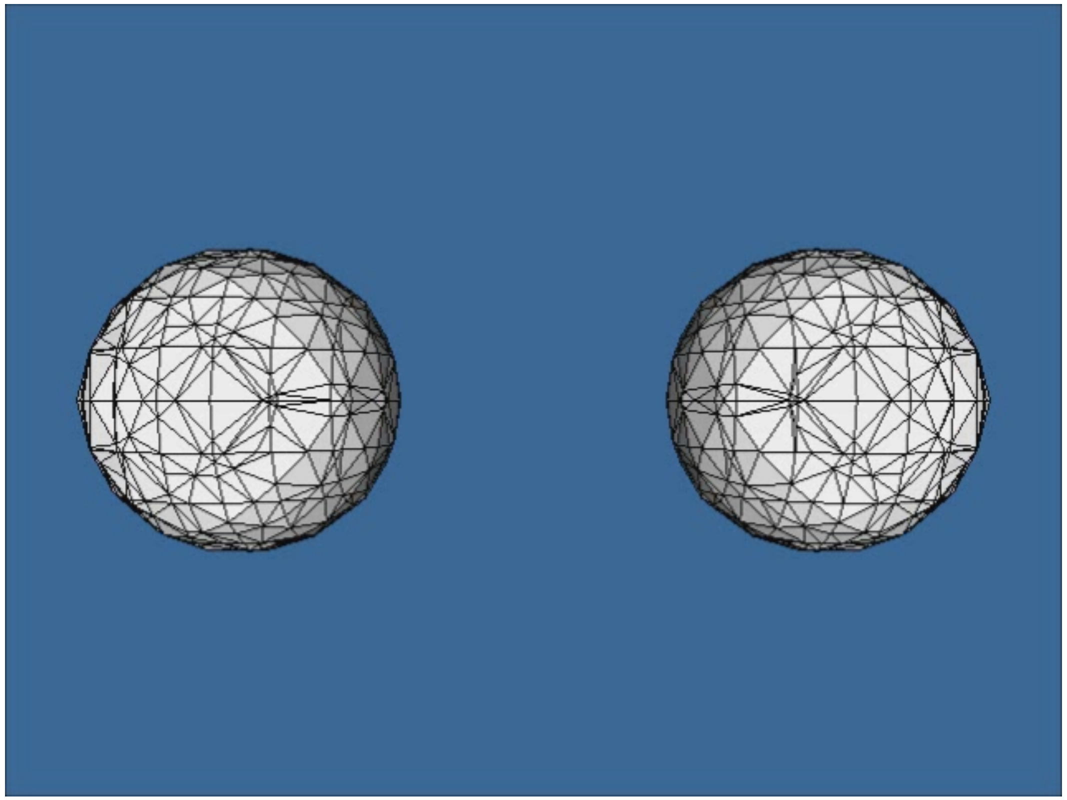
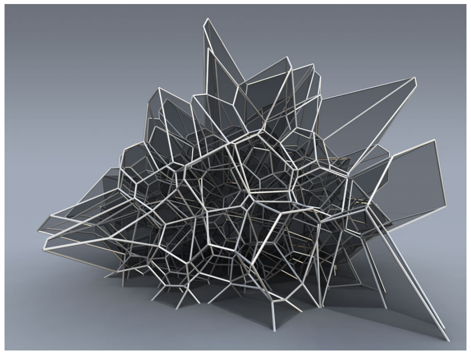
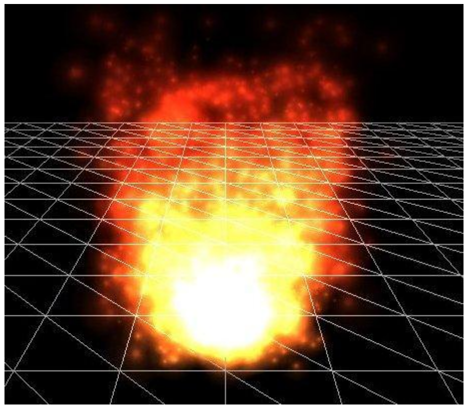
Advanced Graphics

Triangulation with Voronoi Diagrams and Implicit Surfaces; Voxels, Particle Systems, and Clever Rendering Tricks

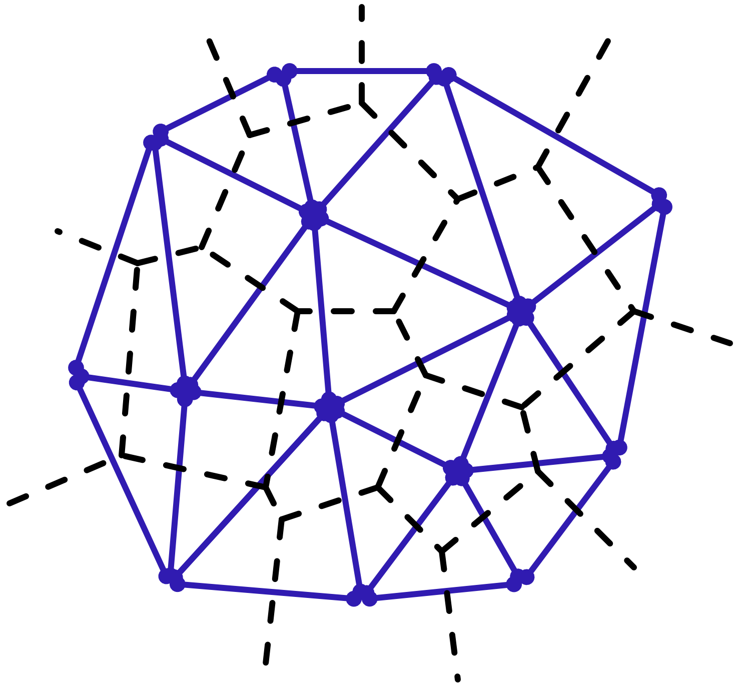
Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

Voronoi diagrams

The *Voronoi diagram*[1] 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).

[1] AKA "Voronoi tesselation", "Dirichelet domain", "Thiessen polygons", "plesiohedra", "fundamental areas", "domain of action"…

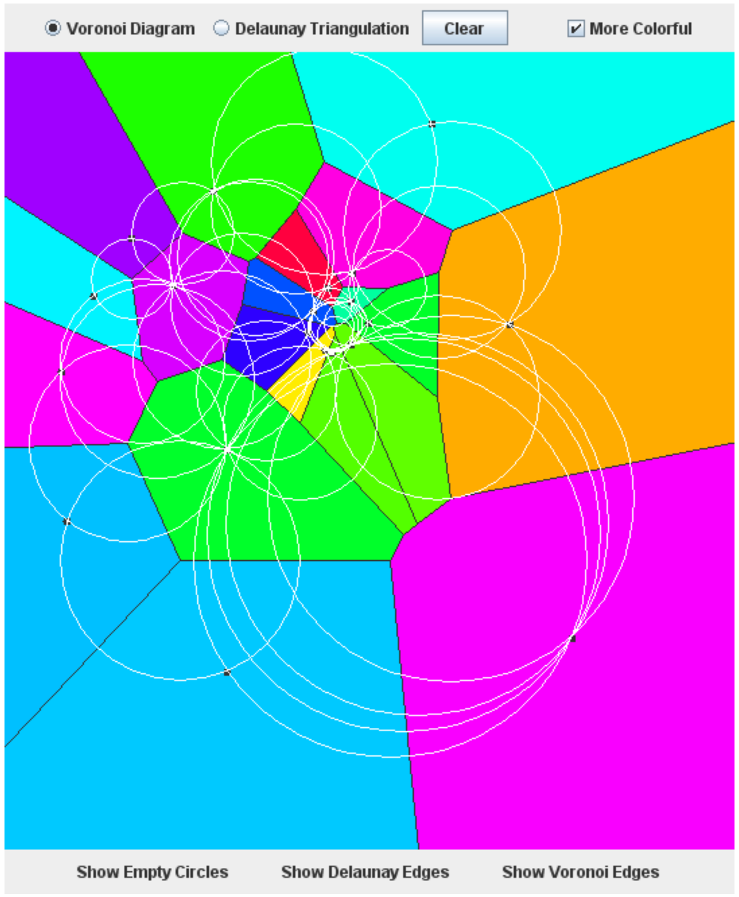
Voronoi diagrams

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 a 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 equiangular.

