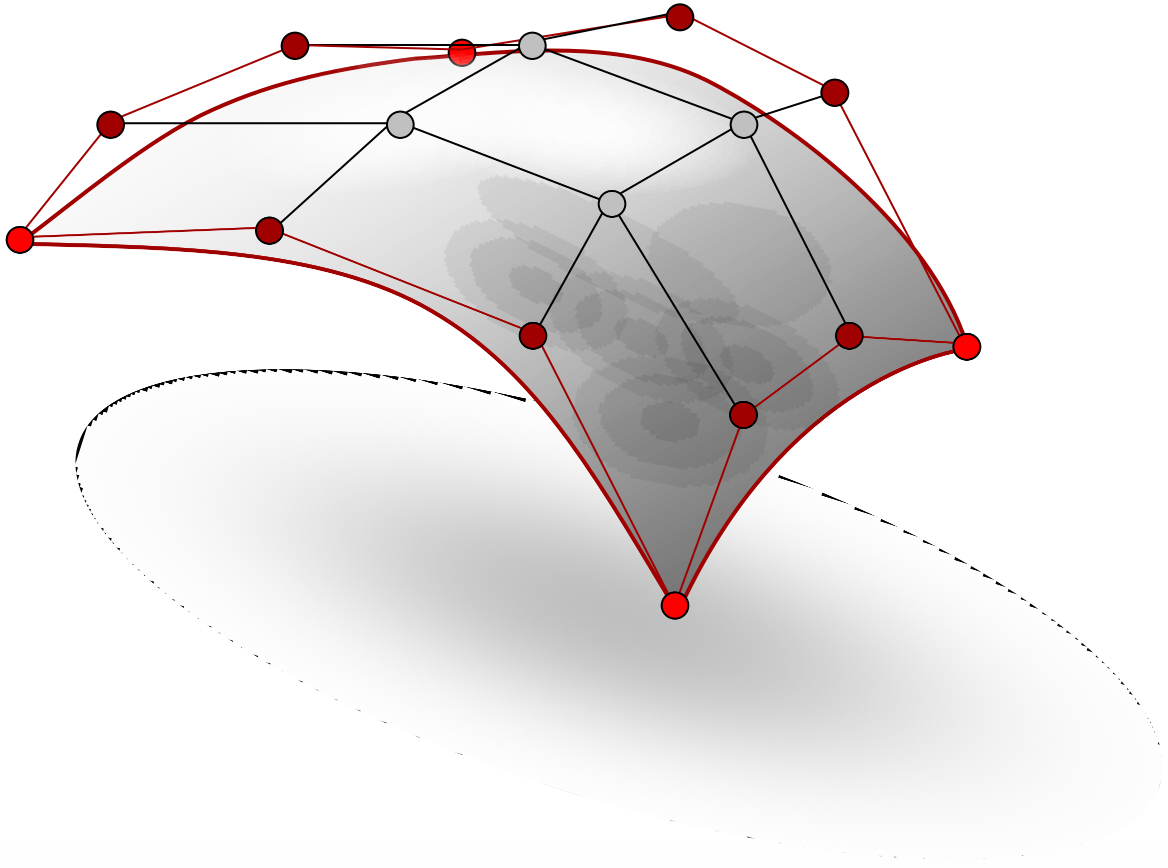
Consequence:

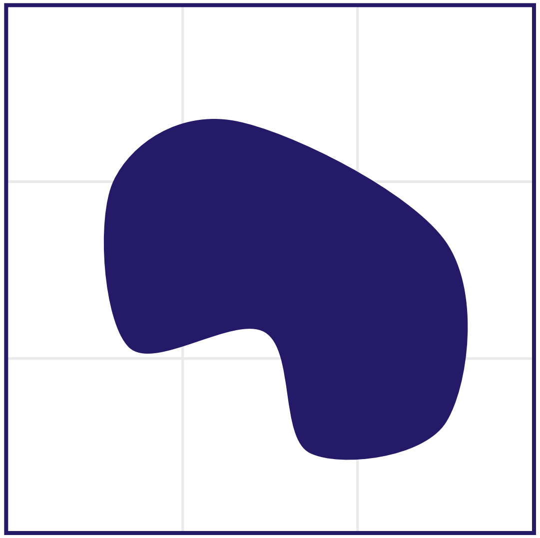
- We need more control over the parameter domain
- One solution is *trimming* using *curves on surfaces (CONS)*
- Standard tool in CAD: *trimmed NURBS*

Basic idea:

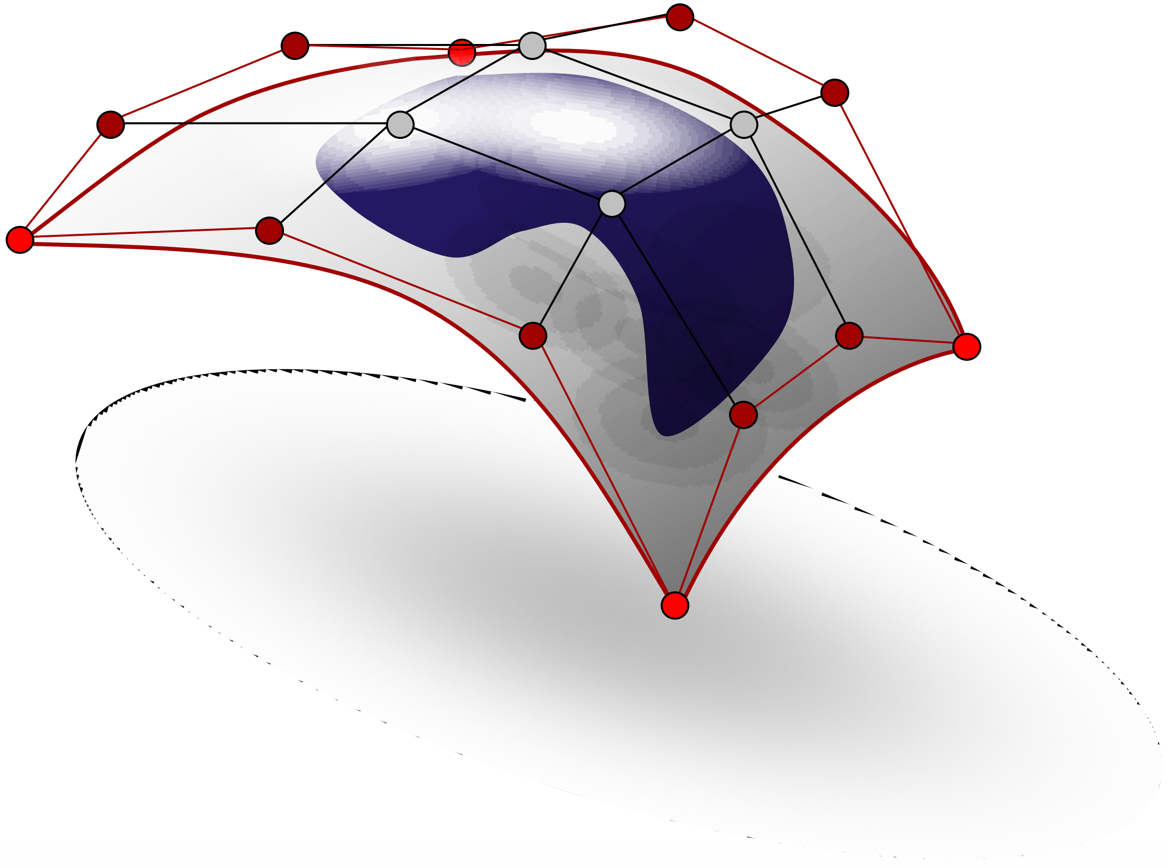
- Specify a curve in the parameter domain that encapsulates one (or more) pieces of area
- Tessellate the parameter domain accordingly to cut out the trimmed piece (rendering)

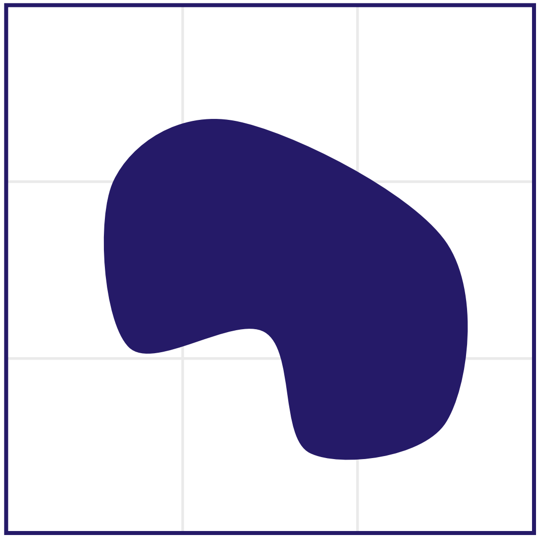
Curves-on-Surfaces (CONS)



