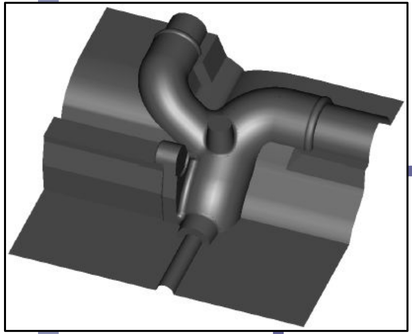
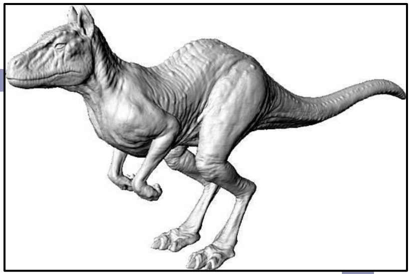




Further Graphics

Splines
Delaunay Triangulations

Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

Drawing a Bezier cubic: Iterative method

Fixed-step iteration:

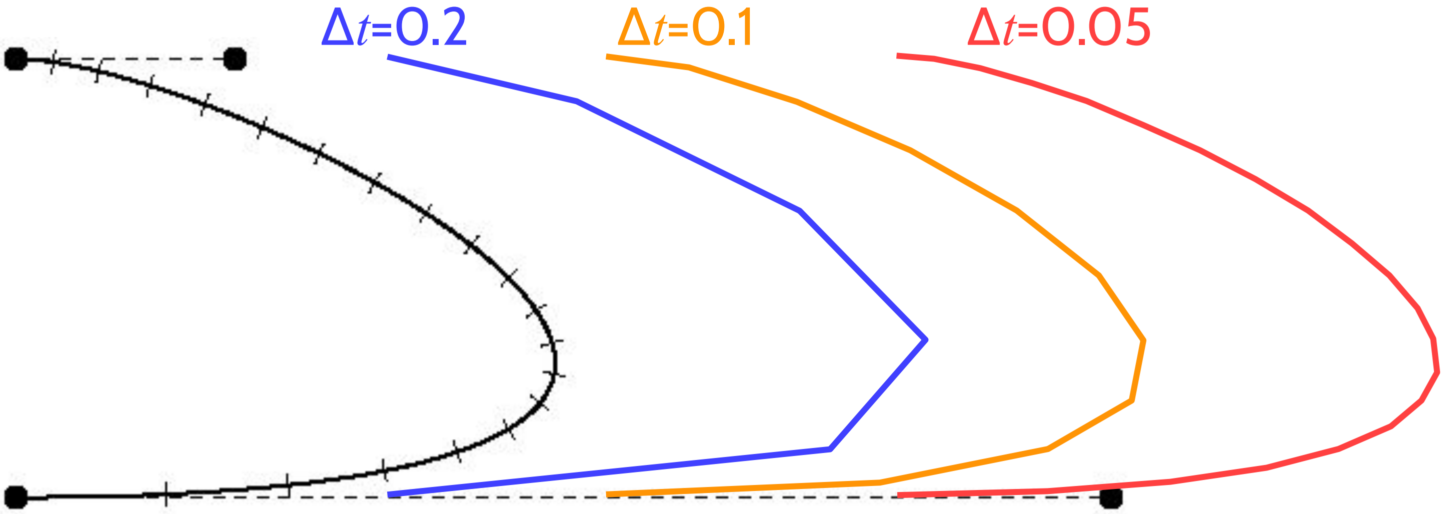
- Draw as a set of short line segments equispaced in parameter space, *t*:

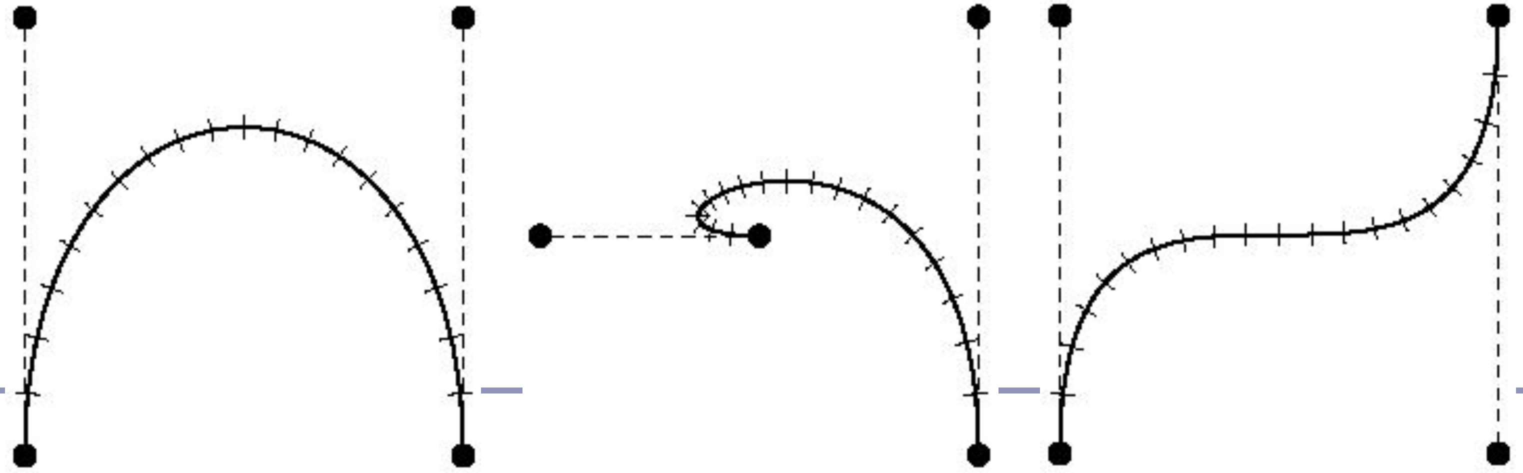
```
(x0,y0) = Bezier(0)
FOR t = 0.05 TO 1 STEP 0.05 DO
    (x1,y1) = Bezier(t)
    DrawLine( (x0,y0), (x1,y1) )
    (x0,y0) = (x1,y1)
END FOR
```