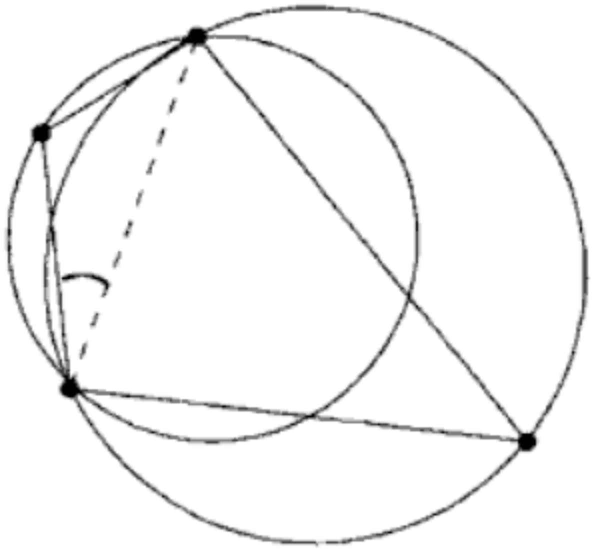


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

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.

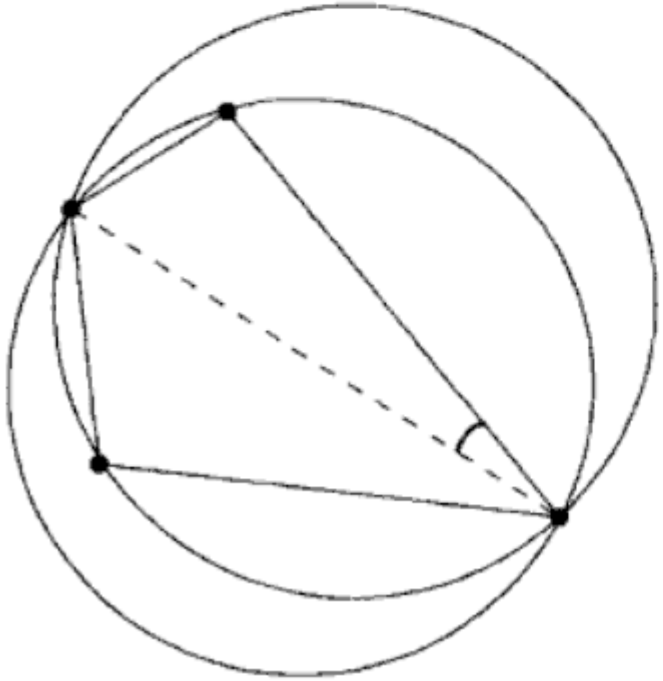


Image from *Handbook of Computational Geometry* (2000) Jörg-Rüdiger Sack and Jorge Urrutia, p. 227

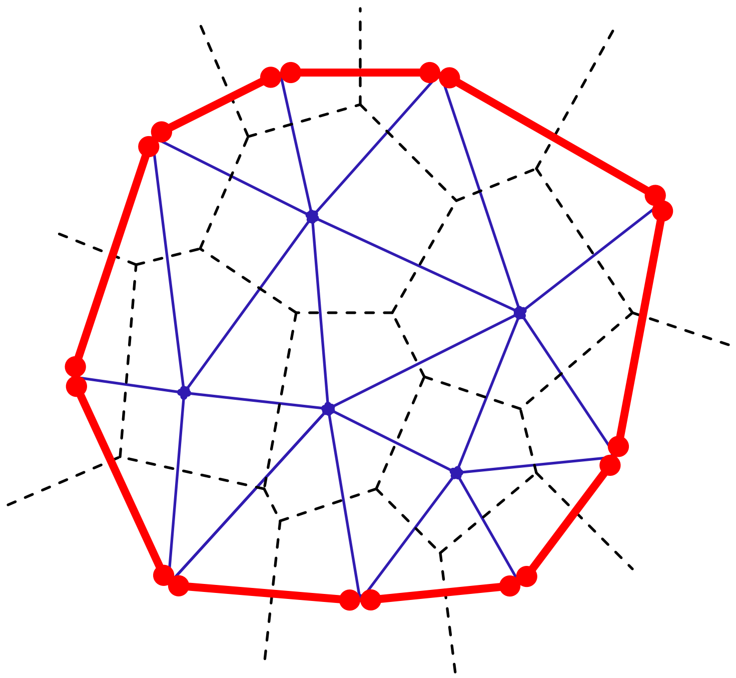
Delaunay triangulations and convex hulls

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}.

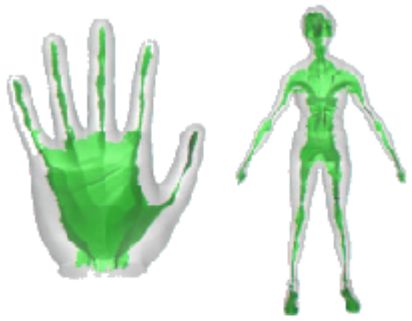
- Ex: from 2D ($P_i=\{x,y\}_i$), loft the points upwards, onto a parabola in 3D ($P'_i=\{x,y,x^2+y^2\}_i$). The resulting polyhedral mesh will still be convex in 3D.



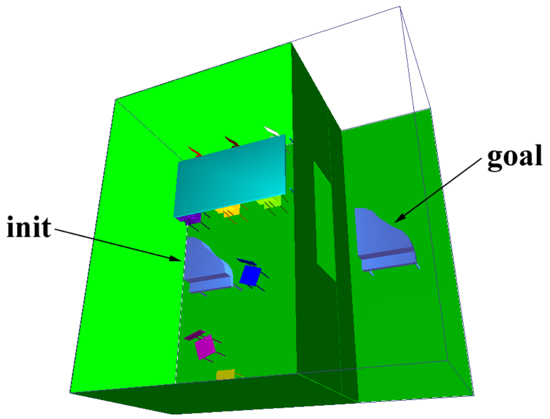
Voronoi diagrams and the *medial axis*

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.

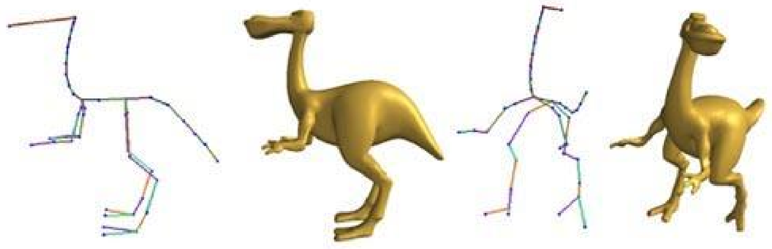
- This can be used to extract a skeleton of the surface, for (for example) path-planning solutions, surface deformation, and animation.



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

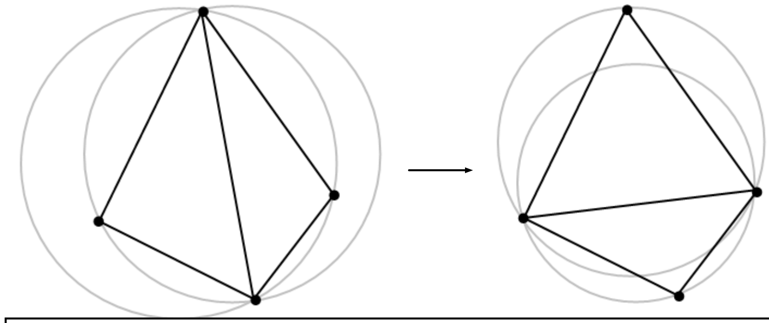


Shape Deformation using a Skeleton to Drive Simplex Transformations
IEEE Transaction on Visualization and Computer Graphics, Vol. 14, No. 3, May/June 2008, Page 693-706
Han-Bing Yan, Shi-Min Hu, Ralph R Martin, and Yong-Liang Yang