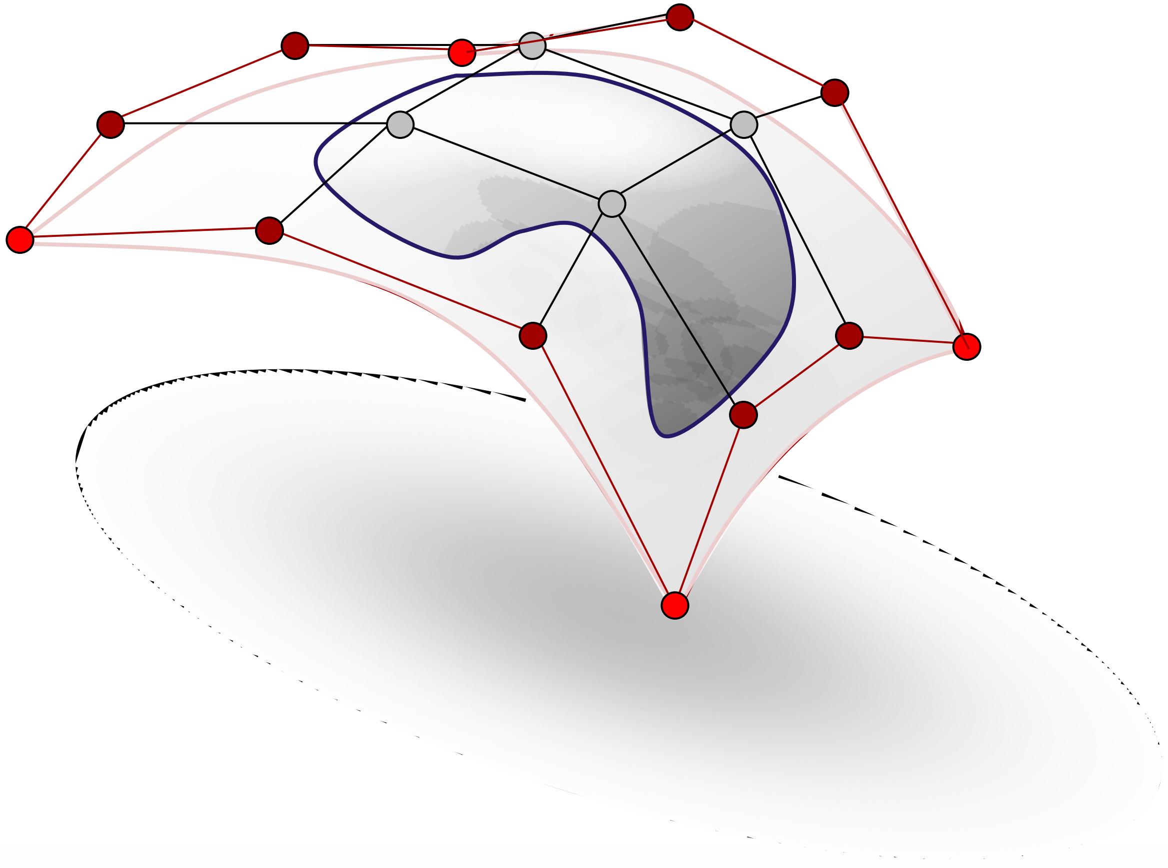


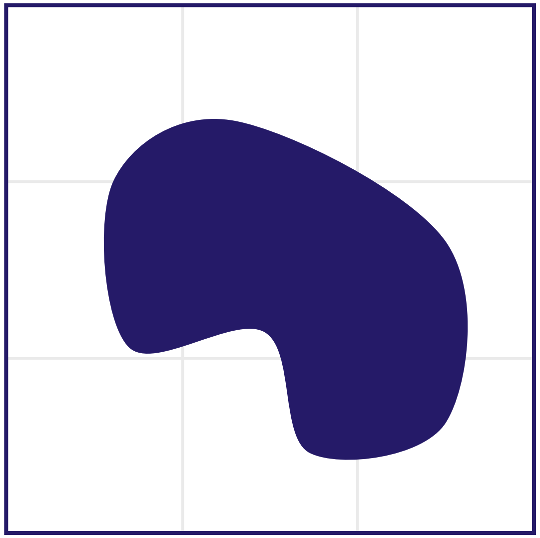
Curves-on-Surfaces (CONS)





Curves-on-Surfaces (CONS)

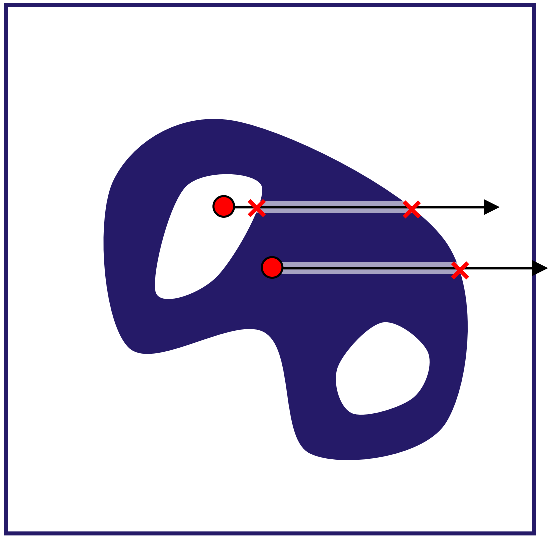


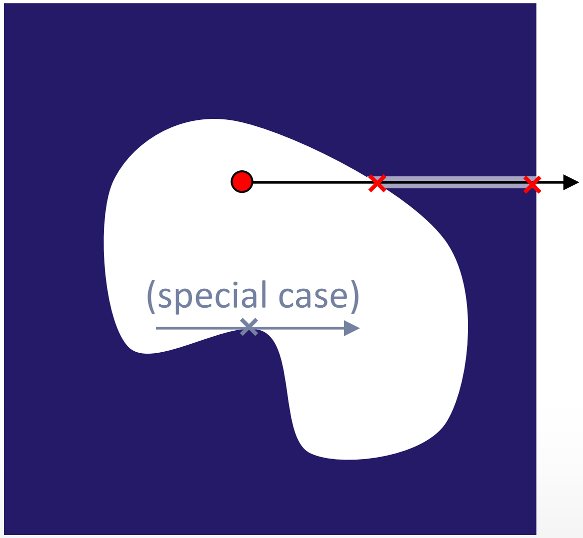


General Shapes

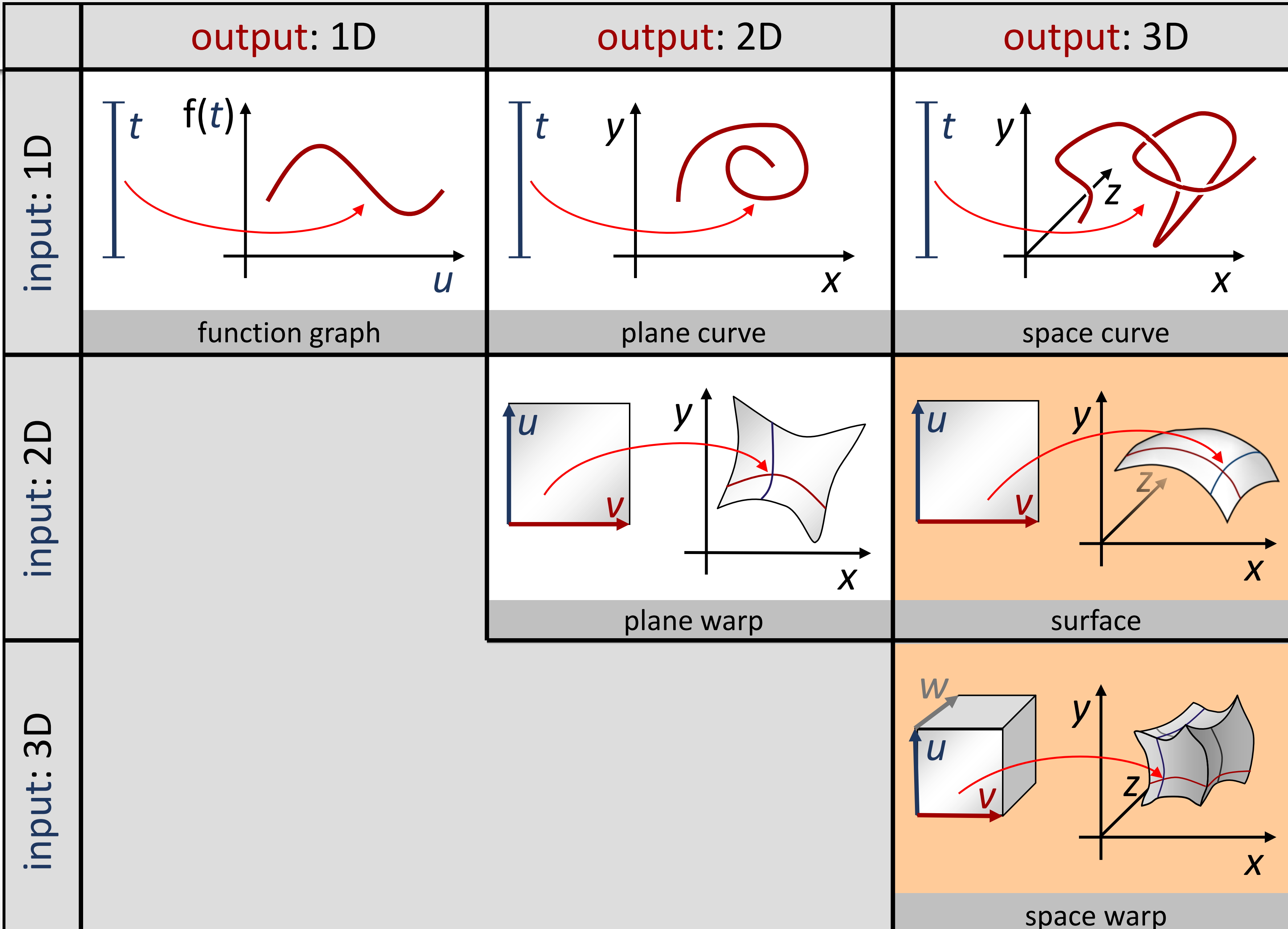
General shapes with holes:

- Draw multiple curves
- Inside / outside test:
  - If any ray in the parameter domain intersects the boundary curves an odd number of times, the point is inside
  - Outside otherwise
  - Implementation needs to take care of special cases (critical points with respect to normal of the ray)
  - Nasty, but doable