- Problems:
  - Cannot fix a number of segments that is appropriate for all possible Beziers: too many or too few segments
  - distance in real space, *(x,y)*, is not linearly related to distance in parameter space, *t*

Drawing a Bezier cubic

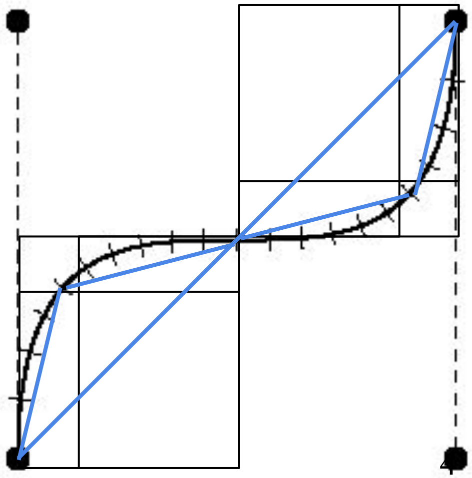
...but not very well





Drawing a Bezier cubic: Adaptive method

- Subdivision:
  - check if a straight line between P_0 and P_3 is an adequate approximation to the Bezier
  - if so: draw the straight line
  - if not: divide the Bezier into two halves, each a Bezier, and repeat for the two new Beziers
- Need to specify some tolerance for when a straight line is an adequate approximation
  - when the Bezier lies within half a pixel width of the straight line along its entire length



Drawing a Bezier cubic: Adaptive method

```
Procedure DrawCurve( Bezier curve )
VAR Bezier left, right
BEGIN DrawCurve
  IF Flat(curve) THEN
     DrawLine(curve)
  ELSE
     SubdivideCurve(curve, left, right)
     DrawCurve(left)
     DrawCurve(right)
  END IF
END DrawCurve
```

e.g. if P_1 and P_2 both lie within half a pixel width of the line joining P_0 to P_3, then...

...draw a line from P_0 to P_3; otherwise,

...split the curve into two Beziers covering the first and second halves of the original and draw recursively

Checking for flatness

$$P(t) = (1-t)\,A + t\,B$$

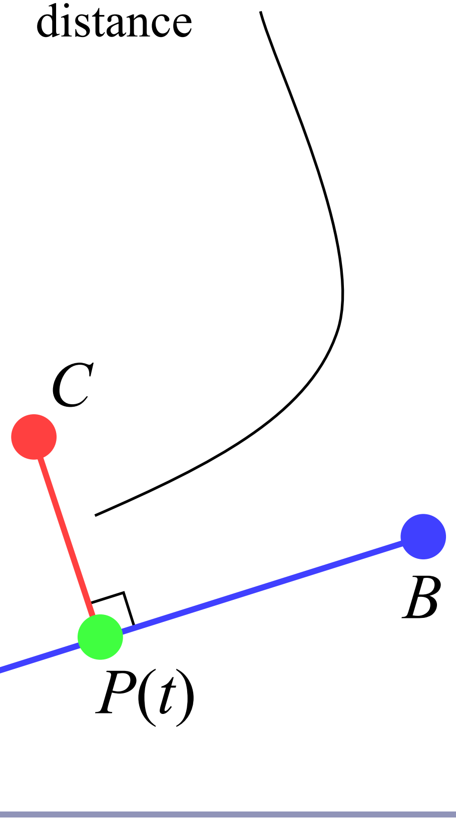
$$AB \cdot CP(t) = 0$$

$$\rightarrow (x_B - x_A)(x_P - x_C) + (y_B - y_A)(y_P - y_C) = 0$$

$$\rightarrow t = \frac{(x_B - x_A)(x_C - x_A) + (y_B - y_A)(y_C - y_A)}{(x_B - x_A)^2 + (y_B - y_A)^2}$$

$$\rightarrow t = \frac{AB \cdot AC}{|AB|^2}$$

Careful! If $t < 0$ or $t > 1$,
use $|AC|$ or $|BC|$ respectively.



we need to know this distance

Subdividing a Bezier cubic in two

To split a Bezier cubic into two smaller Bezier cubics:

$Q_0 = P_0$

$Q_1 = \tfrac{1}{2} P_0 + \tfrac{1}{2} P_1$

$Q_2 = \tfrac{1}{4} P_0 + \tfrac{1}{2} P_1 + \tfrac{1}{4} P_2$

$Q_3 = \tfrac{1}{8} P_0 + \tfrac{3}{8} P_1 + \tfrac{3}{8} P_2 + \tfrac{1}{8} P_3$

$R_3 = \tfrac{1}{8} P_0 + \tfrac{3}{8} P_1 + \tfrac{3}{8} P_2 + \tfrac{1}{8} P_3$

$R_2 = \tfrac{1}{4} P_1 + \tfrac{1}{2} P_2 + \tfrac{1}{4} P_3$

$R_1 = \tfrac{1}{2} P_2 + \tfrac{1}{2} P_3$

$R_0 = P_3$

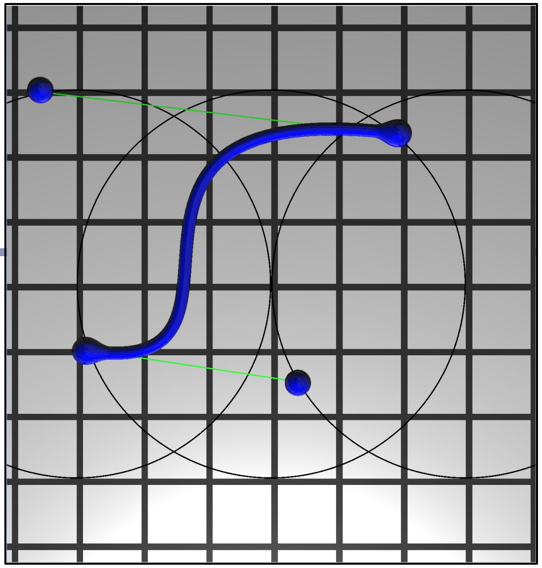
These cubics will lie atop the halves of their parent exactly, so rendering them = rendering the parent.