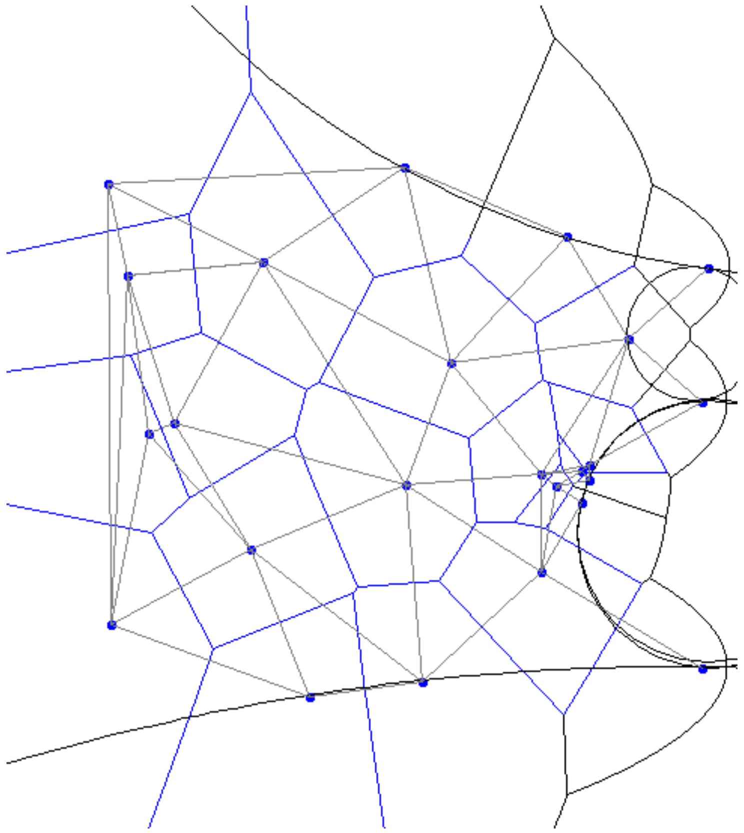
Finding the Voronoi diagram

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

- Divide-and-conquer
- Sweep plane
  - Fortune's algorithm →
- Incremental insertion
- "Flipping": repairing an existing triangulation until it becomes Delaunay

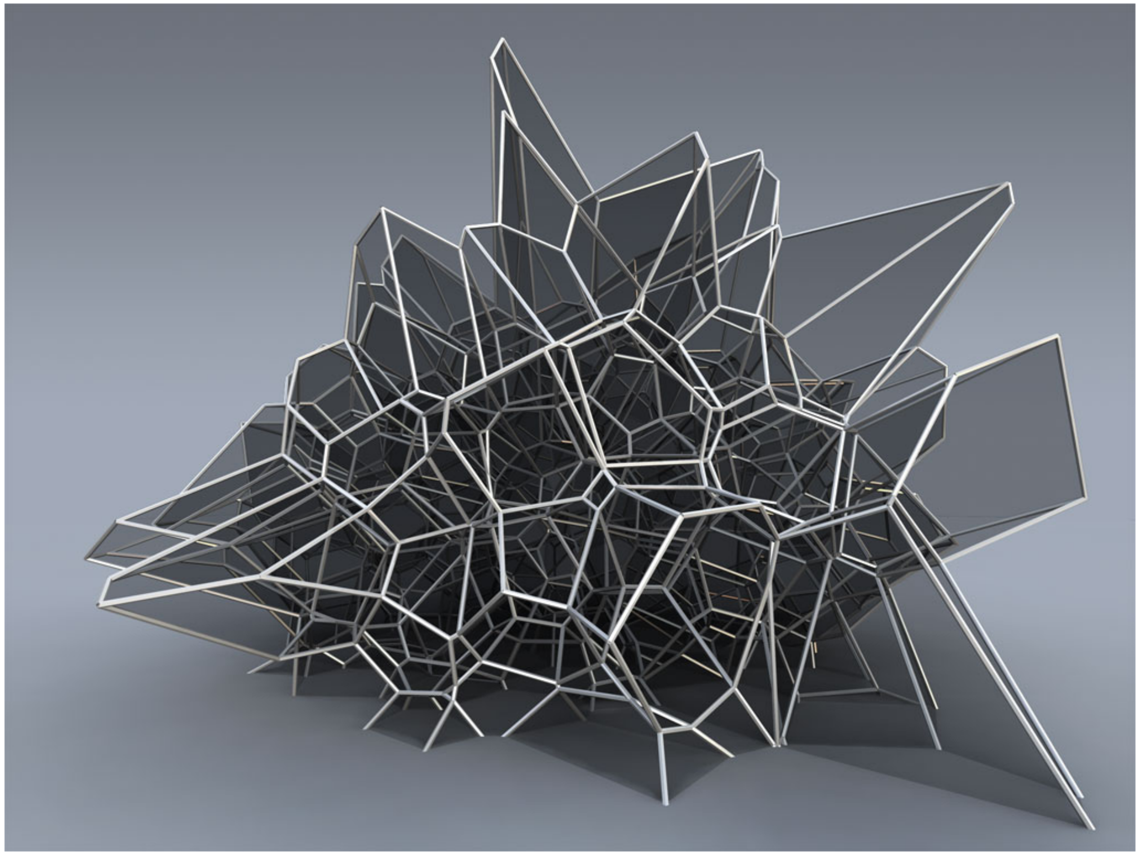


This triangulation fails the circumcircle definition; we flip the inner edge and it becomes Delaunay.
(Image from the wonderful people at Wikipedia.)



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

Voronoi cells in 3D



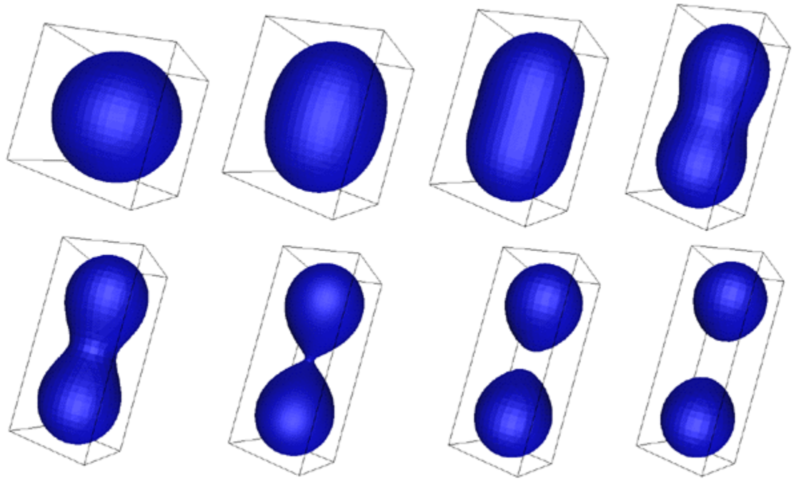
Silvan Oesterle, Michael Knauss

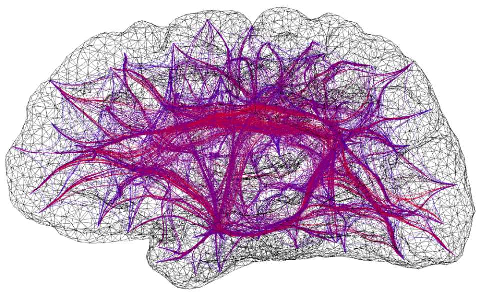
Implicit surfaces

Implicit surface modeling[2] is a way to produce very 'organic' or 'bulbous' surfaces very quickly without subdivision or NURBS.

Uses of implicit surface modelling:

- Organic forms and nonlinear shapes
- Scientific modeling (electron orbitals, gravity shells in space, some medical imaging)
- Muscles and joints with skin
- Rapid prototyping
- CAD/CAM solid geometry