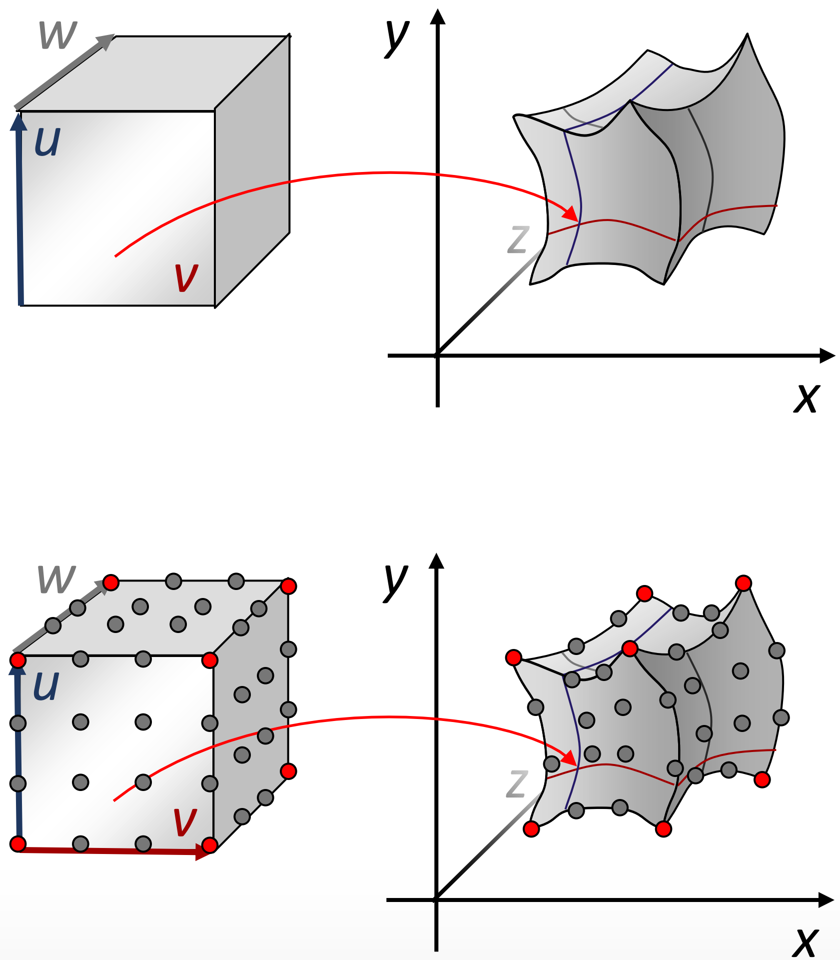
Free Form Deformation


output: 1D
output: 2D
output: 3D
input: 1D
input: 2D
input: 3D
t
f(t)
u
function graph
t
y
x
plane curve
t
y
z
x
space curve
u
v
y
x
plane warp
u
v
y
z
x
surface
w
u
v
y
z
x
space warp

FFD

Free Form Deformations

- Use a 3D tensor-product B-Spline (or Bezier spline)
- Defines a bend mapping $\mathbb{R}^3 \rightarrow \mathbb{R}^3$
- Can be used to change the shape of objects globally
- We will see other shape deformation techniques later in the lecture (time permitting)

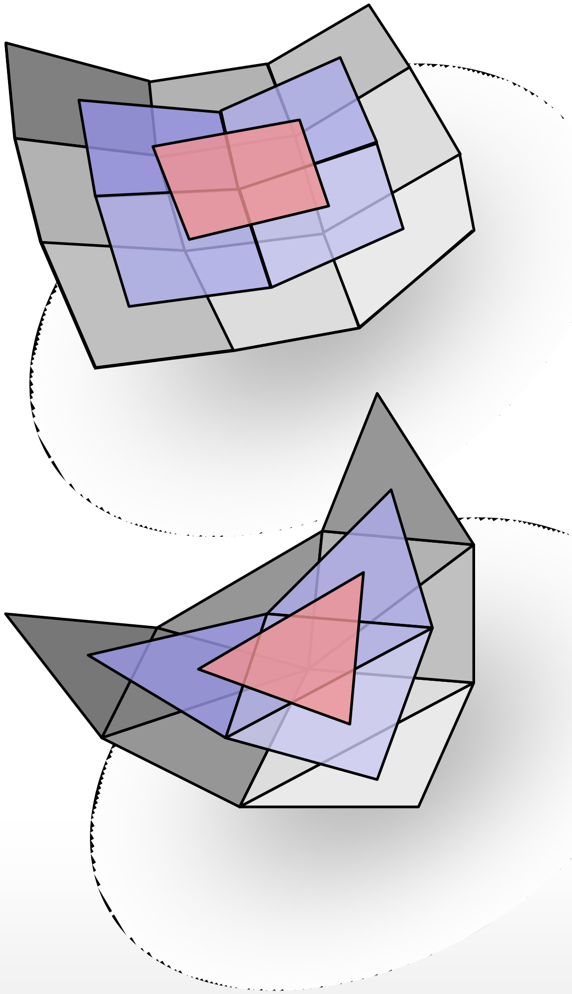


Total Degree Surfaces

Bezier Triangles

Alternative surface definition: Bezier triangles

- Constructed according to given total degree
  - Completely symmetric: No degree anisotropy
- Can be derived using a triangular de Casteljau algorithm
  - Blossoming formalism is very helpful for defining Bezier Triangles
  - Barycentric interpolation of blossom values



Blossoms for Total Degree Surfaces

Blossoms with points as arguments:

- Polar form degree d with points as input und output:

  $\mathbf{F}: \mathbb{R}^{n} \to \mathbb{R}^{m}$

  $\mathbf{f}: \mathbb{R}^{d \times n} \to \mathbb{R}^{m}$

  points as arguments

- Required Properties:
  - Diagonality: $\mathbf{f}(\mathbf{t}, \mathbf{t}, \ldots, \mathbf{t}) = \mathbf{F}(\mathbf{t})$
  - Symmetry: $\mathbf{f}(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_d) = \mathbf{f}(\mathbf{t}_{\pi(1)}, \mathbf{t}_{\pi(2)}, \ldots, \mathbf{t}_{\pi(d)})$
    for all permutations of indices π.
  - Multi-affine: $\Sigma\alpha_k = 1$

    $\Rightarrow \mathbf{f}(\mathbf{t}_1, \mathbf{t}_2, \ldots, \Sigma\alpha_k \mathbf{t}_i^{(k)}, \ldots, \mathbf{t}_d)$

    $= \alpha_1 \mathbf{f}(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_i^{(1)}, \ldots, \mathbf{t}_d) + \ldots + \alpha_n \mathbf{f}(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_i^{(n)}, \ldots, \mathbf{t}_d)$

Example

Example: bivariate monomial basis

- In powers of (u,v):
  $B = \{1, u, v, u^2, uv, v^2\}$
- Blossom form: multilinear in (u_1,u_2,v_1,v_2)

$$B = \Big\{1,$$

$$\frac{1}{2}(u_1 + u_2),\ \ \frac{1}{2}(v_1 + v_2),$$

$$u_1 u_2,\ \ \frac{1}{4}(u_1 v_1 + u_1 v_2 + u_2 v_1 + v_2 u_2),\ \ v_1 v_2 \Big\}$$

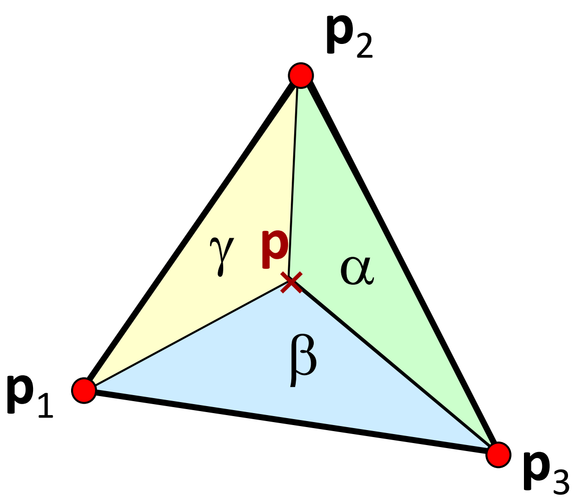
Barycentric Coordinates

Barycentric Coordinates:

- Planar case:
  Barycentric combinations of 3 points

  $$\mathbf{p} = \alpha \mathbf{p}_1 + \beta \mathbf{p}_2 + \gamma \mathbf{p}_3, \text{with} : \alpha + \beta + \gamma = 1$$

  $$\gamma = 1 - \alpha - \beta$$



- Area formulation:

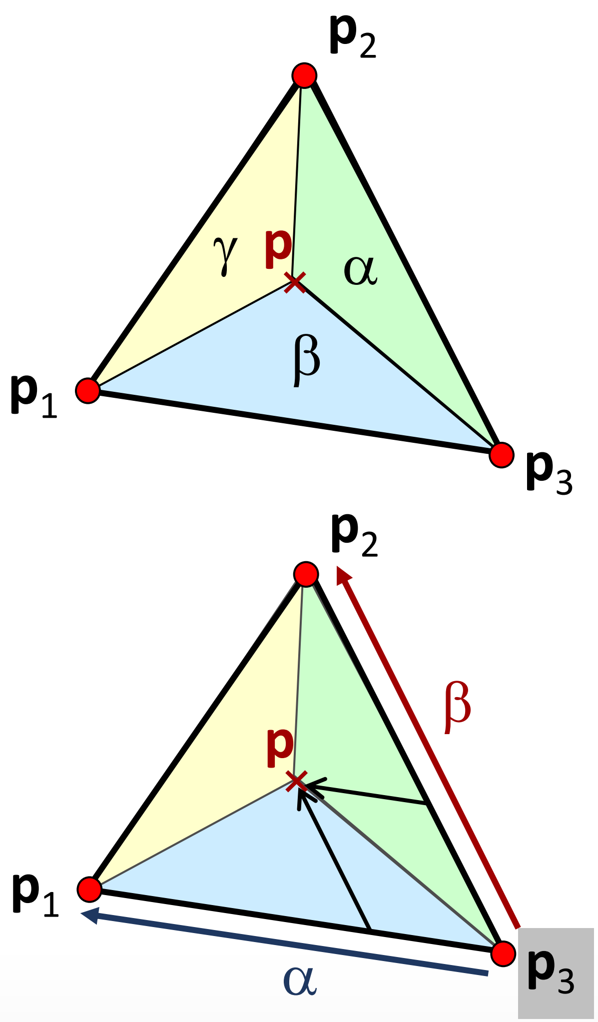
$$\alpha = \frac{area(\Delta(\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}))}{area(\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3))}, \beta = \frac{area(\Delta(\mathbf{p}_1, \mathbf{p}_3, \mathbf{p}))}{area(\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3))}, \gamma = \frac{area(\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}))}{area(\Delta(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3))}$$

Barycentric Coordinates

Barycentric Coordinates:

- Linear formulation:

$$\begin{aligned}\mathbf{p} &= \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3 \\ &= \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + (1-\alpha-\beta)\mathbf{p}_3 \\ &= \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \mathbf{p}_3 - \alpha\mathbf{p}_3 - \beta\mathbf{p}_3 \\ &= \mathbf{p}_3 + \underline{\alpha(\mathbf{p}_1 - \mathbf{p}_3)} + \underline{\beta(\mathbf{p}_2 - \mathbf{p}_3)}\end{aligned}$$