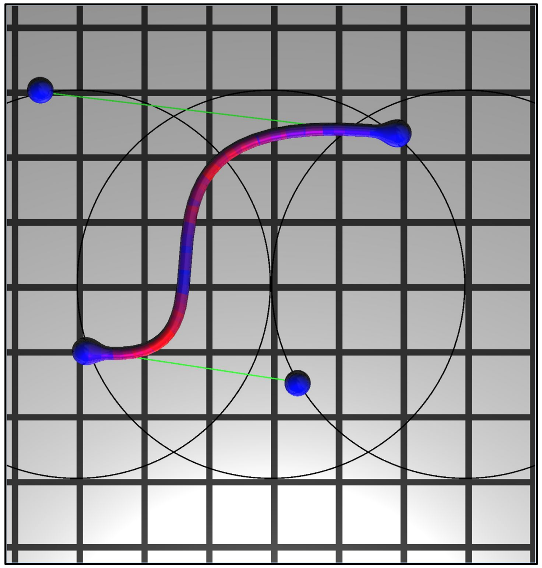
Overhauser's cubic

Overhauser's cubic: a Bezier cubic which passes through four target data points

- Calculate the appropriate Bezier control point locations from the given data points
  - e.g. given points A, B, C, D, the Bezier control points are:
  - P0 = B P1 = B + (C-A)/6
  - P3 = C P2 = C - (D-B)/6
- Overhauser's cubic *interpolates* its controlling points
  - good for animation, movies; less for CAD/CAM
  - moving a single point modifies four adjacent curve segments
  - compare with Bezier, where moving a single point modifies just the two segments connected to that point

Drawing a Bezier cubic: Signed Distance Fields





1. Iterative implementation

   SDF(P) = *min*(distance from *P* to each of *n* line segments)

   - In the demo, 50 steps suffices

2. Adaptive implementation

   SDF(P) = *min*(distance to each sub-curve whose bounding box contains *P*)

   - Can fast-discard sub-curves whose bbox doesn't contain *P*
   - In the demo, 25 subdivisions suffices

Into the Third Dimension

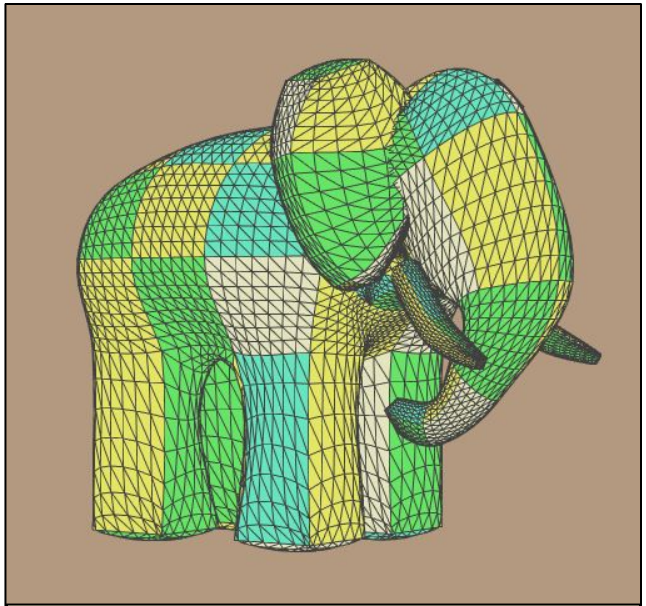
A Bezier *patch* can be defined by sixteen control points,

$$\begin{matrix} P_{0,0} & \dots & P_{0,3} \\ \vdots & & \vdots \\ P_{3,0} & \dots & P_{3,3} \end{matrix}$$

The weighted average of these 16 points uses Bernstein polynomials just like the 2D form:

$$P(s,t) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(s) b_j(t) P_{i,j}$$

$$b_{v,n}(t) = \binom{n}{v} t^v (1-t)^{n-v}$$



Ed Catmull's "Gumbo" model, composed from patches (source: https://en.wikipedia.org/wiki/Bézier_surface)