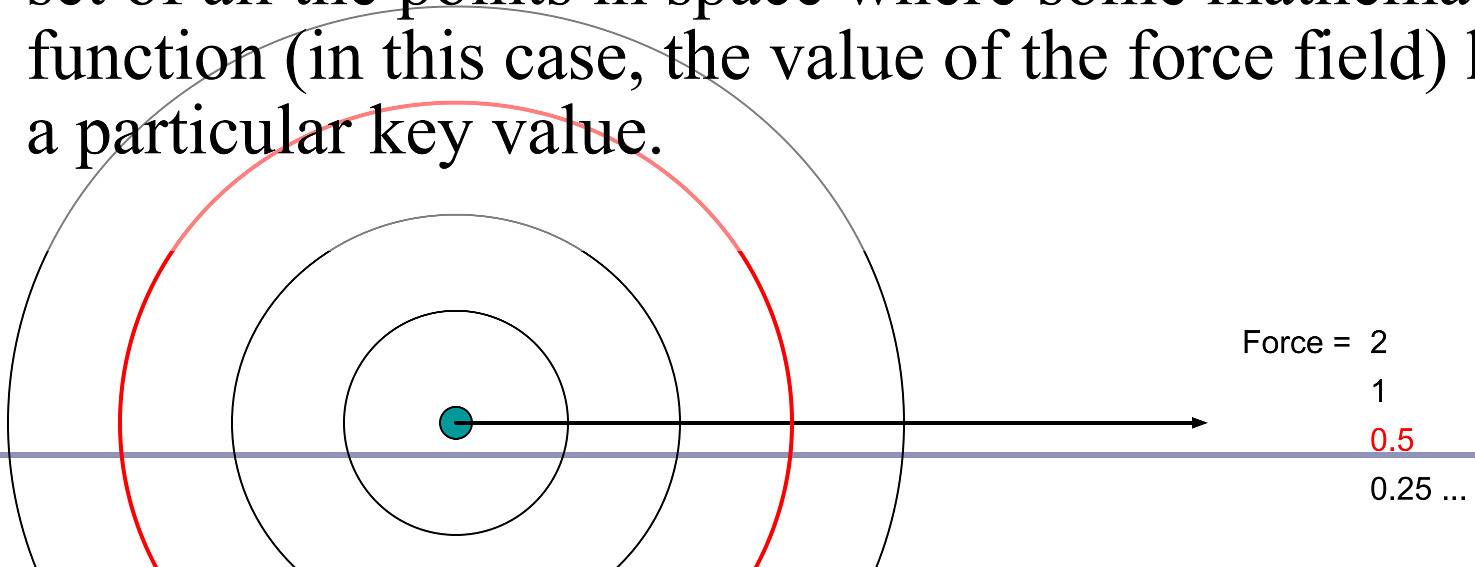


[2] AKA "*metaball modeling*", "*force functions*", "*blobby modeling*"…

How it works

The user controls a set of *control points*, like NURBS; each point in space generates a field of force, which drops off as a function of distance from the point (like gravity weakening with distance.)

This 3D field of forces defines an *implicit surface*: the set of all the points in space where some mathematical function (in this case, the value of the force field) has a particular key value.



Force functions

A few popular force field functions:

- "Blobby Molecules" – Jim Blinn

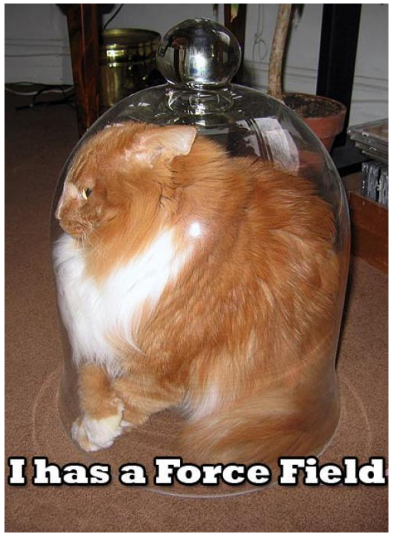
  $F(r) = a\,e^{-br2}$

- "Metaballs" – Jim Blinn

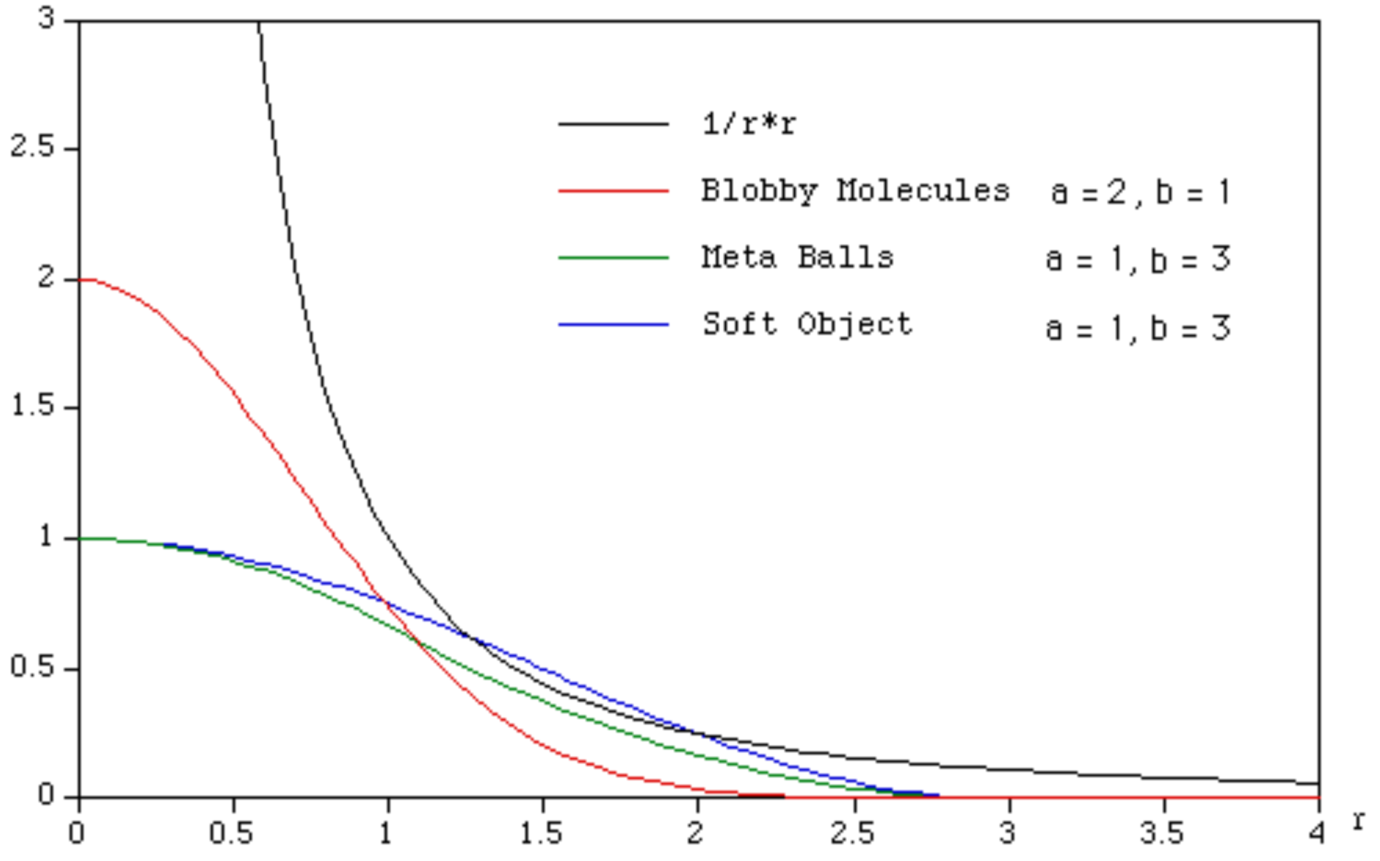
$$F(r) = \begin{cases} a(1 - 3r^2 / b^2) & 0 \leq r < {}^{b}/_{3} \\ (3a/2)(1-r/b)^2 & {}^{b}/_{3} \leq r < b \\ 0 & b \leq r \end{cases}$$

- "Soft Objects" – Wyvill & Wyvill

  $F(r) = a(1 - 4r^6/9b^6 + 17r^4/9b^4 - 22r^2 / 9b^2)$



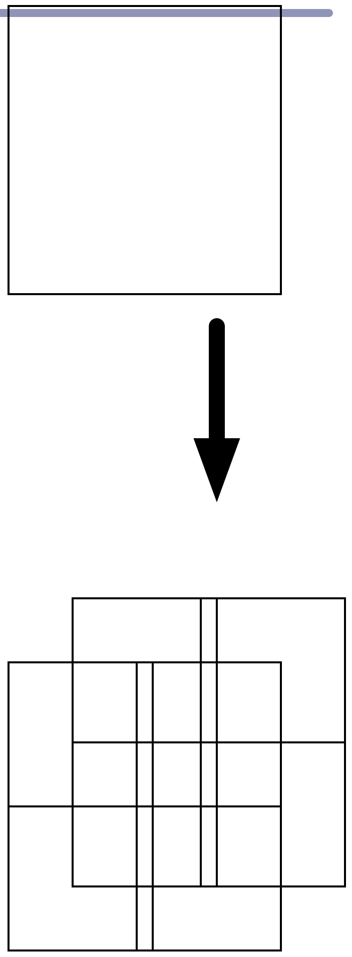
Comparison of force functions



Discovering the surface

An *octree* is a recursive subdivision of space which "homes in" on the surface, from larger to finer detail.