Bezier Triangles: Overview

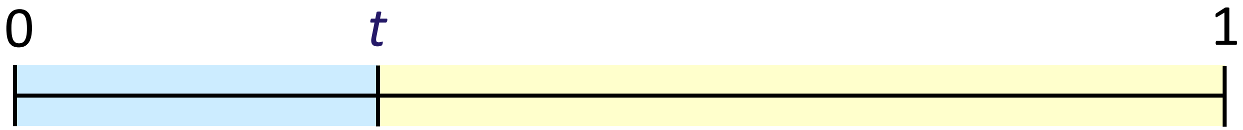
Bezier Triangles: Main Ideas

- Use 3D points as inputs to the blossoms
- These are Barycentric coordinates of a parameter triangle {**a**, **b**, **c**}
- Use 3D points as outputs
- Form control points by multiplying parameter points, just as in the curve case: $\mathbf{p}(\underbrace{\mathbf{a},\ldots,\mathbf{a}}_{i}, \underbrace{\mathbf{b},\ldots,\mathbf{b}}_{j}, \underbrace{\mathbf{c},\ldots,\mathbf{c}}_{k})$
- De Casteljau Algorithm: Compute polynomial values **p**(**x**, ..., **x**) by barycentric interpolation

Plugging in the Barycentric Coord's

Analogy: 2D Curves in barycentric coordinates

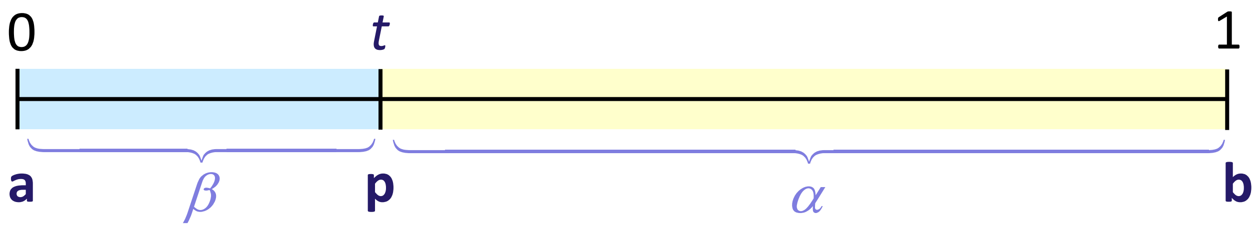
- Barycentric coordinates for 2D curves:



Plugging in the Barycentric Coord's

Analogy: 2D Curves in barycentric coordinates

- Barycentric coordinates for 2D curves:



- $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b}, \quad \alpha + \beta = 1$

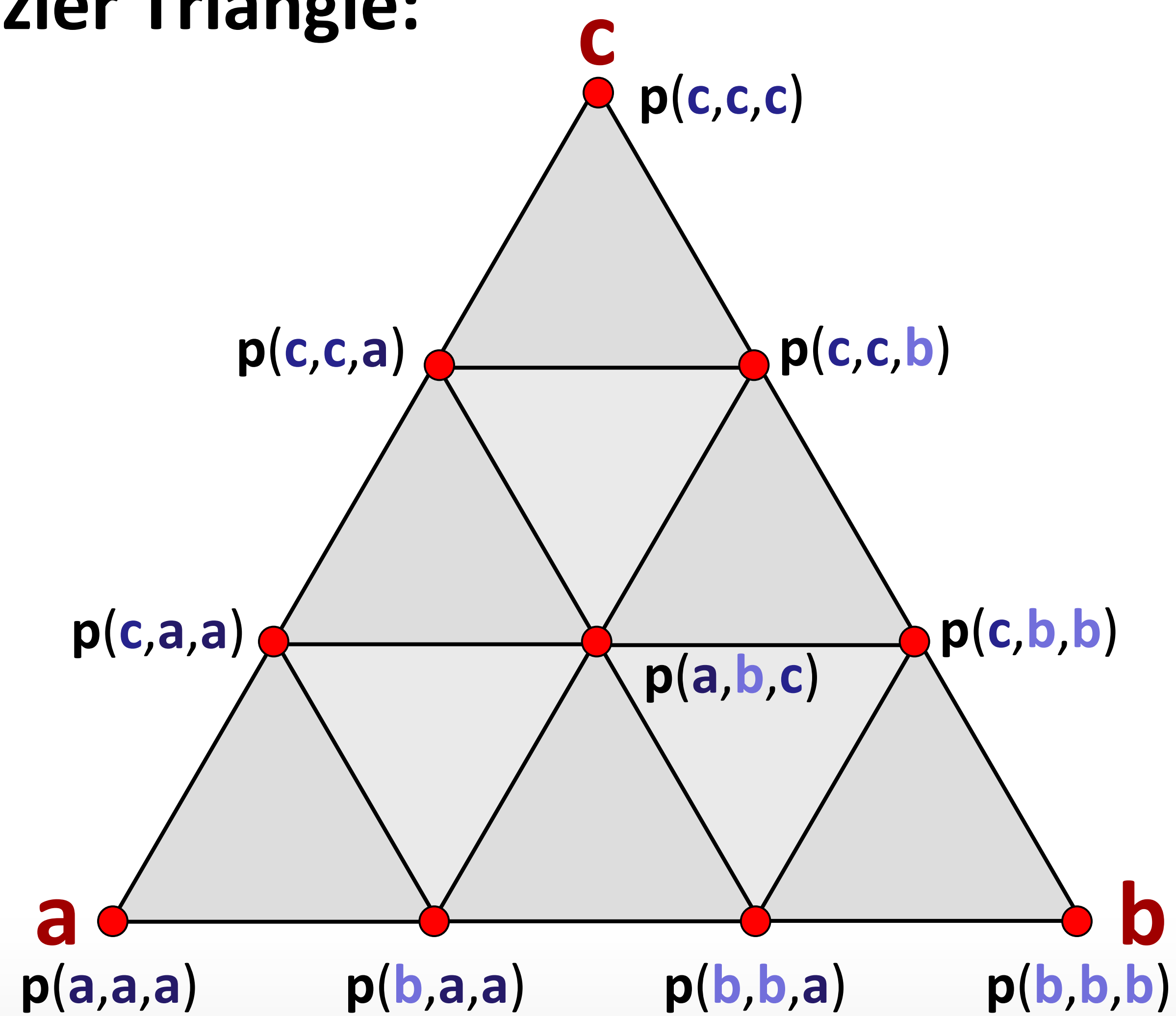
- Bezier splines:

$$\mathbf{F}(t) = \sum_{i=0}^{d} \binom{d}{i} (1-t)^i t^{d-i} \mathbf{f}(\underbrace{\mathbf{a},\ldots,\mathbf{a}}_{i}, \underbrace{\mathbf{b},\ldots,\mathbf{b}}_{d-i})$$ (standard form)

$$\mathbf{F}(\mathbf{p}) = \sum_{\substack{i+j=d \\ i\geq 0, j\geq 0}} \frac{d!}{i!\,j!} \alpha^i \beta^j \mathbf{f}(\underbrace{\mathbf{a},\ldots,\mathbf{a}}_{i}, \underbrace{\mathbf{b},\ldots,\mathbf{b}}_{j})$$ (barycentric form)

Example

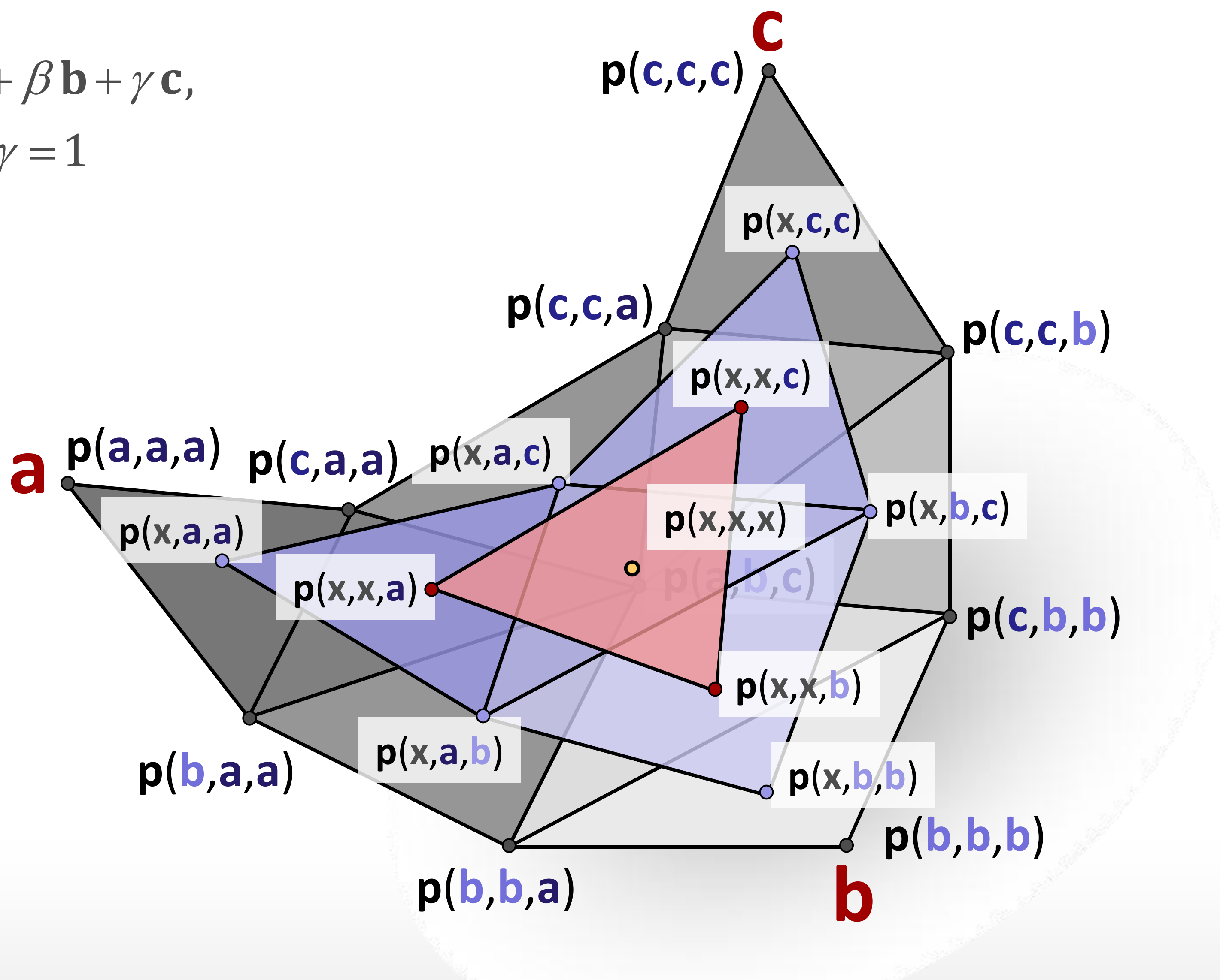
Cubic Bezier Triangle:



De Casteljau Algorithm

$$\mathbf{x} = \alpha\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c},$$

$$\alpha + \beta + \gamma = 1$$



Bernstein Form

Writing this recursion out, we obtain:

- $$F(\mathbf{x}) = \sum_{\substack{i+j+k=d \\ i,j,k \geq 0}} \frac{d!}{i!\,j!\,k!} \alpha^i \beta^j \gamma^k \mathbf{f}(\underbrace{a,\ldots,a}_{i}, \underbrace{b,\ldots,b}_{j}, \underbrace{c,\ldots,c}_{k})$$

  $$\mathbf{x} = \alpha\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c},$$

  $$\alpha + \beta + \gamma = 1$$

- This is the *Bernstein form* of a Bezier triangle surface
- (Proof by induction)

Rendering

Rendering trimmed NURBS

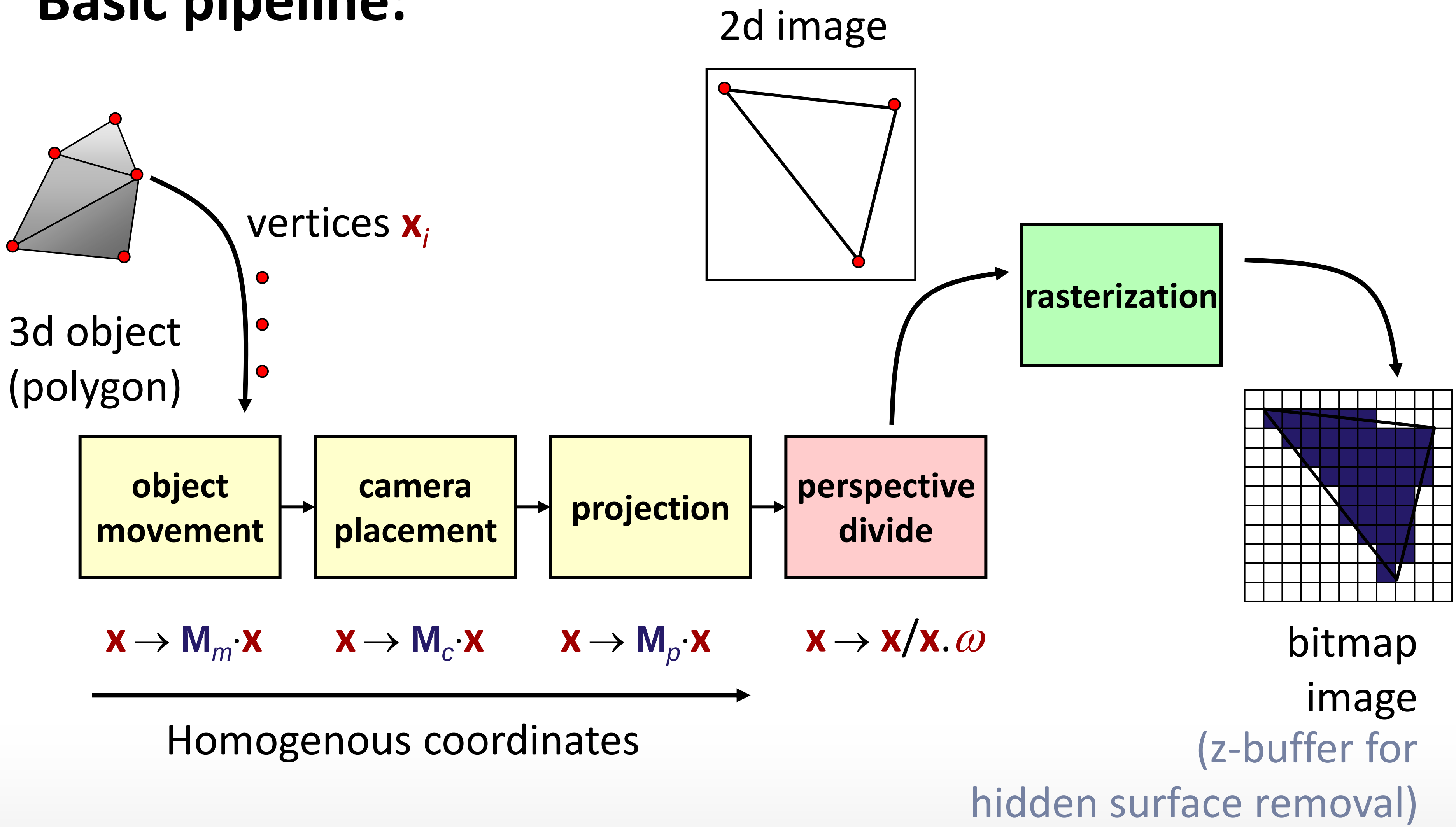
How can we render trimmed NURBS?

We will look at three variants:

- Rasterization
- Raytracing
- Hardware-friendly rasterization algorithm

Rasterization

Basic pipeline:



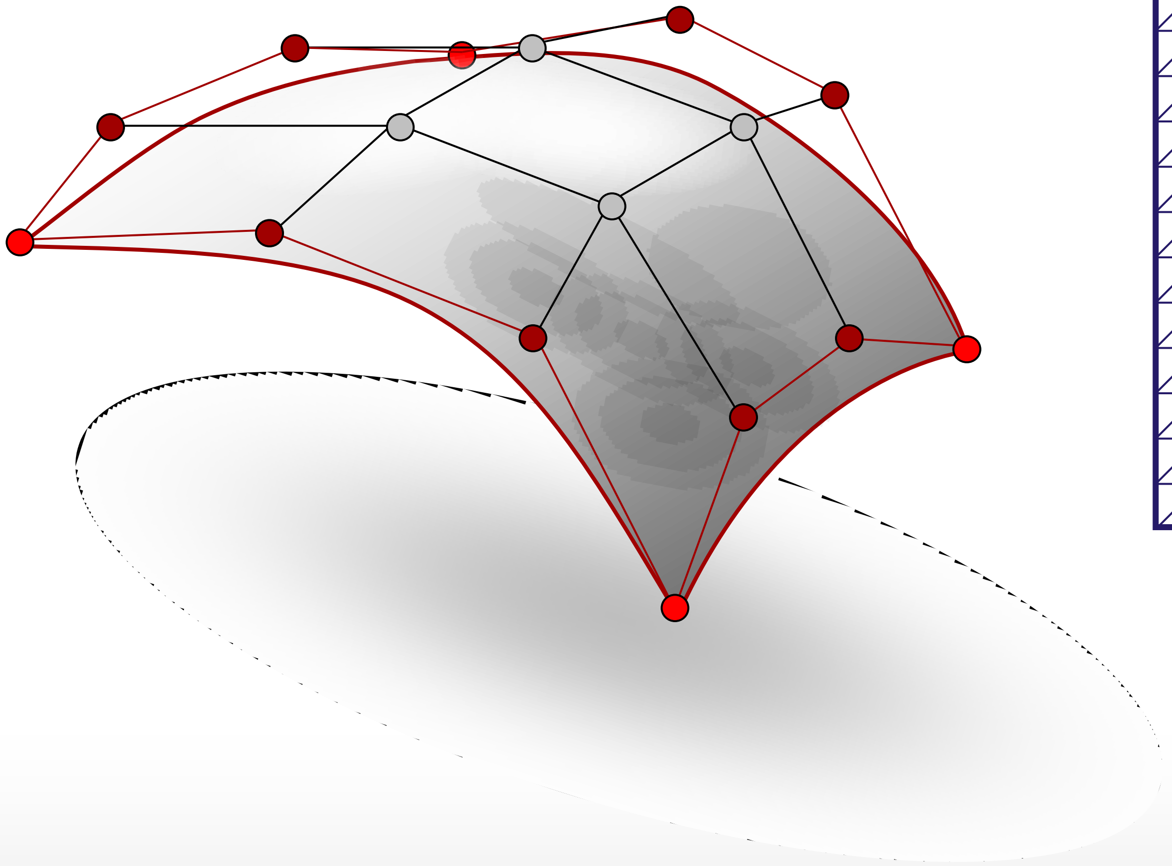
Rasterization Pipeline

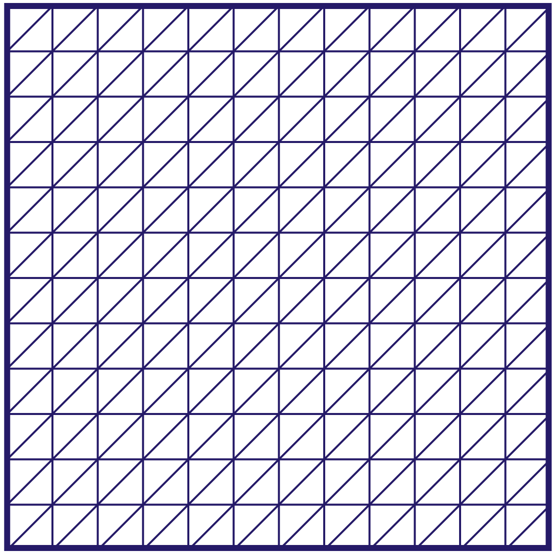
Basically:

- We can draw triangles
- Very efficient due to hardware support
  (standard GPU: 100 M triangles/sec, 1000 M pixels/sec)
- We need to convert our surfaces into triangles
  ("tessellation")
- Nowadays: We can afford high resolution tessellations

