Modeling from scan data

- Components
  - Scanning
    - example using structured light:
      - calibrate a light source and a camera
      - project light stripe so it hits the scanned object
      - which pixel in camera image is illuminated determines 3D position
  - Registration
    - of scans into a single coordinate system
  - Merging
    - scans into a single representation
  - Representing meshes
  - Compressing meshes

Scanning

- Sweep a vertical light stripe in small steps
- Detect the stripe from camera images
- Project a line from a pixel to the right image
- Triangulate

Scanner setup

Dense range (and color) from stereo + active light

- 4 Sony video cameras
- light projector
- turntable
- lamps

- Relative orientation of laser and camera known (calibrated)
  - in this case, the more left the curve on camera image, the closer it is to the laser source
Two sub-problems to solve registration

- **Correspondence**
  - Given a point in one view, which point in the other view corresponds to the same surface point?

- **Alignment**
  - Given point correspondences, which rigid motion aligns the surfaces (or paired points)?

- Which is harder?

  Correspondence. It is a combinatorial problem, and we can usually solve it only indirectly (just comparing two points, it is hard to say whether they match unless you consider a lot of other data) whereas there is a closed form solution for the alignment problem.

Iterative closest points (ICP)

- Start from an approximate registration
- Pairing heuristic
  - match a point with the closest compatible point
  - compatibility heuristics typically local surface properties such as color, normal, curvature, ...
- Iterate between solving the correspondence and alignment problems
  - # P, Q are point sets
  - while not close_enough(P, Q):
    - for p in P:
      - # find matching pairs
        - pairs.append((p, Q.closest(p)))
    - Q.move_closer(pairs) # align

Multiview alignment

- In pairwise registrations small errors accumulate
  - scans A, B, C, D, E, F
  - A and B registered, very small error
  - same with others
  - F and A should close the circle, but F has drifted far from A
- Solution:
  - gather solutions for pairwise alignments
  - enforce them all at the same time
  - especially make sure that F must remain close to A
    - gives a strong constraint for correct registration
  - this diffuses the errors evenly among all the pairs
Merging

- Registration moves meshes into a single coordinate system
- We still need to merge them into a single mesh
- First, a couple of preliminaries
  - signed-distance functions and isosurfaces
  - the marching cubes algorithm
- Then a few methods to create (or merge) meshes

Signed-distance function, isosurface

- An implicit definition of surface
- Create a function $f$ that you can evaluate at any $(x,y,z)$
  - returns a distance to the surface
  - distance from outside of the object has a different sign (say negative) than from inside
  - don’t need to explicitly represent this, only evaluate
- The surface is defined by the isosurface $f(x,y,z) = 0$
- Extract the surface using, e.g., the marching cubes algorithm
  - coming in a few slides…

Marching cubes

- An algorithm for extracting isosurfaces $f(x,y,z) = c$
- Evaluate
  - in a regular grid
  - whether $f(x,y,z) > c$ or $f(x,y,z) \leq c$
- The grid forms a set of cubes
  - find a cube that intersects the surface
  - triangulate the isosurface within the cube
  - propagate the cubes along the surface
  - obtain the full isosurface, one cube at a time
- Create a table of all vertex colorings
  - black if $< c$
  - white otherwise
  - store the triangle connectivity
  - how many entries in the table?
  \[ 2^8 = 256 \]

14 distinct entries

- The rest 242 entries are various rotations/mirrorings of these
Tessellations

- Introduce a vertex on each edge with vertices of opposite colors
  - the structure (triangle tessellations) is pre-calculated
  - the exact vertex location on the edge e.g. by linear interpolation:
    at start \( f(x) = -1 \), at end \( f(x) = 2 \) => put the vertex at 1/3 of the way

The ambiguity problem

- Tessellations not unique!
- We don't have enough information to always make the correct interpretation
- Have to be careful in order to not introduce holes
- One possible solution:
  - create a table of all 256 entries
  - figure out which entries could be neighbors
    - i.e., if the opposite walls have a matching pattern of black and white vertices
  - make sure that tessellations and the triangle edges match

Marching squares Java applet

- A 2D isosurface defined by two centers and radii
  - the function is symmetric, but sampling on a regular grid produces irregular shapes
  - notice how the extracted regions join already before the functions join

A mesh by combining patches

- Example: zippering
  - Turk & Levoy 94
- Two overlapping patches
- Figure out the overlap
- Keep one of the meshes
  - or peel from both meshes until barely overlap
- Clip the other mesh to the boundary of the mesh we'll keep
- Re-triangulate, connect
**A mesh from unorganized points**

- Hoppe *et al.* 92
  - define a signed distance function for a set of unorganized points
  - extract the isosurface using marching cubes
- Preprocessing
  - for each 3D point, define an outside normal vector
    - fit a plane to the point and \( n \) neighbors
  - create a data structure for finding closest points (e.g., KD-tree)
- Evaluation
  - for a given point \( p \), find the closest point \( c \)
  - determine the distance from its tangent plane
  - if the point is on the outer side of the plane, the distance is positive
    \[ n \cdot (p - c) \]

**A mesh from range grids**

- Curless & Levoy 96
  - Each scan defines the signed-distance function
    - scan by laser-camera combination as we saw before
    - close to the surface along line of sight
  - Combine several registered scans into a voxel grid
    - update a weighted average of the distances for each voxel

**Space carving**

- If you can see a surface, the intervening space is free
- Define
  - unseen (inside) has distance \( D_{\text{max}} = D \) and empty \( D_{\text{min}} = -D \)
  - close to surface it's the distance to the observed surface
  - creates a zero-crossing (=surface) between unseen and empty

**Drill bit example**

- Several noisy scans of a very small drill bit
  - registration from known rotation of target between scans
  - individual meshes very noisy, zippering fails
  - but adding and averaging signed distances into voxels works
    - noise is averaged out, lots of real data remains

*The scanned object*  
*Scan data*
Mesh representations

- Regular grid
- Indexed representation
- Ordered vertex neighbors
- Edge-based data structures
  - Winged-edge
  - Half-edge
  - Quad-edge

Regular grid

- Store the 3D data like an image, just the vertex data
- The faces and edges are implicit
- No overhead to store the connectivity!