- An octree encloses a cubical volume in space. You evaluate the force function F(v) at each vertex v of the cube.
- As the octree subdivides and splits into smaller octrees, only the octrees which contain some of the surface are processed; empty octrees are discarded.



Polygonalizing the surface

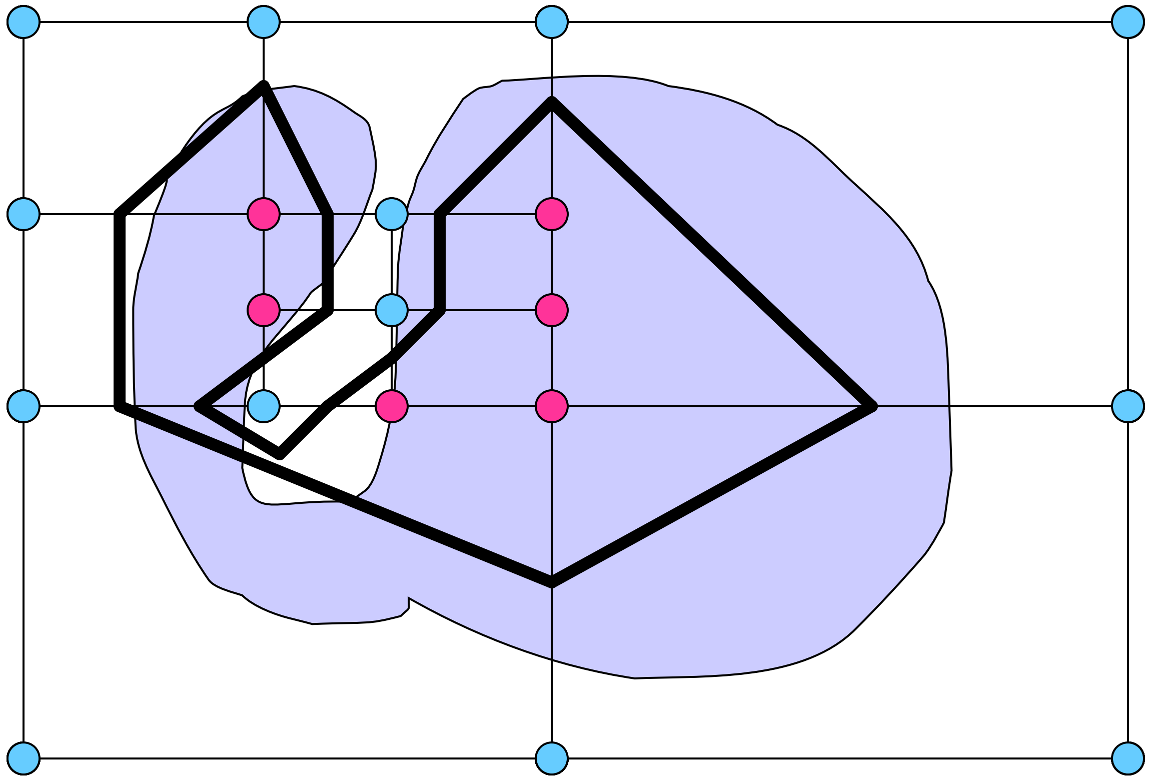
To display a set of octrees, convert the octrees into polygons.

- If some corners are “hot” (above the force limit) and others are “cold” (below the force limit) then the implicit surface crosses the cube edges in between.
- The set of midpoints of adjacent crossed edges forms one or more rings, which can be triangulated. The normal is known from the hot/cold direction on the edges.

To refine the polygonalization, subdivide recursively; discard any child whose vertices are all hot or all cold.

Polygonalizing the surface

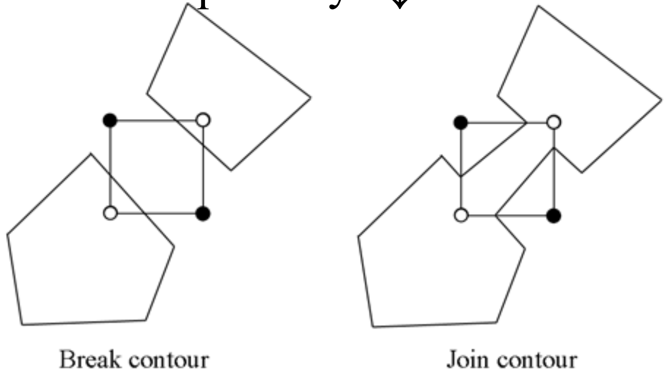
Recursive subdivision (on a quadtree):

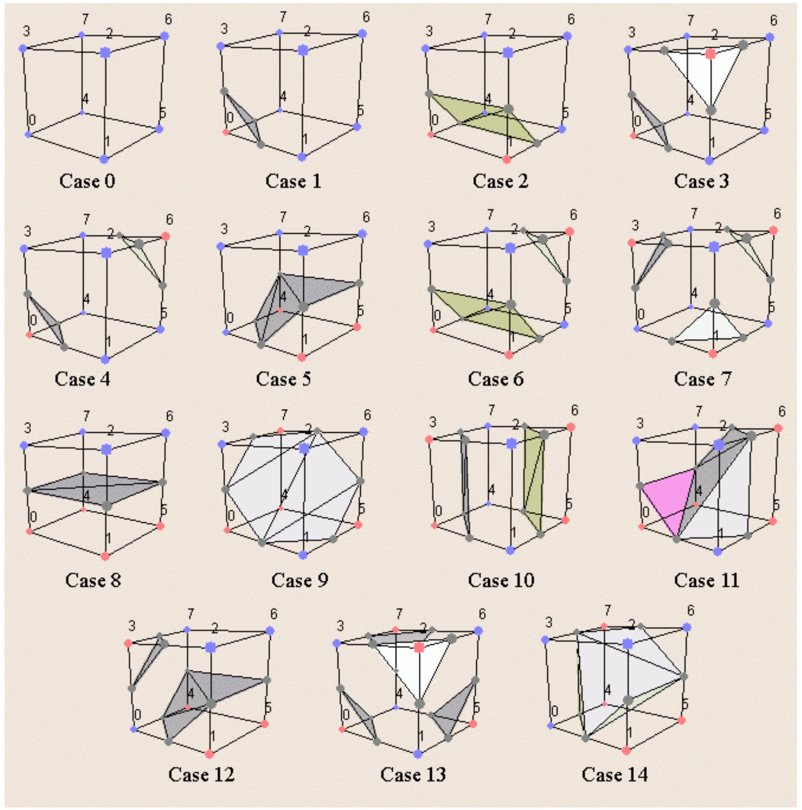


Polygonalizing the surface

There are fifteen possible configurations (up to symmetry) of hot/cold vertices in the cube. →

With rotations, that's 256 cases. Beware: there are *ambiguous cases* in the polygonalization which must be addressed separately. ↓





Images courtesy of Diane Lingrand

Polygonalizing the surface

One way to overcome the ambiguities that arise from the cube is to decompose the cube into tretrahedra.

- A common decomposition is into five tetrahedra. →
- Caveat: need to flip every other cube. (Why?)
- Can also split into six.

Another way is to do the subdivision itself on tetrahedra—no cubes at all.