Simple Idea

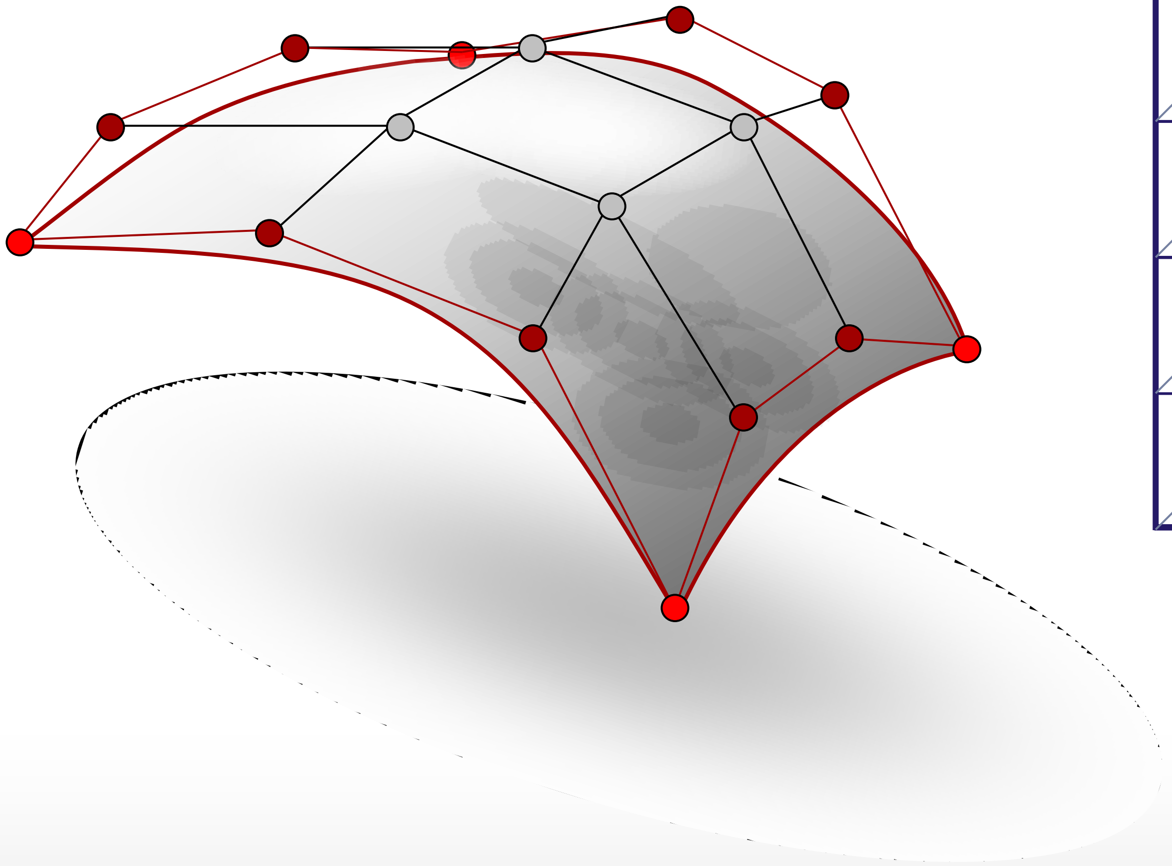
Simplest solution: Uniform tessellation

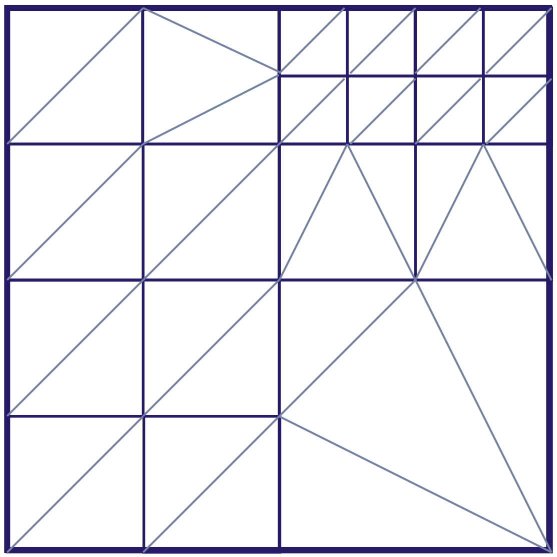




Fancier Idea

Better solution: Adaptive tessellation

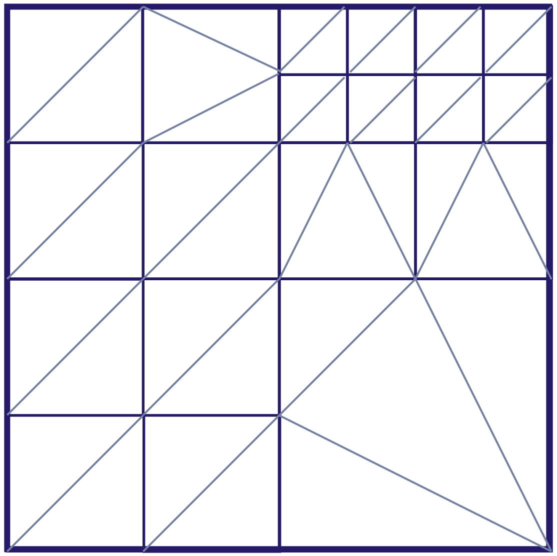




Adaptive Tesselation

Adaptive Tessellation:

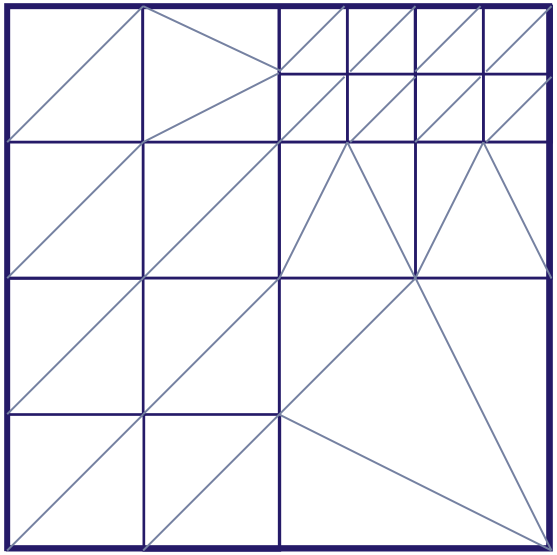
- Subdivide parameter domain recursively
- Divide rectangle into four smaller parts ("Quadtree")
- Possible stopping criterion:
  - Distance between planar faces and surface
  - Approximately: planarity of control points



Adaptive Tesselation

Adaptive Tessellation:

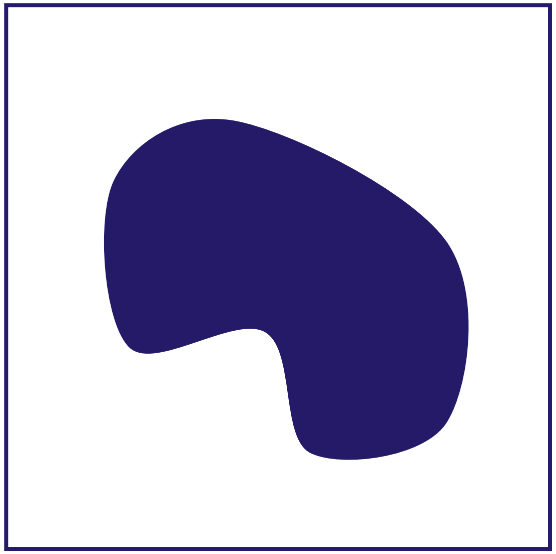
- Balanced Quadtree:
  - Make sure that the subdivision level of adjacent cells does not differ by more than one level
- Divide cells into triangles
- Look at direct neighbors to create a closed mesh
- Only $2^4 = 16$ cases



So what about the curves?

Remaining problem:

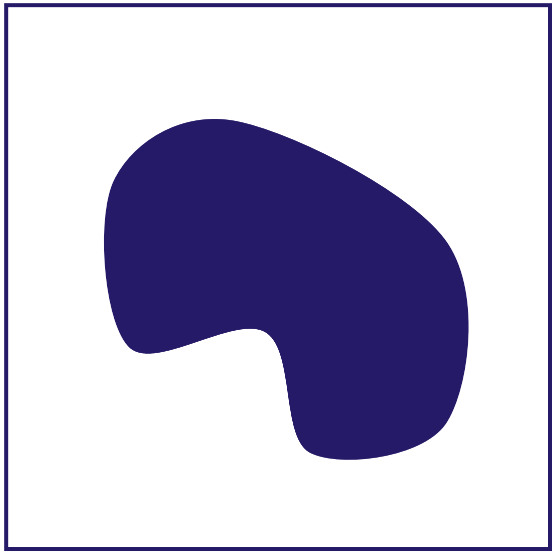
- Need to render trimmed patches
- Super-simple solution (“cheating”):
  - add a texture map, remove “white” pixels with (do not draw empty space)
  - Supported in hardware (“alpha test”)
  - But this looks ugly
  - And does not help in geometric computation (if we need a triangulation of the trimmed object for further processing)



So what about the curves?