Vertex lists and indices

- We saw this already in VRML
  - define a list of vertices
  - connectivity:
    - polygons refer to the vertices by their index
    - special index (-1) to mark end of a polygon
- Pros
  - general
  - simple
  - fast to render
- Cons
  - hard to access neighbors
- How many indices do we need?
  - assume triangle mesh, on the average 6 triangles adjacent to each vertex

Ignoring endings (-1):

- Triangles:  6 indices / vertex
- Triangle strips: >2 indices / vertex

Ordered vertex neighbors

- For each vertex, store indices to the neighbors
  - order the indices consistently, e.g., CCW order
- Need as many indices as with separate polygons
  - number of edges x 2
- Now you can traverse the mesh
  - traversal somewhat awkward
  - but in some applications you need only the neighbors
    (such as subdivision surfaces)
**Winged-edge data structure**

- Designed to make finding neighbors easy
- Contains
  - $E_1$ is this edge
  - edges $E_2$-$E_5$ make out the wings
  - faces $F_1$, $F_2$ both sides of the edge
  - vertices $V_1$, $V_2$ at the ends of the edge
- Faces and vertices have pointers to one incident edge
- Lots of overhead!

**Half-edge data structure**

- Store only "half" of the edge ($E$)
  - the half that belongs to the adjacent face
  - include a pointer to the other half
- Slightly less overhead and a bit more flexible
- If you have only triangles, you can simplify a bit more
  - How?
    $$E_2 \text{ is not needed: } E \rightarrow E_2 = (E \rightarrow E_1) \rightarrow E_1$$

**Quad-edge data structure**

- Simple, elegant, optimal in storage space
  - encodes both the mesh and its dual
    - dual of a mesh means replacing faces with vertices and vice versa
    - what's the dual of a tetrahedron? of a cube? *tetrahedron octahedron*?
  - encodes even non-orientable surfaces
    - Moebius strip and Klein bottle are non-orientable
- But difficult to get the correct intuition when manipulating
- Four directed edges, each with pointers to
  - "next" edge
  - data
- For one pair of edges data is
  - vertices
  - for the other, its faces

**Quad edges**

- Both vertices and faces are defined implicitly as connected loops
**Mesh simplification**
- Find a new smaller mesh that is still similar to the old one
  - when the object is far, and appears small, can draw it cheaper with no visual loss

**Subsampling**
- Simple and fast
- Only for regular meshes
- Severe aliasing artifacts, since which vertices remain are not affected by where the features, such as edges, are

**Bin-and-connect**
- Related to subsampling, but also for irregular meshes
- Procedure
  - subdivide the space into regular bins
  - average all the vertices in a bin into a single new vertex
  - reconnect vertices
- Less aliasing
- Fast
- Still not very good results

**Mesh optimization**
- Assume triangle mesh
- Apply mesh operations:
  - edge can be
    - collapsed to a vertex
    - split into two
    - (must introduce two more)
    - swapped in orientation
- Minimize a function with two terms
  - geometric term tries to keep the mesh close to original data
  - connectivity term favors simpler meshes
- Slow, but quality of output pretty good
Quadric error metric

- Generalized vertex contraction
  - can also contract vertices that are not neighbors, i.e., not connected by an edge

- Avoids fragmenting things that are nearby but not connected

Garland & Heckbert 97

The metric

- Each vertex
  - has a number of associated faces
  - is located close to the faces

- A 4x4 symmetric matrix compactly stores information about the associated planes
  - interpretation: the matrix describes an ellipsoid
  - the optimal location for a vertex is at the center, then it's closest to all associated planes
  - merging vertices combines the ellipsoids
    - merge if the ellipsoid doesn't grow much

Wavelet decomposition

- The signal processing approach
- Break the data into a smoother/sparser representation
  - get wavelet coefficients that encode the difference between the two levels
  - compression: ignore the small coefficients

- Problem: need regular subdivision connectivity!

Remeshing

- Fit a new mesh that has the subdivision connectivity

Lounsbery et al. 94

Eck et al. 95
Then reconstruct

- Now can perform the wavelet decomposition
  - keep only the larger coefficients
  - when they are added back, that creates triangles
  - but on flat portions, triangles remain large

Simplification envelopes

- Create an envelope to the surface
  - two offset surfaces
    - inwards
    - outwards
  - empty space remains between envelopes
- Now simplify the mesh
  - edge/vertex contractions, binning, ...
  - just make sure the mesh remains fully within the envelope

Progressive meshes

- Idea
  - keep track of the vertex contractions to make a continuous set of progressively more and more simplified meshes
  - also consider other data such as colors, normals, ...
  - when adding/removing detail, smoothly morph to avoid "popping"
  - in the end, get the original mesh

Progressive forest split compression

- Taubin et al. 98
- Remove simply connected trees of triangles
- Can encode connectivity for 1-2 bits per triangle
- Encode vertex positions
  - predict positions
  - store differences from predictions
  - vector quantize the differences
Vertex compression

- Quantify the vertex locations to save bits
  - can do in addition to compact storage of connectivity
- Artifacts can be reduced by smoothing in the end