Implicit Surfaces & Polygon Meshes

Polygon Meshes
Implicit Surfaces
Constructive Solid Geometry

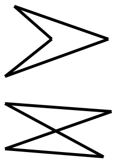
5 Apr. 2000

Modeling Complex Shapes

- **We want to build models of very complicated objects**
- **An equation for a sphere is possible, but how about an equation for a telephone, or a face, or a cloud?**
- **Complexity is achieved using simple pieces**
  - **The pieces could be polygons, parametric surfaces, or implicit surfaces**
- **Requirements**
  - **Model *anything* with arbitrary precision (in principle)**
  - **Easy to build and modify**
  - **Efficient computations (for rendering, collisions, etc.)**
  - **Easy to implement (a minor consideration...)**

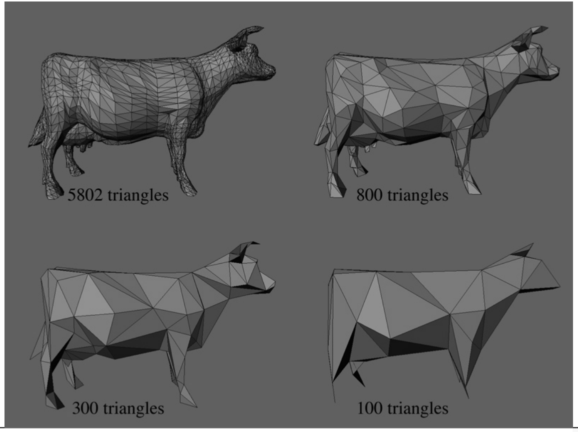
Polygon Meshes

- **Any shape can be modeled out of polygons**
  - **if you use enough of them…**
- **Polygons with how many sides?**
  - **Can use triangles, quadrilaterals, pentagons, … n-gons**
  - **Triangles are most common.**
  - **When > 3 sides are used, ambiguity about what to do when polygon nonplanar, or concave, or self-intersecting.**



- **Polygon meshes are built out of**
  - ***vertices* (points)**
  - ***edges* (line segments between vertices)**
  - ***faces* (polygons bounded by edges)**

How Many Polygons to Use?



A File Format for Polygon Models: OBJ

```
# OBJ file for a 2x2x2 cube
v -1.0  1.0  1.0   - vertex 1
v -1.0 -1.0  1.0   - vertex 2
v  1.0 -1.0  1.0   - vertex 3
v  1.0  1.0  1.0   - …
v -1.0  1.0 -1.0
v -1.0 -1.0 -1.0
v  1.0 -1.0 -1.0
v  1.0  1.0 -1.0
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

Syntax:

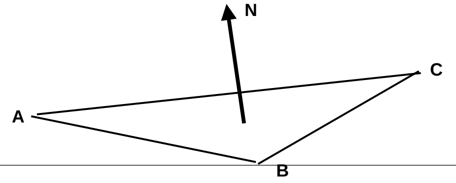
`v` *x y z* - *a vertex at (x,y,z)*

`f` $v_1\ v_2\ \dots\ v_n$
a face with vertices $v_1, v_2, \dots v_n$

`#` *anything* - *comment*

Frontfacing / Backfacing

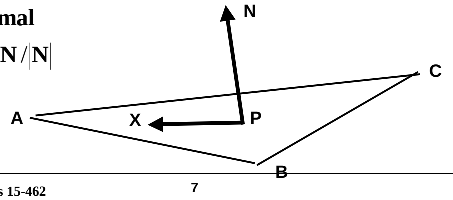
- **A polygon has two sides, of course.**
- **It's customary in CG to use the right hand rule to pick one side to call the *front face*.**
- **Counterclockwise = front, clockwise = back**
- **This is important for:**
  - **lighting**
  - **backface culling**
  - **For the triangle ABC below, the front face is *up*.**



Normals and Plane Equations

- **Need normals for shading, plane eqns for intersection tests**
- **A *normal* to a plane is a vector that is perpendicular to that plane (two possible choices)**
- **A plane is specified by a point P and a normal vector N**
- **N•(X-P) = 0 if and only if X lies in the plane; this is an *implicit equation* for the plane**
  - **Expand this out: 0 = N•X - N•P = ax + by + cz + d**
- **3 vertices define a plane, its normal is: N=(B-A) x (C-A)**
- **Unit normal**

$$\hat{\mathbf{N}} = \mathbf{N} / |\mathbf{N}|$$



Polygon Models in OpenGL

- **for faceted shading**