Second try:

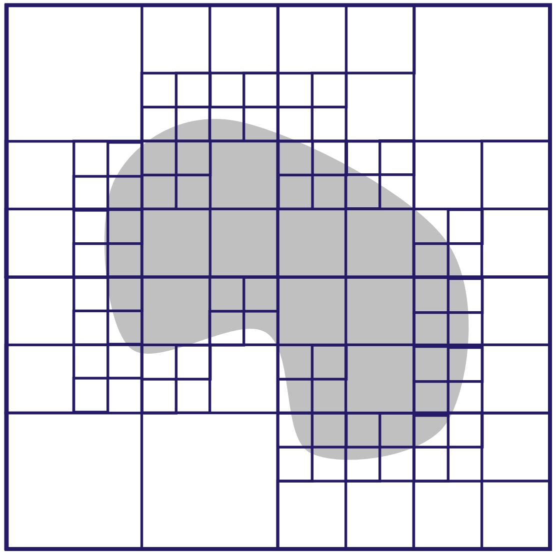
- We have to tessellate the trimming area in the domain
- Need to place triangles in the domain that approximate the shape
- Curve tessellation problem
  - Classic computational geometry problem
  - Several solutions
  - E.g. constrained Delaunay triangulation
- Easy to implement: Quadtree triangulation method



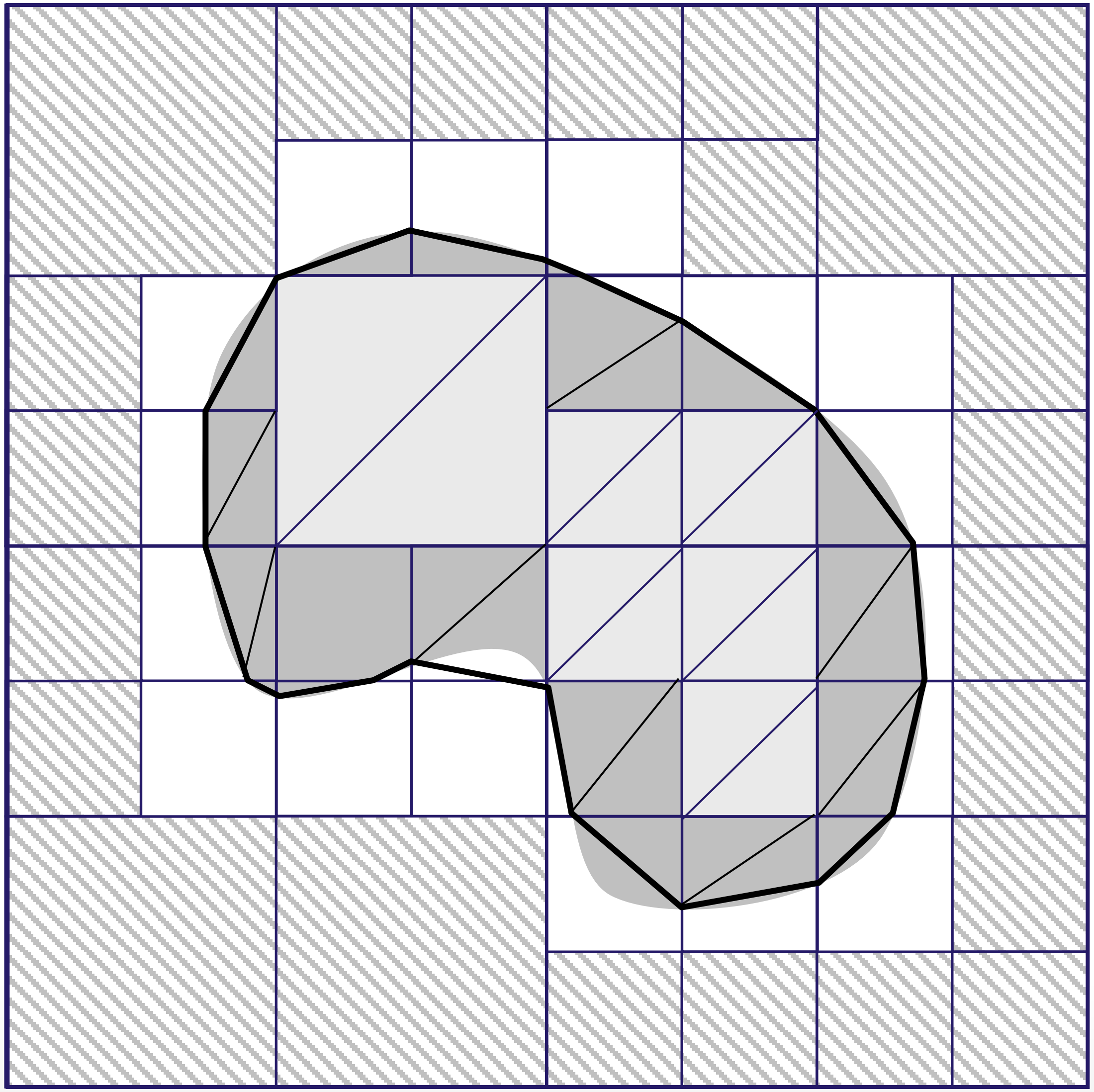
Quadtree triangulation

Quadtree triangulation:

- Subdivide recursively as before
- New stopping criterion
  - If the bounding box intersects the area:
    - Do not stop until surface is well approximated
    - **And**: No boundary curve inside, or the boundary curves intersects exactly twice
    - Limit recursion depth to avoid trouble at degeneracies
  - If the bounding box covers empty space:
    - Stop immediately



Quadtree triangulation



Quadtree triangulation

Tessellation Algorithm:

- Compute balanced quadtree
- Stop when accuracy is met and only two curve intersections are in each box
- Tesselate interior the same way as before
- Tessellate intersections with fixed scheme (at most two triangles)
- Drop exterior boxes

Interior holes:

- Use ray-based inside/outside test

Hardware friendly version

Problem:

- The adaptive tessellation is computationally costly
- Algorithm with complex data structures and pointers, not easy to implement on special purpose hardware
- Even a standard CPU needs its time

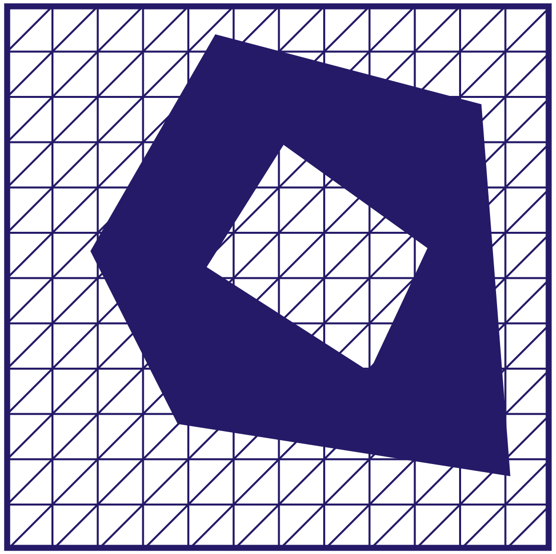
Hardware friendly algorithm: [Guthe et al. 2005]

- Basic idea: graphics hardware is so fast, we can waste a few triangles
- Runs completely on programmable graphics hardware
- We will discuss a simplified version (no gory GPU details)

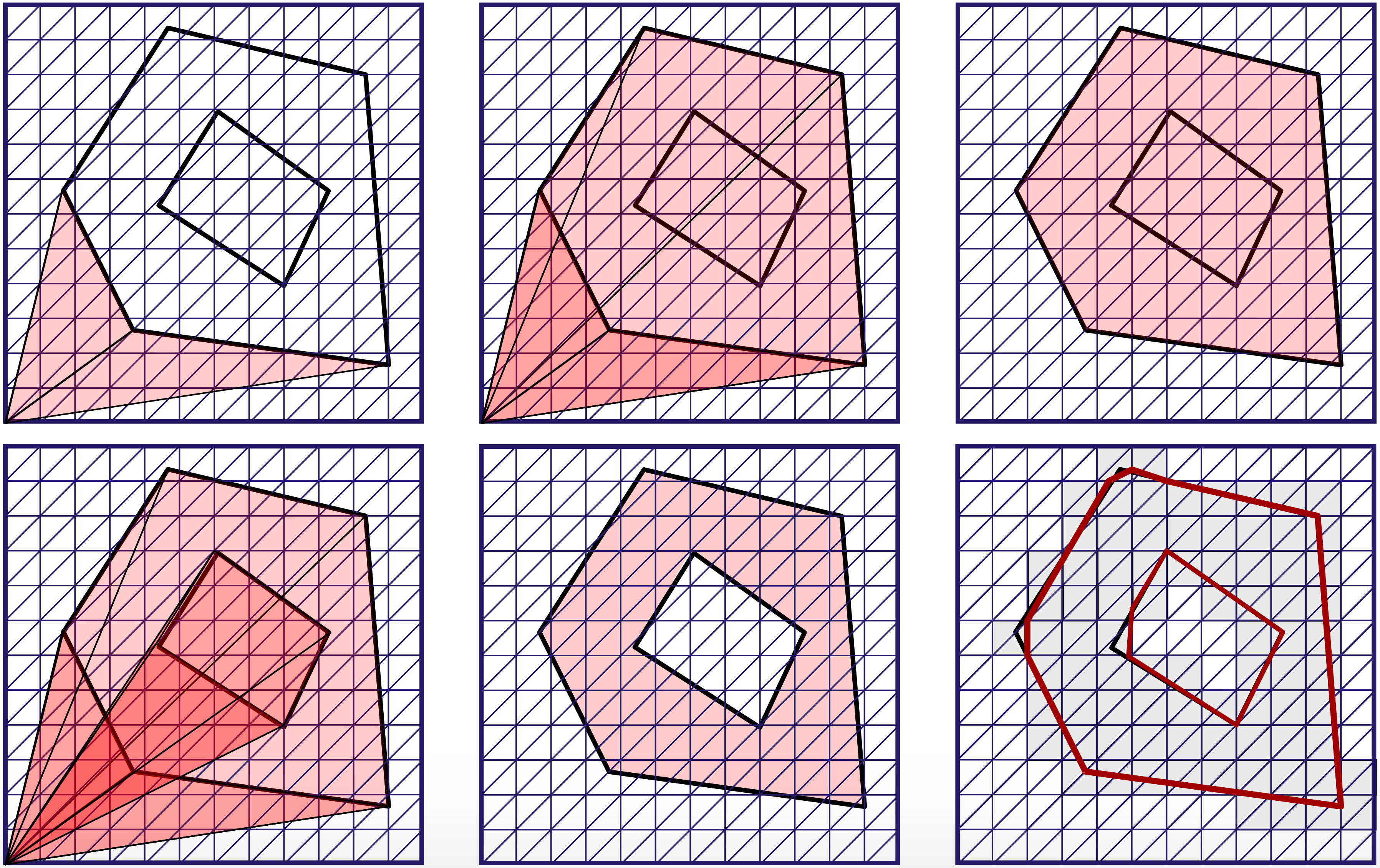
Guthe's Algorithm

Basic Idea:

- Use a uniform grid
- Represent each quad as a pixel
- Now render sequence of triangles along the curve, connected with one corner, in XOR mode



Guthe's Algorithm



Hardware friendly algorithm

After XOR-polygon drawing:

- Knowing the pixels that cover the domain, each one can be easily tessellated
- The spline surface is evaluated on the graphics hardware (programmable shaders)
- This algorithm is much faster than standard techniques
- In case the accuracy is not sufficient, a hierarchical refinement “on demand” is implemented
- Increases the resolution in surface parts close to the viewer