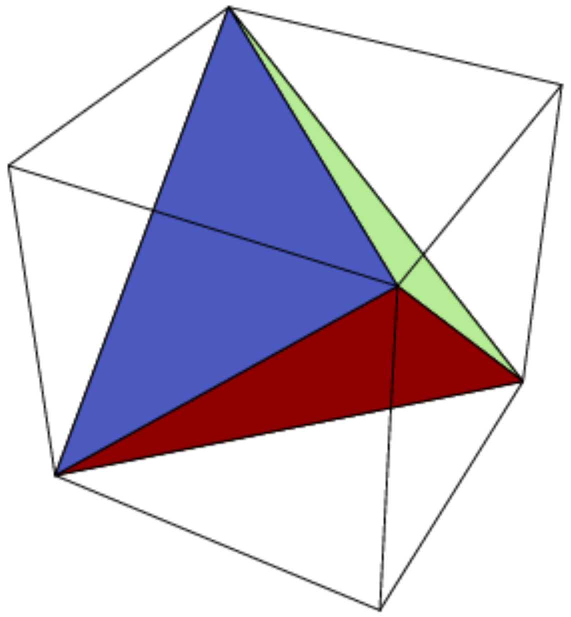


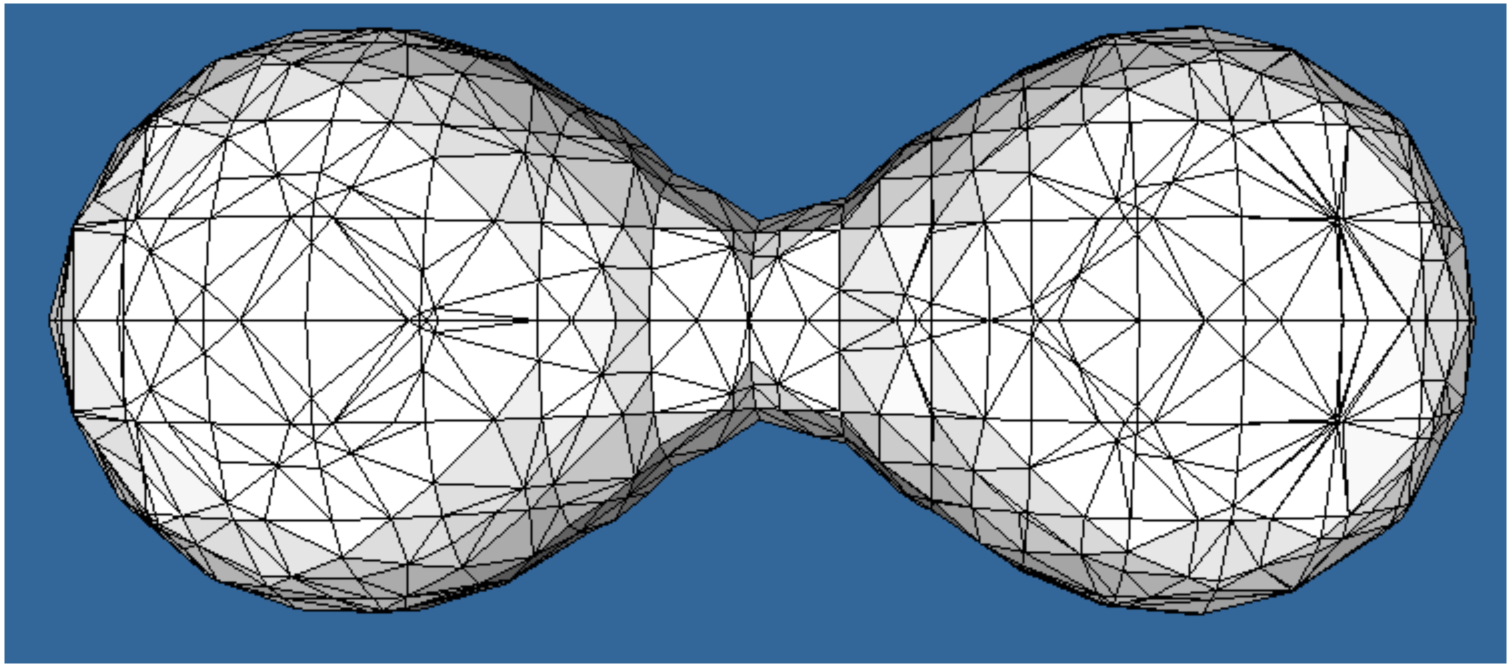
Image from the Open Problem Garden

Smoothing the surface

Improved edge vertices

- The naïve implementation builds polygons whose vertices are the midpoints of the edges which lie between hot and cold vertices.
- The vertices of the implicit surface can be more closely approximated by points linearly interpolated along the edges of the cube by the weights of the relative values of the force function.
  - $t = (0.5 - F(P1)) / (F(P2) - F(P1))$
  - $P = P1 + t\,(P2 - P1)$

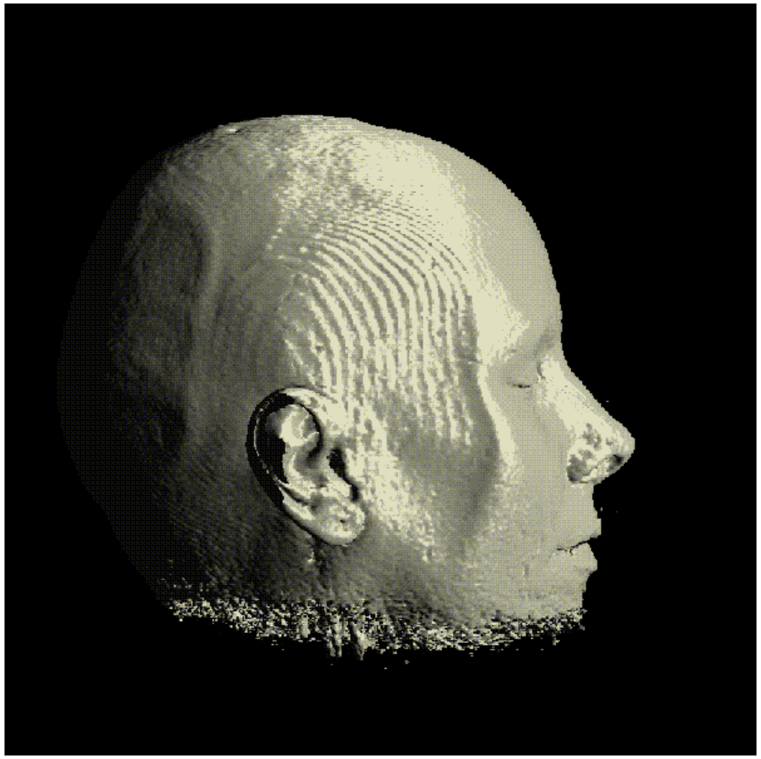
Implicit surfaces -- demo



Marching cubes

An alternative to octrees if you only want to compute the final stage is the *marching cubes* algorithm (Lorensen & Cline, 1985):

- Fire a ray from any point known to be inside the surface.
- Using Newton's method or binary search, find where the ray crosses the surface.
  - Newton: derivative estimated from discrete local sampling
  - There may be many crossings
- Drop a cube around the intersection point: it will have some vertices hot, some cold.
- While there exists a cube which has at least one hot vertex and at least one cold vertex on a side and no neighbor on that side, create a neighboring cube on that side. Repeat.

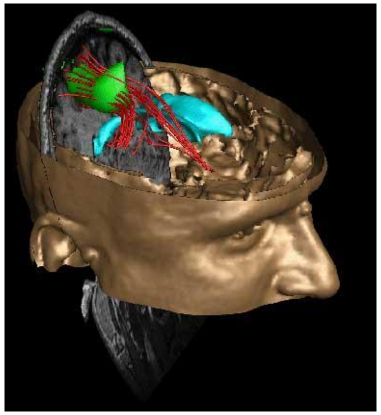


Marching cubes is common in medical imaging such as MRI scans.
It was first demonstrated (and patented!) by researchers at GE in 1984, modeling a human spine.

Voxels and volume rendering

A *voxel* ("volume pixel") is a cube in space with a given color; like a 3D pixel.

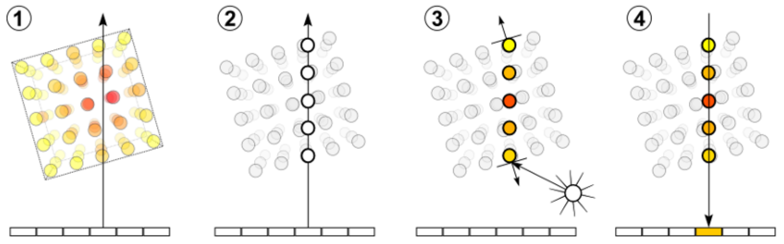
- Voxels are often used for medical imaging, terrain, scanning and model reconstruction, and other very large datasets.
- Voxels usually contain color but could contain other data as well—flow rates (in medical imaging), density functions (analogous to implicit surface modeling), lighting data, surface normals, 3D texture coordinates, etc.
- Often the goal is to render the voxel data directly, not to polygonalize it.