Tensor product $\otimes$

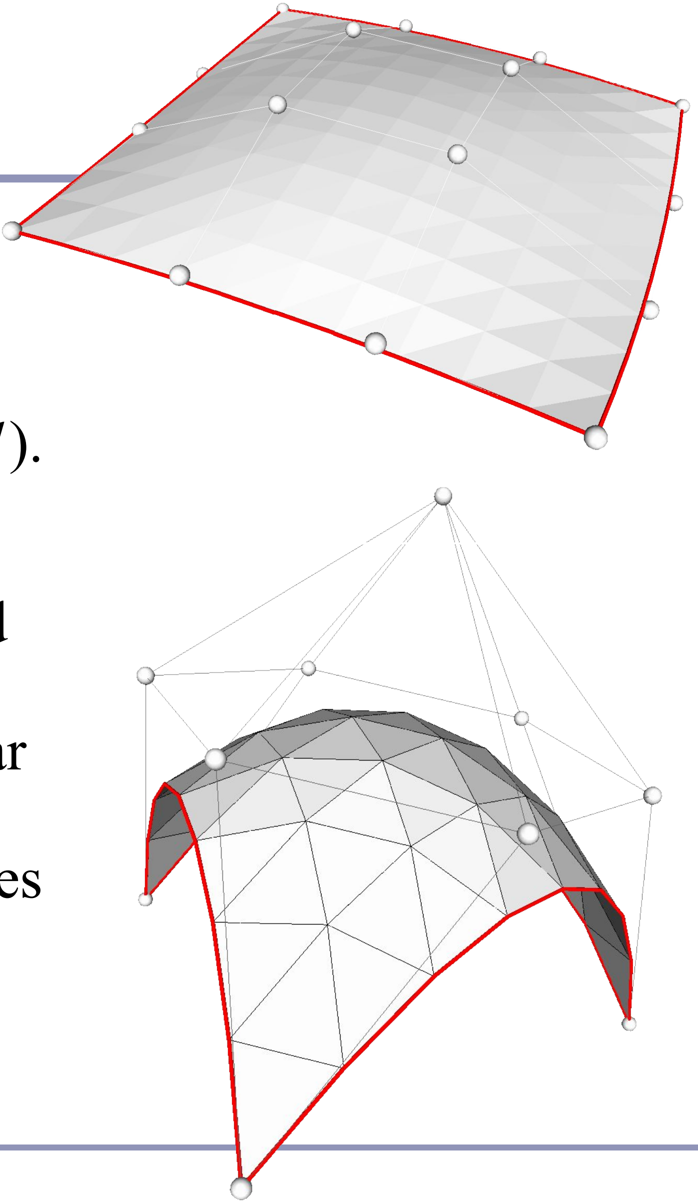
- The *tensor product* of two vectors is a matrix.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \otimes \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

- Can take the tensor of two polynomials
  - Each coefficient represents a piece of each of the two original expressions, so the cumulative polynomial represents both original polynomials completely

Bezier patches

- If curve A has *n* control points and curve B has *m* control points then $A \otimes B$ is an $(n) \times (m)$ matrix of polynomials of degree $max(n-1, m-1)$.
  - $\otimes$ = *tensor product*
- Multiply this matrix against an $(n) \times (m)$ matrix of control points and sum them all up and you've got a bivariate expression for a rectangular surface patch, in 3D
- This approach generalizes to triangles and arbitrary *n*-gons.



Continuity between Bezier patches

Ensuring continuity in 3D:

- C0 – continuous in position
  - the four edge control points must match
- C1 – continuous in position and tangent vector
  - the four edge control points must match
  - the two control points on either side of each of the four edge control points must be co-linear with both the edge point, and each other, *and* be equidistant from the edge point
- G1 – continuous in position and tangent direction the four edge control points must match the relevant control points must be co-linear

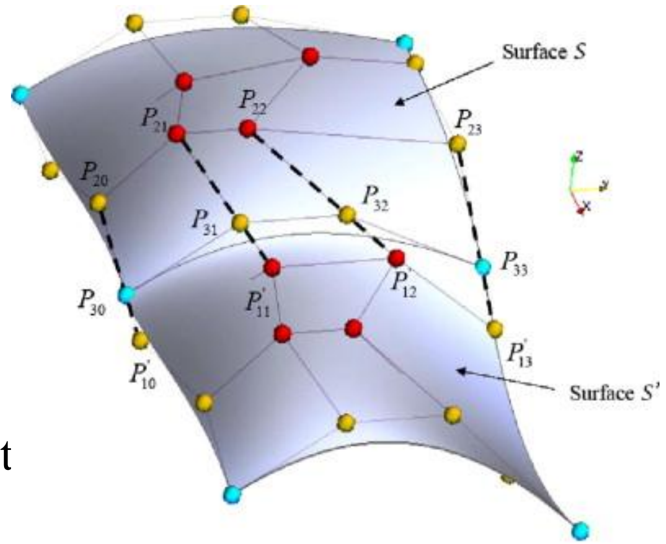
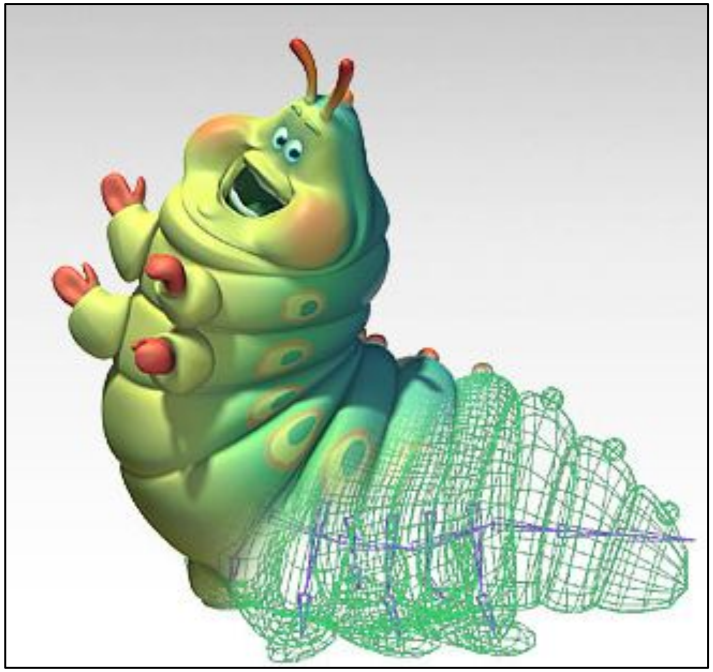


Image credit: Olivier Czarny, Guido Huysmans. *Bézier surfaces and finite elements for MHD simulations*. Journal of Computational Physics Volume 227, Issue 16, 10 August 2008

NURBS in 3D

Like Bezier patches, NURBS surfaces are the bivariate generalisation of the univariate NURBS form:

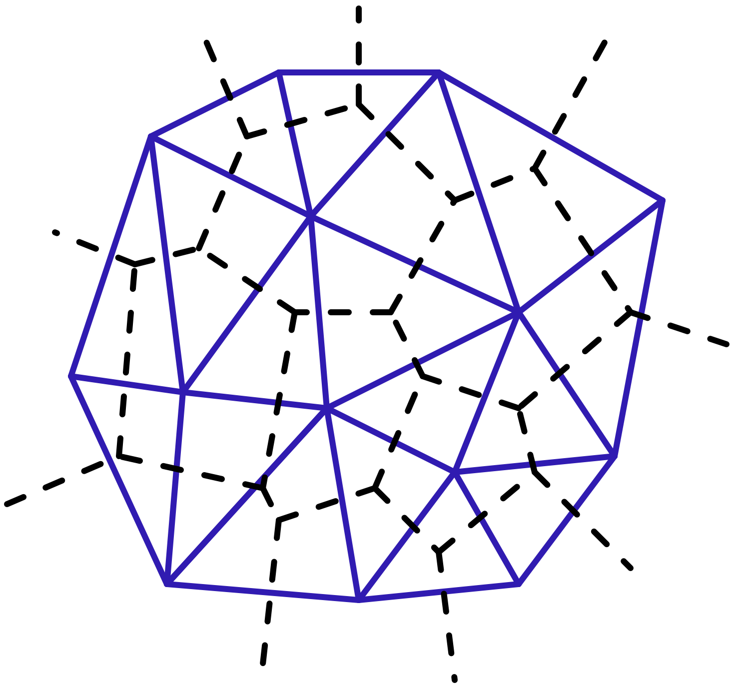


$$P(t) = \sum_{i=1}^{n} N_{i,k}(t) P_i$$

$$P(s,t) = \sum_{i=1}^{m} \sum_{j=1}^{n} N_{i,k}(s) N_{j,k}(t) P_{i,j}$$

Voronoi diagrams

The *Voronoi diagram*(2) of a set of points P_i divides space into ‘cells’, where each cell C_i contains the points in space closer to P_i than any other P_j. The *Delaunay triangulation* is the dual of the Voronoi diagram: a graph in which an edge connects every P_i which share a common edge in the Voronoi diagram.



A Voronoi diagram (dotted lines) and its dual Delaunay triangulation (solid).

(2) AKA “Voronoi tesselation”, “Dirichelet domain”, “Thiessen polygons”, “plesiohedra”, “fundamental areas”, “domain of action”…

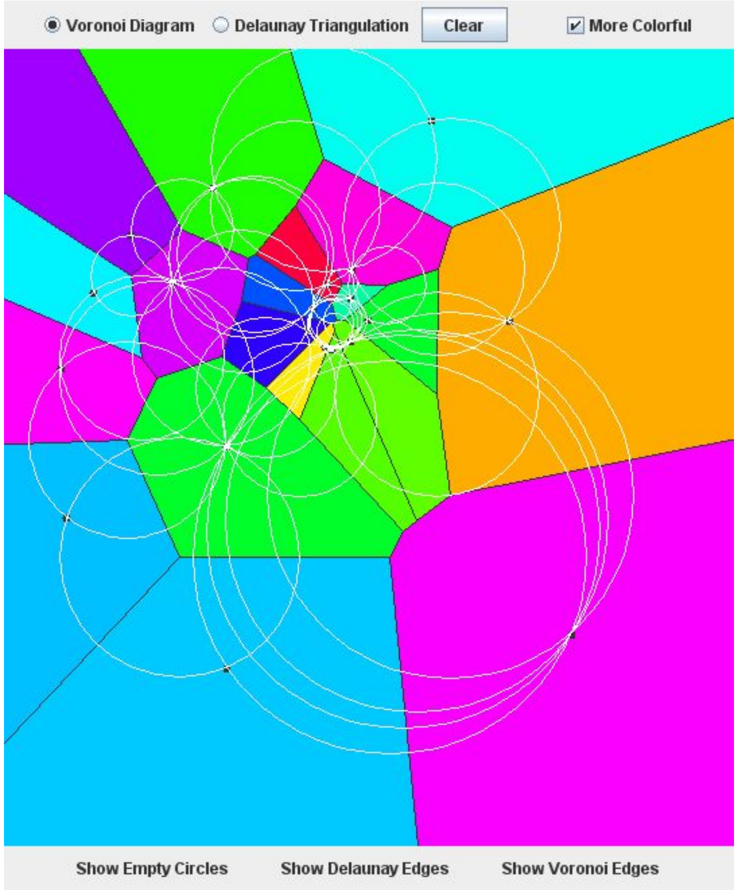
Voronoi diagrams

Given a set $S=\{p_1,p_2,\ldots,p_n\}$, the formal definition of a Voronoi cell $C(S,p_i)$ is

$C(S,p_i)=\{p \in R^d \mid |p-p_i|<|p-p_j|, i\neq j\}$

The p_i are called the *generating points* of the diagram.

Where three or more boundary edges meet is a *Voronoi point*. Each Voronoi point is at the center of a circle (or sphere, or hypersphere…) which passes through the associated generating points and which is guaranteed to be empty of all other generating points.



Delaunay triangulation applet by Paul Chew ©1997—2007
http://www.cs.cornell.edu/home/chew/Delaunay.html

Delaunay triangulations and *equi-angularity*

The *equiangularity* of any triangulation of a set of points *S* is an ascended sorted list of the angles ($\alpha_1 \ldots \alpha_{3t}$) of the triangles.

- A triangulation is said to be *equiangular* if it possesses lexicographically largest equiangularity amongst all possible triangulations of *S*.
- The Delaunay triangulation is <u>equiangular</u>.