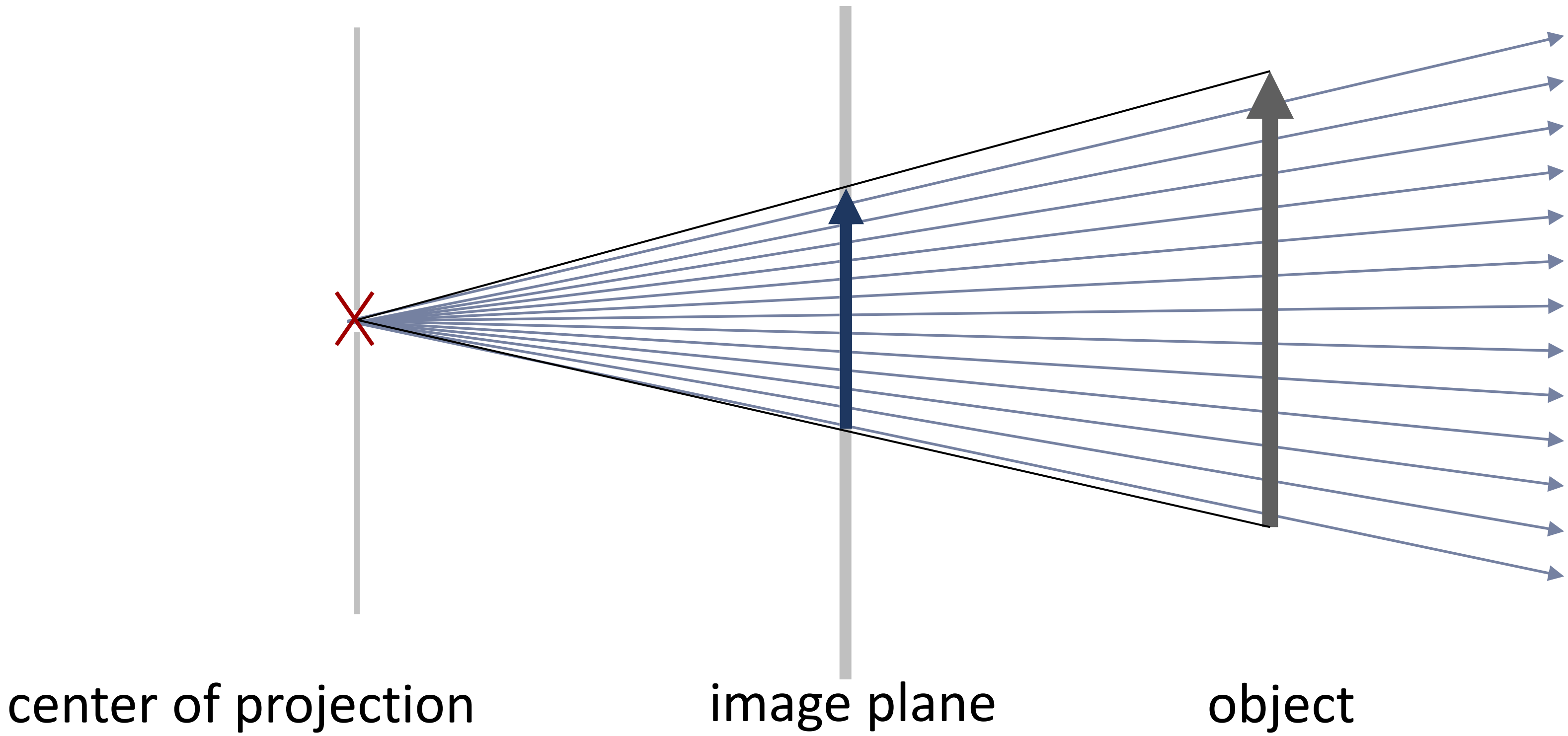
Raytracing

How can we raytrace NURBS patches?

Raytracing algorithm:

- Shoot a ray through each pixel of the image
- Test objects in the scene for intersection
- Display closest object
- For shading the object, further rays can be sent recursively
  - Shadow rays to the light source(s) – if blocked, object is in shadow
  - Reflected / refracted rays for mirroring / refractions

Raytracing



Intersection Problem

Intersection Problem

- Rendering with raytracing reduces to determine whether a ray intersects a spline patch
- Non-linear system of equations:

$$\left.\begin{aligned} \mathbf{f}(u,v) &= \sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u)B_j^{(d)}(v)\mathbf{p}_{i,j} \\ \mathbf{r}(t) &= t\mathbf{a}+\mathbf{b} \end{aligned}\right\} \underbrace{\sum_{i=0}^{d}\sum_{j=0}^{d} B_i^{(d)}(u)B_j^{(d)}(v)\mathbf{p}_{i,j} - t\mathbf{a} + \mathbf{b}}_{\mathbf{F}(u,v,t)} = 0$$

$$\mathbf{F}(u,v,t) = 0$$

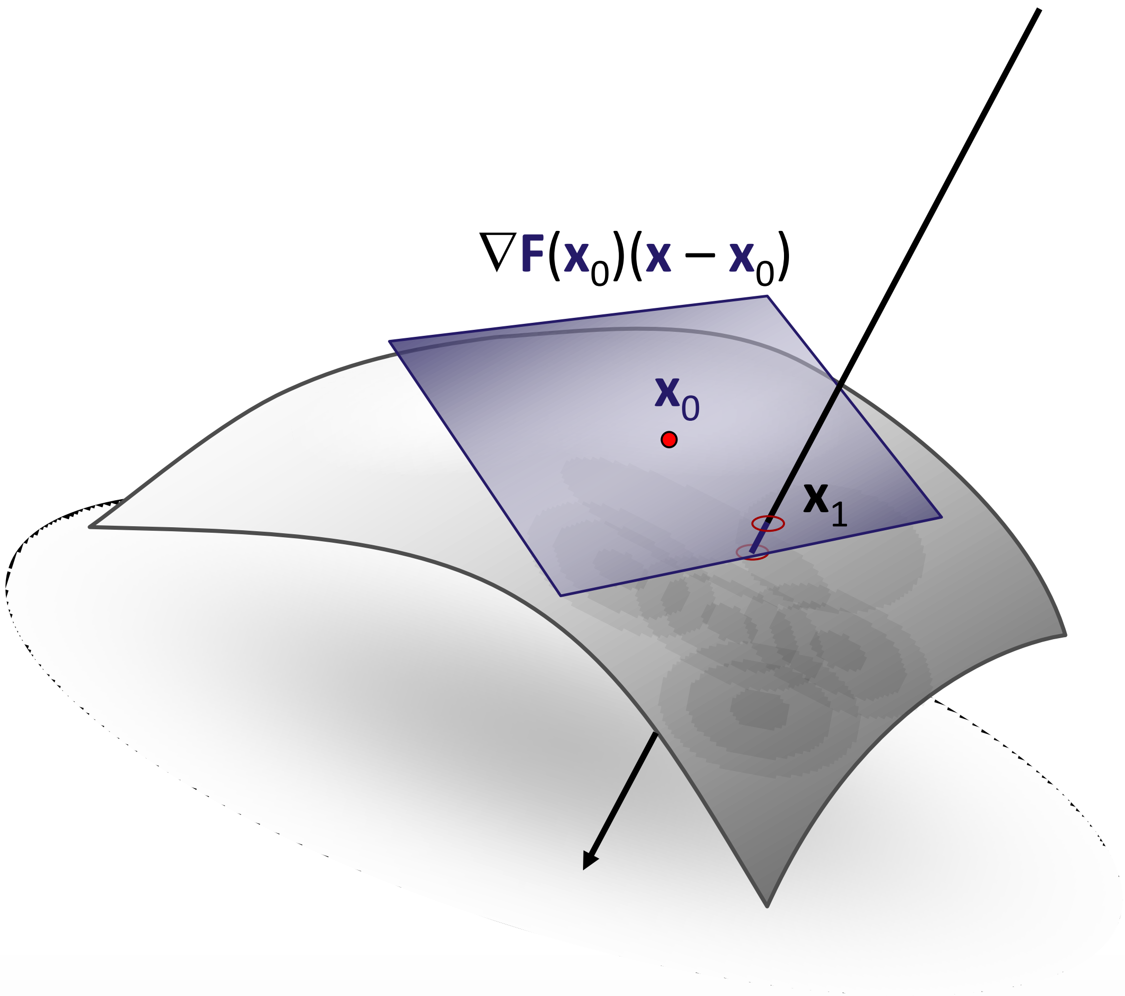
solve for *u*, *v*, *t*

Solution Strategies

Numerical optimization

- No closed form solution
- Therefore: Numerical approach
  - Need a starting value $\mathbf{x}_0$ (e.g. $\mathbf{x}_0 = (u,v,t) = (0,0,0)$)
  - Then iteratively improve solution
- Numerical techniques
  - (Gradient decent on squared residue)
  - Newton's method: Linearize problem
    - Compute Jacobian
    - Solve linear system $\nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \mathbf{F}(\mathbf{x}_0) = 0$
    - Iterate
  - Newton-like geometric technique

Newton-like technique



Problem

Properties of Newton-based algorithm

- Quite efficient – typically needs only a few iterations
- However: No convergence guarantees
  - In general: does not always converge to the correct solution
- Need good initialization

Brute-Force approach:

- Restart iteration from a number of starting points on the surface
- But that takes forever to compute

Alternative

Alternative: Hierarchical subdivision algorithm

- Compute bounding volume of control points (convex hull property)
  - We can use the convex hull
  - Simpler to implement: bounding sphere
- Test for intersection
  - No intersection found $\rightarrow$ return false, we are done
  - Otherwise continue recursively
- Recursion: subdivide patch into four parts (de Casteljau)
- Call recursive test for all patches
- Always terminate, if precision is sufficient

Alternative

Alternative: Hierarchical subdivision algorithm

- Guaranteed to converge
- But slower
  - Linear convergence, i.e. number of correct digits in solution increases proportional to #iterations (asymptotical)
  - Newton method typically converges quadratically (number of correct digits increases quadratically)

“Best of both worlds”

- Start with a few iterations of hierarchical subdivision
  - Stopping criterion: Test for “flatness of control points”
- Then use Newton iteration to boost accuracy rapidly