Volume ray casting

If speed can be sacrificed for accuracy, render voxels with *volume ray casting*:

- Fire a ray through each pixel;
- Sample the voxel data along the ray, computing the weighted average (*trili* filter) of the contributions to the ray o each voxel it passes through or near;
- Compute surface gradient from of each voxel from local sampling; generate surface normals; compute lighting with the standard lighting equation;
- ‘Paint’ the ray from back to front, occluding more distant voxels with nearer voxels; this gives hidden-surface removal and easy support for transparency.

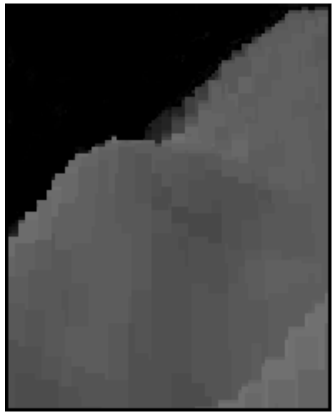


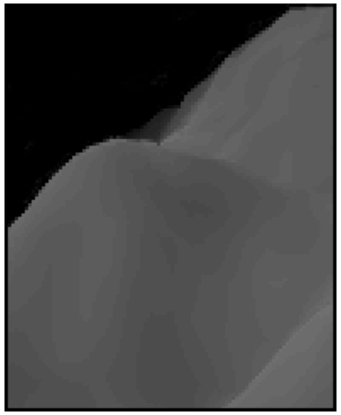
The steps of volume rendering; a volume ray-cast skull. Images from wikipedia.

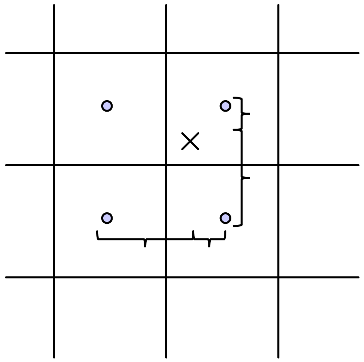
Sampling in voxel rendering

Why trilinear filtering?

- If we just show the color of the voxel we hit, we'll see the exact edges of every cube.
- Instead, choose the weighted average between adjacent voxels.
  - Trilinear: averaging across X, Y, and Z







Your sample will fall somewhere between eight (in 3d) voxel centers. Weight the color of the sample by the inverse of its distance from the center of each voxel.

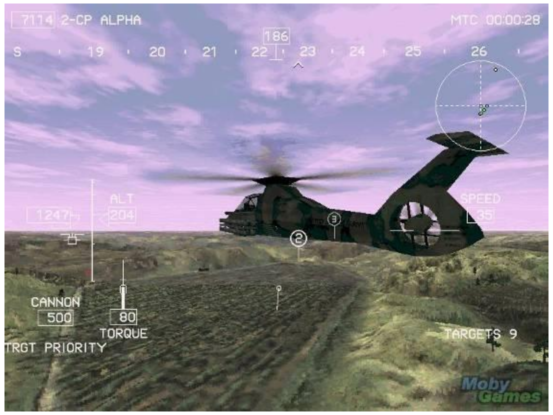
Reasonably fast voxels

If speed is of the essence, cast your rays but stop at the first opaque voxel.

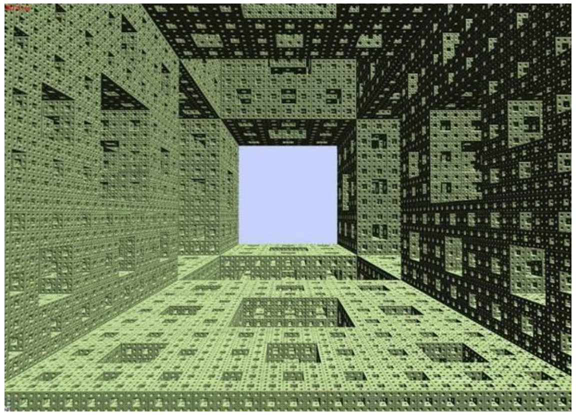
- Store precomputed lighting directly in the voxel
- Works for diffuse and ambient but not specular
- Popular technique for video games (e.g. Comanche →)

Another clever trick: store voxels in a *sparse voxel octree*.

- Watch for it in id's next-generation engine…



Comanche Gold, NovaLogic Inc (1998)

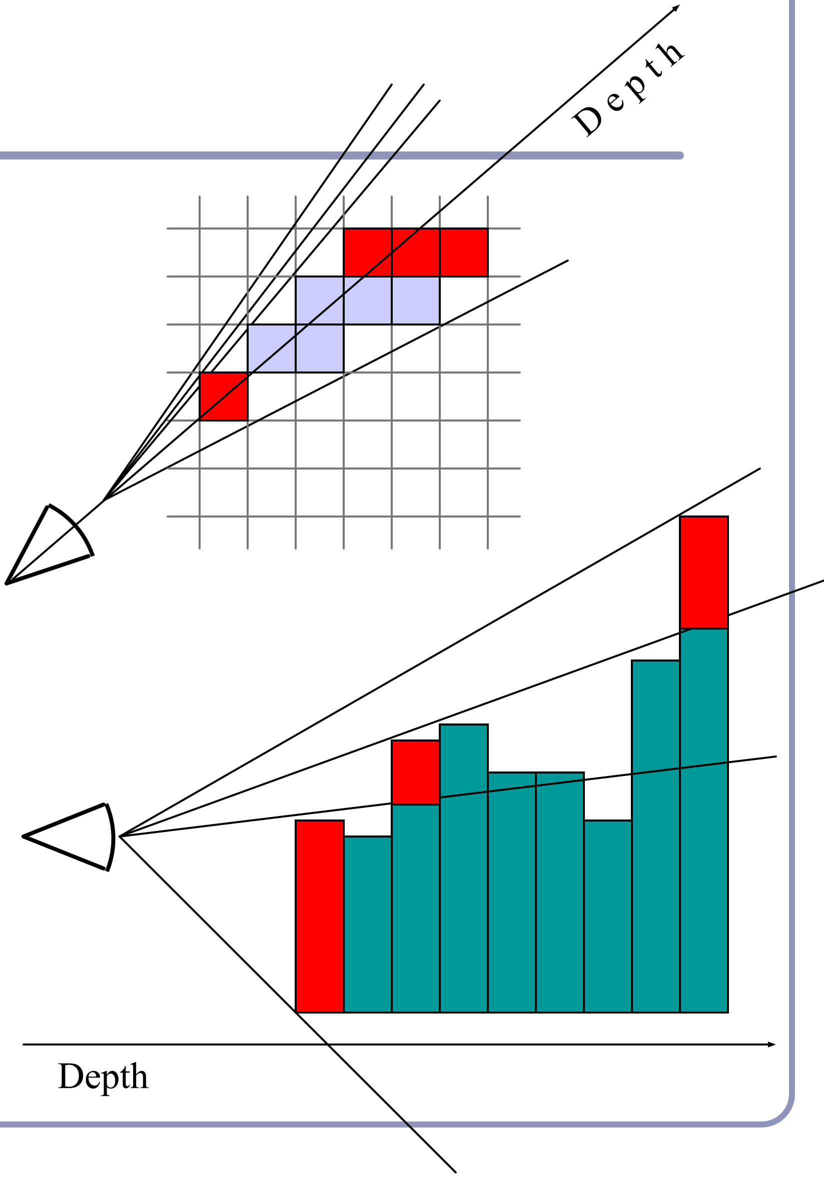


Sparse Voxel Octree Ray-Casting, Cyril Crassin

Ludicrously fast voxels

If speed is essential (like if, say, you're writing a video game in 1992) and you know that your terrain can be represented as a height-map (ie., without overhangs), replace ray-casting with 'column'-casting and use a "*Y-buffer*":

- Draw from *front to back*, drawing columns of pixels from the bottom of the screen up. For each pixel in receding order, track the current max *y* height painted and only draw new pixels above that *y*. Anything shorter must be behind something that's nearer, and it's shorter; so don't draw it.



Optimizing first contact in hardware

To accelerate first raycast, don’t raycast: use existing hardware.