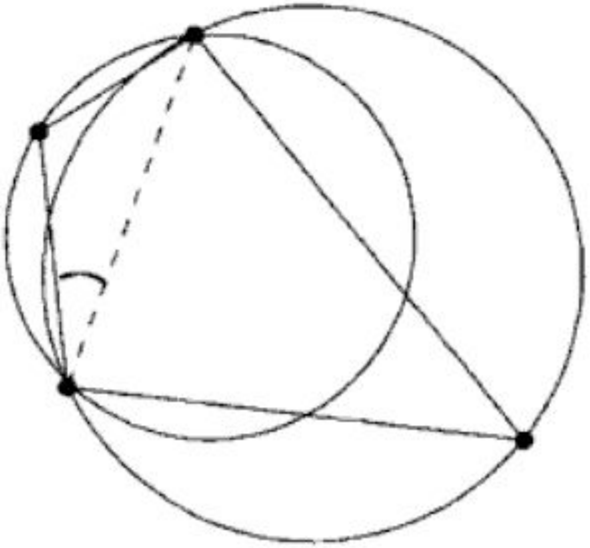


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

Delaunay triangulations and *empty circles*

Voronoi triangulations have the *empty circle* property*:* in any Voronoi triangulation of *S*, no point of *S* will lie inside the circle circumscribing any three points sharing a triangle in the Voronoi diagram.

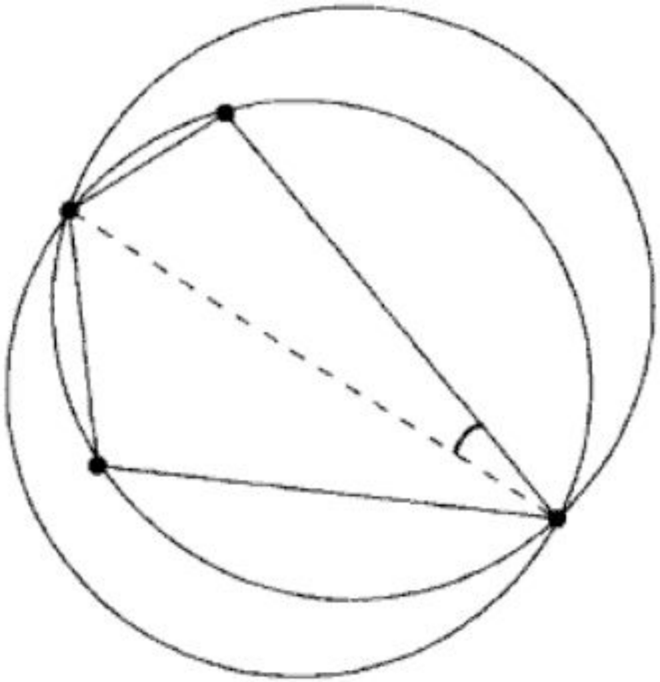


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

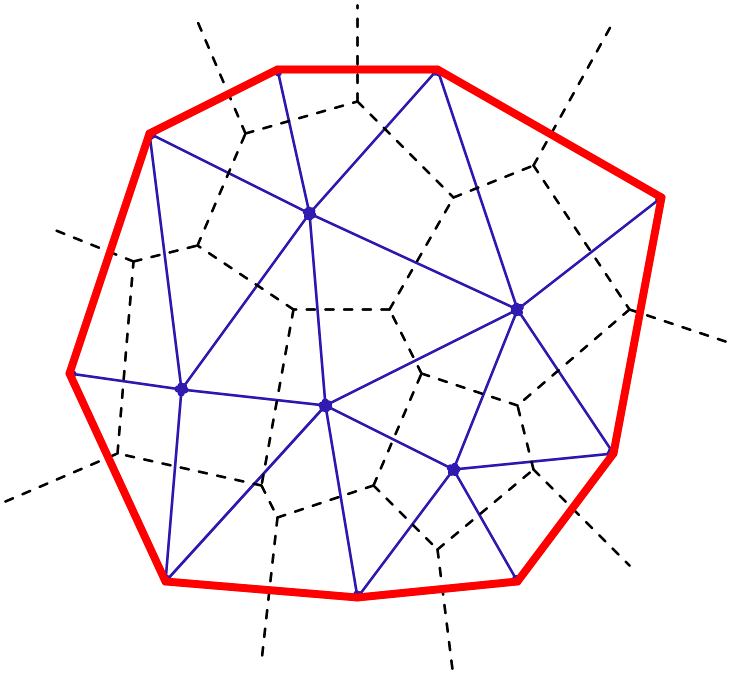
Delaunay triangulations and convex hulls

The border of the Delaunay triangulation of a set of points is always convex.

- This is true in 2D, 3D, 4D…

The Delaunay triangulation of a set of points in R^n is the planar projection of a convex hull in R^{n+1}.

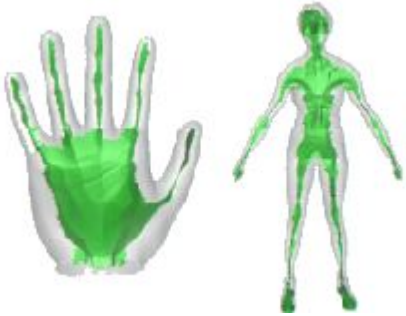
- Ex: from 2D ($P_i=\{x,y\}_i$), loft the points upwards, onto a parabola in 3D ($P'_i=\{x,y,x^2+y^2\}_i$). The resulting polyhedral mesh will still be convex in 3D.