```
< calculate face normal n
  using cross product rule >
glNormal3fv(n);
glBegin(GL_POLYGONS);
glVertex3fv(vert1);
glVertex3fv(vert2);
glVertex3fv(vert3);
glEnd();
```

- **for smooth shading**

```
glBegin(GL_POLYGONS);
glNormal3fv(normal1);
glVertex3fv(vert1);
glNormal3fv(normal2);
glVertex3fv(vert2);
glNormal3fv(normal3);
glVertex3fv(vert3);
glEnd();
```

Data Structures for Polygon Meshes

- **Simplest (but dumb)**
  - `float triangle[n][3][3];` **(each triangle stores 3 (x,y,z) points)**
  - **redundant: each vertex stored multiple times**
- **Vertex List, Face List**
  - **List of vertices, each vertex consists of (x,y,z)**
    geometric (shape) info only
  - **List of triangles, each a triple of vertex id's (or pointers)**
    topological (connectivity, adjacency) info only

  Fine for many purposes, but finding the faces adjacent to a vertex takes O(F) time for a model with F faces. Such queries are important for topological editing.

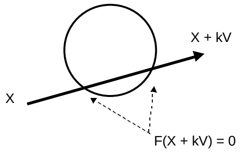
- **Fancier schemes:**

  Store more topological info so adjacency queries can be answered in O(1) time.

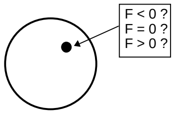
  ***Winged-edge data structure* – edge structures contain all topological info (pointers to adjacent vertices, edges, and faces).**

  ***Quad-edge data structure* – a variant of the above.**

What Implicit Functions are Good For



Ray - Surface Intersection Test

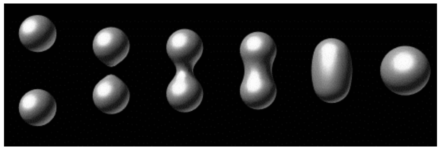


Inside/Outside Test

Surfaces from Implicit Functions

- **Constant Value Surfaces are called (depending on whom you ask):**
  - **constant value surfaces**
  - **level sets**
  - **isosurfaces**
- **Nice Feature: you can add them! (and other tricks)**
  - **this merges the shapes**
  - **When you use this with spherical exponential potentials, it's called *Blobs*, *Metaballs,* or *Soft Objects*. Great for modeling animals.**

Blobby Models

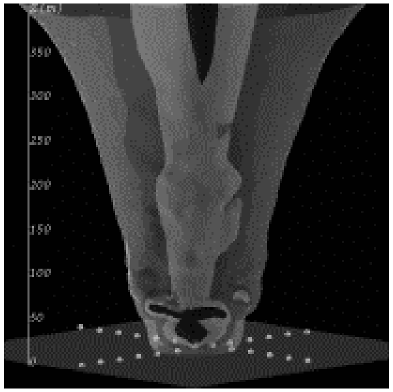


- **Implicit function is the sum of Gaussians centered at several points in space, minus a threshold**
- **varying the standard deviations of the Gaussians makes each blob bigger**
- **varying the threshold makes blobs merge or separate**

How to draw implicit surfaces?

- **Note: the surface normal is -Grad F = -∇ F**
- **It’s easy to ray trace implicit surfaces**
  - **(as we’ll see later…)**
- **Volume Rendering can display them**
- **Convert to polygons: the Marching Cubes algorithm**
  - **Divide space into cubes**
  - **Evaluate implicit function at each cube vertex**
  - **Do root finding or linear interpolation along each edge**
  - **Polygonize on a cube-by-cube basis**

Isosurfaces of Simulated Tornado



Sets of Points, Surfaces and Solids

- **Implicit surface: set of all points that satisfy F(x,y,z)=0**
- **The points that satisfy F(x,y,z)<0 define a solid (or solids) bounded by the surface**
- **The solid is directly defined (unlike definitions using parametric surfaces)**
- **Example**
  - **An infinitely long (solid) cylinder with radius r:**

$$F = x^2 + y^2 - r^2$$

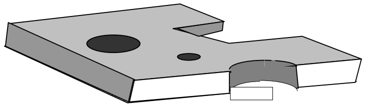
  - **To limit cylinder to length L, abs(z) < L/2 and keep the function implicit use max:**

$$F = \max\left(\operatorname{abs}(z) - L/2, x^2 + y^2 - r^2\right)$$

- **Implicit functions for a cube? Any convex polyhedron?**

Constructive Solid Geometry (CSG)

- **Generate complex shapes with basic building blocks**
  - **machine an object - saw parts off, drill holes**