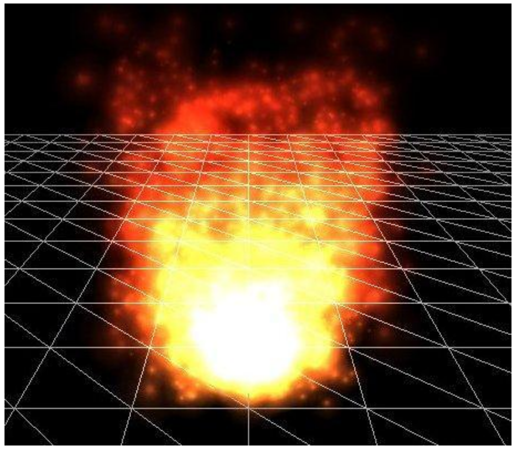
- Use hardware rendering (eg OpenGL) to write to an offscreen 32-bit buffer.
- Set the color of each primitive equal to a pointer to that primitive.
- Render your scene in gl with z-buffering and no lighting.
- The ‘color’ value at each pixel in the buffer is now a pointer to the primitive under that pixel.

Particle systems

Particle systems are a monte-carlo style technique which uses thousands (or millions) or tiny graphical artefacts to create large-scale visual effects.

Particle systems are used for hair, fire, smoke, water, clouds, explosions, energy glows, in-game special effects and much more.

The basic idea:
"If lots of little dots all do something the same way, our brains will see the thing they do and not the dots doing it."



A particle system created with 3dengfx, from wikipedia.

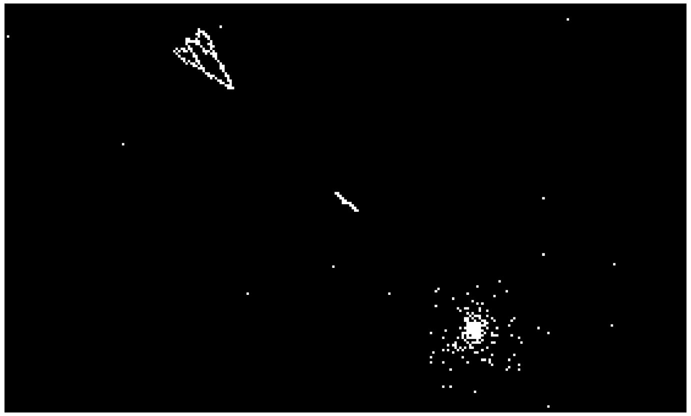
Screenshot from the game *Command and Conquer 3* (2007) by Electronic Arts; the "lasers" are particle effects.

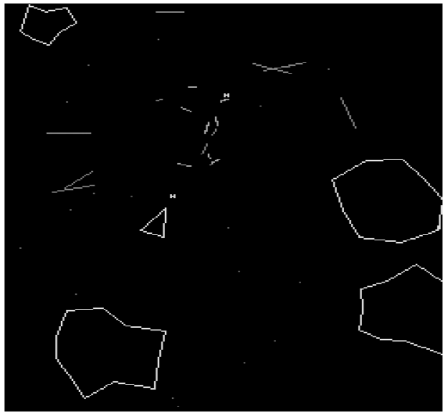
History of particle systems

1962: Ships explode into pixel clouds in "Spacewar!", the 2nd video game *ever*.

1978: Ships explode into broken lines in "Asteroid".

1982: The Genesis Effect in "*Star Trek II: The Wrath of Khan*".





Fanboy note: OMG. You can play the original Spacewar! at http://spacewar.oversigma.com/ -- the actual original game, running in *a PDP-1 emulator inside a Java applet*.

“The Genesis Effect” – William Reeves *Star Trek II: The Wrath of Khan* (1982)

Particle systems

How it works:

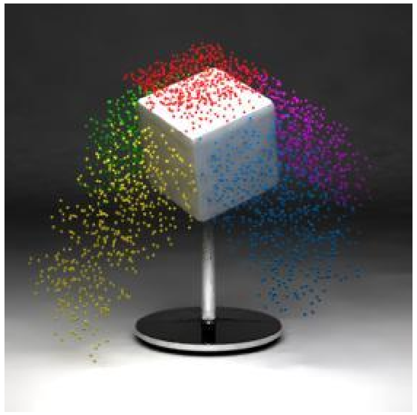
- Particles are generated from an *emitter*.
  - Emitter position and orientation are specified discretely;
  - Emitter rate, direction, flow, etc are often specified as a bounded random range (monte carlo)
- Time ticks; at each tick, particles move.
  - New particles are generated; expired particles are deleted
  - Forces (gravity, wind, etc) accelerate each particle
  - Acceleration changes velocity
  - Velocity changes position
- Particles are rendered.

Particle systems — emission

Each frame, your emitter will generate new particles.

Here you have two choices:

- Constrain the average number of particles generated per frame:
  - # new particles = average # particles per frame + rand() * variance
- Constrain the average number of particles per screen area:
  - # new particles = average # particles per area + rand() * variance * screen area



Transient vs persistent particles emitted to create a ‘hair’ effect (source: Wikipedia)

Particle systems — integration

Each new particle will have at least the following attributes:

- initial position
- initial velocity (speed and direction)

You now have a choice of integration technique:

- Evaluate the particles at arbitrary time *t* as a closed-form equation for a stateless system.
- Or, use iterative (numerical) integration:
  - Euler integration
  - Verlet integration
  - Runge-Kutta integration

Particle systems — two integration shortcuts:

Closed-form function:

- Represent every particle as a parametric equation; store only the initial position p_0, initial velocity v_0, and some fixed acceleration (such as gravity *g*.)
- $p(t)=p_0+v_0t+½gt^2$

No storage of state

- *Very* limited possibility of interaction
- Best for water, projectiles, etc—non-responsive particles.

Discrete integration:

- Remember your physics—integrate acceleration to get velocity:
  - $v'=v + a \bullet \Delta t$
- Integrate velocity to get position:
  - $p'=p + v \bullet \Delta t$
- Collapse the two, integrate acceleration to position:
  - $p''=2p'-p + a \bullet \Delta t^2$

Timestep must be nigh-constant; collisions are hard.

Particle systems—rendering

Can render particles as points, textured polys, or primitive geometry

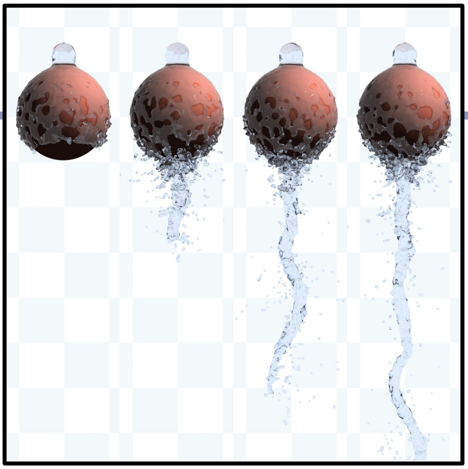
- Minimize the data sent down the pipe!
- Polygons with alpha-blended images make pretty good fire, smoke, etc

Transitioning one particle type to another creates realistic interactive effects

- Ex: a ‘rain’ particle becomes an emitter for ‘splash’ particles on impact

Particles can be the force sources for a blobby model implicit surface

- This is sometimes an effective way to simulate liquids



Hagit Schechter
http://www.cs.ubc.ca/~hagitsch/Research/

nvidia

References

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
Implicit modelling:
D. Ricci, *A Constructive Geometry for Computer Graphics,* Computer Journal, May 1973
J Bloomenthal, *Polygonization of Implicit Surfaces,* Computer Aided Geometric Design, Issue 5, 1988
B Wyvill, C McPheeters, G Wyvill, *Soft Objects*, Advanced Computer Graphics (Proc. CG Tokyo 1986)
B Wyvill, C McPheeters, G Wyvill, *Animating Soft Objects,* The Visual Computer, Issue 4 1986
http://astronomy.swin.edu.au/~pbourke/modelling/implicitsurf/
http://www.cs.berkeley.edu/~job/Papers/turk-2002-MIS.pdf
http://www.unchainedgeometry.com/jbloom/papers/interactive.pdf
http://www-courses.cs.uiuc.edu/~cs319/polygonization.pdf
Voxels:
J. Wilhelms and A. Van Gelder, *A Coherent Projection Approach for Direct Volume Rendering*, Computer Graphics, 35(4):275-284, July 1991.
http://en.wikipedia.org/wiki/Volume_ray_casting
Particle Systems:
William T. Reeves, "*Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*", Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83).
Lutz Latta, *Building a Million Particle System*, http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf , 2004
http://en.wikipedia.org/wiki/Particle_system