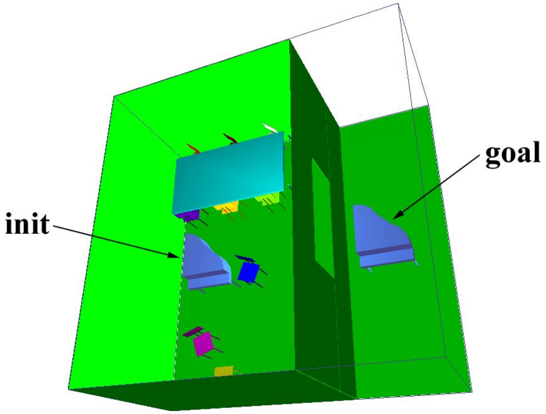
Voronoi diagrams and the *medial axis*

The *medial axis* of a surface is the set of all points within the surface equidistant to the two or more nearest points on the surface.

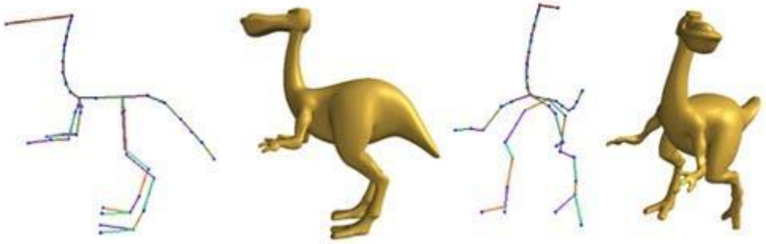
- This can be used to extract a skeleton of the surface, for (for example) path-planning solutions, surface deformation, and animation.



[Approximating the Medial Axis from the Voronoi Diagram with a Convergence Guarantee](#)
Tamal K. Dey, Wulue Zhao



[A Voronoi-Based Hybrid Motion Planner for Rigid Bodies](#)
M Foskey, M Garber, M Lin, DManocha

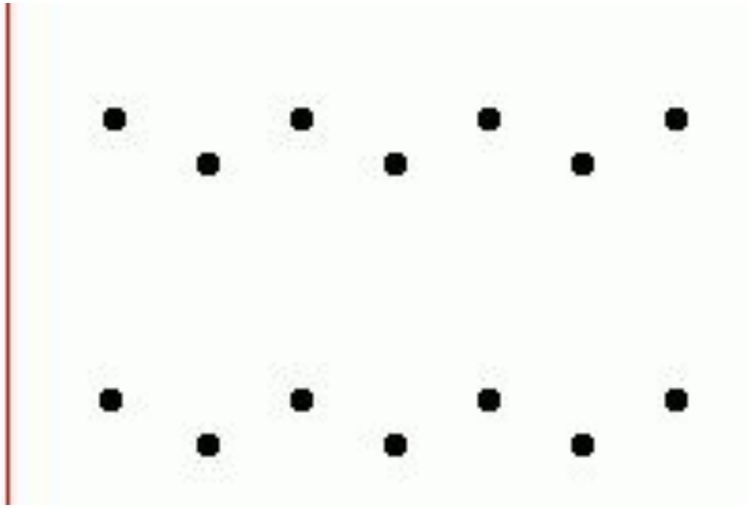


[Shape Deformation using a Skeleton to Drive Simplex Transformations](#)
IEEE Transaction on Visualization and Computer Graphics, Vol. 14, No. 3, May/June 2008, Page 693-706
Han-Bing Yan, Shi-Min Hu, Ralph R Martin, and Yong-Liang Yang

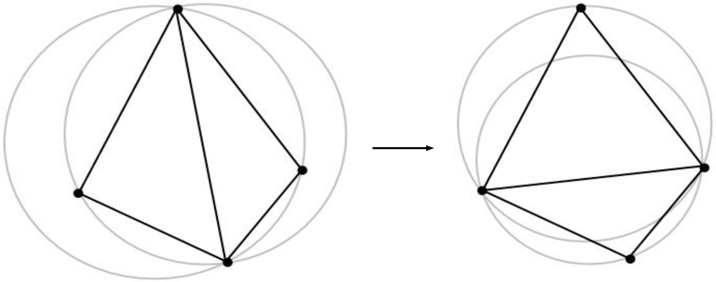
Finding the Voronoi diagram

There are four general classes of algorithm for computing the Delaunay triangulation:

- Divide-and-conquer
- Sweep plane
  - Ex: Fortune’s algorithm →
- Incremental insertion
- “Flipping”: repairing an existing triangulation until it becomes Delaunay



Fortune’s Algorithm for the plane-sweep construction of the Voronoi diagram (Steve Fortune, 1986.)



This triangulation fails the circumcircle definition; we flip its inner edge and it becomes Delaunay. *(Image from Wikipedia.)*

Fortune’s algorithm

1. The algorithm maintains a sweep line and a “beach line”, a set of parabolas advancing left-to-right from each point. The beach line is the union of these parabolas.
    a. The intersection of each pair of parabolas is an edge of the voronoi diagram
    b. All data to the left of the beach line is “known”; nothing to the right can change it
    c. The beach line is stored in a binary tree
2. Maintain a queue of two classes of event: the addition of, or removal of, a parabola
3. There are O(n) such events, so Fortune’s algorithm is O(n log n)

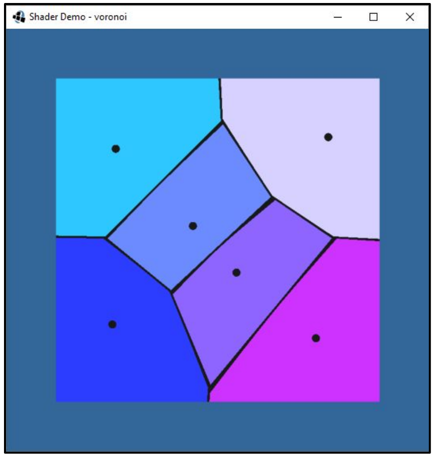
GPU-accelerated Voronoi Diagrams

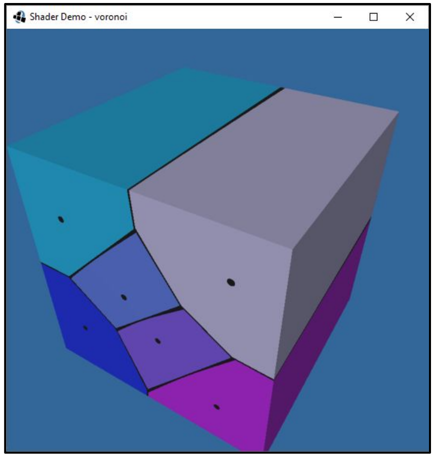
Brute force:

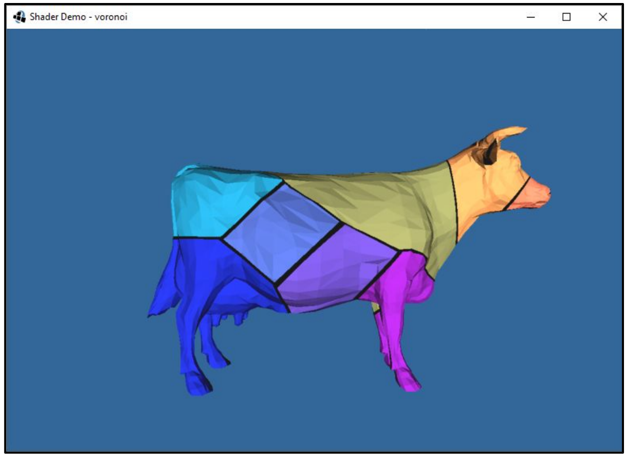
- For each pixel to be rendered on the GPU, search all points for the nearest point

Elegant (and 2D only):

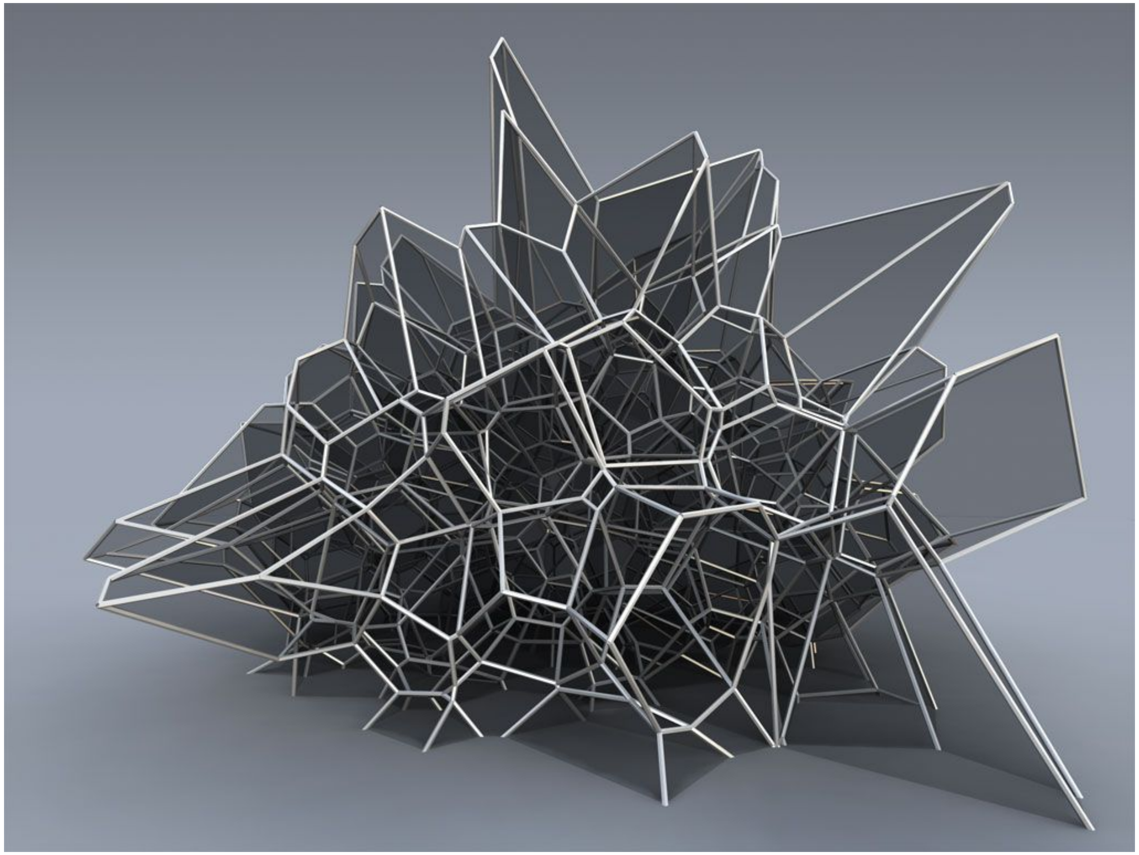
- Render each point as a discrete 3D cone in isometric projection, let z-buffering sort it out







Voronoi cells in 3D



Silvan Oesterle, Michael Knauss

References

Splines, continued

- Les Piegl and Wayne Tiller, *The NURBS Book*, Springer (1997)
- Alan Watt, *3D Computer Graphics*, Addison Wesley (2000)
- G. Farin, J. Hoschek, M.-S. Kim, *Handbook of Computer Aided Geometric Design*, North-Holland (2002)

Voronoi diagrams

- M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, "*Computational Geometry: Algorithms and Applications*", Springer-Verlag
- http://www.cs.uu.nl/geobook
- http://www.ics.uci.edu/~eppstein/junkyard/nn.html
- http://www.iquilezles.org/www/articles/voronoilines/voronoilines.htm