  - **glue pieces together**
- **This is sensible for objects that are actually made that way**

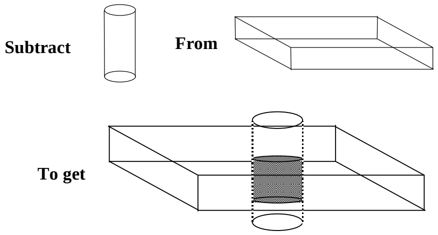


A CSG Train

Brian Wyvill & students, Univ. of Calgary

Negative Objects

- Use point-by-point boolean functions
  - remove a volume by using a negative object
  - e.g. drill a hole by subtracting a cylinder



Inside(BLOCK-CYL) = Inside(BLOCK) And Not(Inside(CYL))

Set Operations

- UNION: Inside(A) || Inside(B)
  — Join A and B
- INTERSECTION: Inside(A) && Inside(B)
  — Chop off any part of A that sticks out of B.
- SUBTRACTION: Inside(A) && (! Inside(B))
  — Use B to Cut A

Examples:

- Use cylinders to drill holes
- Use rectangular blocks to cut slots
- Use half-spaces to cut planar faces
- Use surfaces swept from curves as jigsaws, etc.

Implicit Functions for Booleans

- Recall the implicit function for a solid: F(x,y,z)<0
- Boolean operations are replaced by arithmetic:
  - MINUS replaces NOT(unary subtraction)
  - MAX replaces AND (intersection)
  - MIN replaces OR (union)

- Thus
  - F(Subtract(A,B)) = MAX(F(A), -F(B))
  - F(Intersect(A,B)) = MAX(F(A),F(B))
  - F(Union(A,B)) = MIN(F(A),F(B))

You *can* try this at home

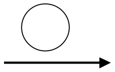
- **Drawing boolean objects - combine parametric and implicit functions**
- **The boolean object has surfaces from all its constituent objects**
- **Draw using polygonal meshes, but test before drawing using implicit func.**
  - **for a hole drilled in a block - the surface of the hole is given by the cylinder used to drill it, the rest of the object's surface is defined by the block**
  - **draw points on the block if they are outside the cylinder**
  - **draw points on the cylinder if they are inside the block**
- **Implementing union:**
  - **draw both objects, use hidden-surface algorithms to take care of visibility**
- **Implementing intersection:**
  - **draw points only if they are inside both objects**
- **Implementing subtraction**
  - **points on the *positive* object's surface are visible outside the negative object**
  - **points on the *negative* object's surface are visible inside the positive object**
- **Draw using parametric functions, trim using implicit functions**
  - **And that's where the tricky part comes in**

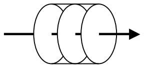
3-D Object Representation

- **Individual elements are *voxels* (volume elements)**
- **Compression is almost mandatory**
- **Use octrees (3-D version of quadtree)**
  - **adaptively subdiviside a cube into 8 sub-cubes forming a tree**
  - **stop dividing when the whole cube is entirely full or empty, or the minimum resolution is reached**
  - **at minimum resolution fill the block if majority is full**
  - **combine sibling cubes if they all have the same state**
- **Partially full cubes are nodes, full or empty cubes are leaves**
- **Data space requirement is proportional to the surface area of the object (except a few worst cases)**

Generalized Cylinders

- **Book calls these *Sweep Representations***
  - **Specify 2D cross-sectional shape**
  - **Specify 3D skeleton curve**
  - **"Sweep" cross-sectional shape along skeleton curve**
    - » **shape normal should be parallel to curve tangent**
- **Simple example: cylinder**
  - **cross-section is circle**
  - **skeleton is line**





- **Can vary cross-sectional shape for more interesting sweeps**
- **You will be implementing these for Assignment 2**