TP — 3D Surface Reconstruction

via Delaunay Triangulation and Alpha Shapes

Course: Geometry Processing / 3D Computer Vision

Tools: Python 3, NumPy, SciPy (Delaunay), Matplotlib, ReportLab

Datasets: Stanford Bunny (35 947 pts) & Bimba (3 700 pts)

---

1. Introduction

Surface reconstruction is a fundamental problem in computer graphics and 3D vision: given an unstructured point cloud acquired by a 3D scanner or similar device, we want to recover a piecewise-linear surface (a triangle mesh) that approximates the original object. The approach explored here is based on **alpha shapes**, a mathematically elegant family of shapes introduced by Edelsbrunner et al. (1983) that generalises the convex hull.

Core idea. Build the full 3D **Delaunay triangulation** of the point set. For a given parameter $\alpha > 0$, a tetrahedron is considered "interior" to the shape if its *circumsphere radius* $r < \alpha$. The reconstructed surface consists of all triangular facets that lie on the boundary between an interior and an exterior tetrahedron. As α grows, the shape grows from nothing to the full convex hull.

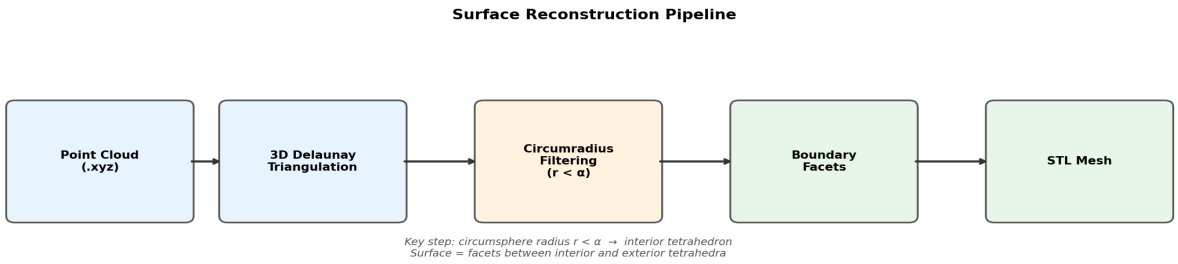

Figure 1 — Processing pipeline from point cloud to STL mesh.

2. Delaunay Triangulation in 3D

A 3D **Delaunay triangulation** of a point set P is a decomposition of the convex hull of P into non-overlapping tetrahedra such that no point of P lies inside the circumsphere of any tetrahedron (empty-sphere property). This property maximises the minimum dihedral angle, producing "well-shaped" cells.

Key CGAL types used in the C++ implementation:

- Exact_predicates_inexact_constructions_kernel — fast kernel with exact geometric predicates.
- Delaunay_triangulation_3 — incremental 3D Delaunay construction.
- Finite_cells_iterator — iterates only over finite (non-infinite) tetrahedra.
- Facet = (Cell_handle, int) — a triangular face identified by its opposite vertex index.

The Python implementation (scipy.spatial.Delaunay) wraps QHull and gives identical combinatorial output. Each row of tri.simplices is a tetrahedron = 4 vertex indices.

3. STL File Format

STL (*STereoLithography*) is the de-facto format for triangle meshes. Each facet is stored as a unit normal followed by three vertex coordinates. The ASCII form used here:

```
solid alpha_shape
  facet normal nx ny nz
    outer loop
      vertex x0 y0 z0
      vertex x1 y1 z1
      vertex x2 y2 z2
    endloop
  endfacet
...
endsolid alpha_shape
```

The normals are computed from the cross product (p1-p0) x (p2-p0), normalised. A binary variant is also implemented (write_stl_binary) which is ~5x smaller.

4. Alpha Shape Filtering — Implementation

4.1 Circumsphere Radius

For a tetrahedron with vertices p0, p1, p2, p3, the circumsphere centre c satisfies ||c - pi|| = R for all i. Subtracting the equation for p0 from the others yields the 3x3 linear system:

```
A = [p1-p0, p2-p0, p3-p0]
b = 0.5 * [||p1-p0||^2, ||p2-p0||^2, ||p3-p0||^2]
c_local = solve(A, b)
R = ||c_local|| (circumradius)
```

4.2 Boundary Facet Detection

For each facet shared by two tetrahedra T1 and T2:

- If R(T1) < α and R(T2) ≥ α (or T2 is infinite): **surface facet** — output to STL.
- If both interior or both exterior: internal or external facet — **skip**.
- A facet on the convex hull boundary (only one adjacent tetrahedron): output if that tetrahedron is interior.

This precisely extracts the oriented boundary of the alpha-complex.

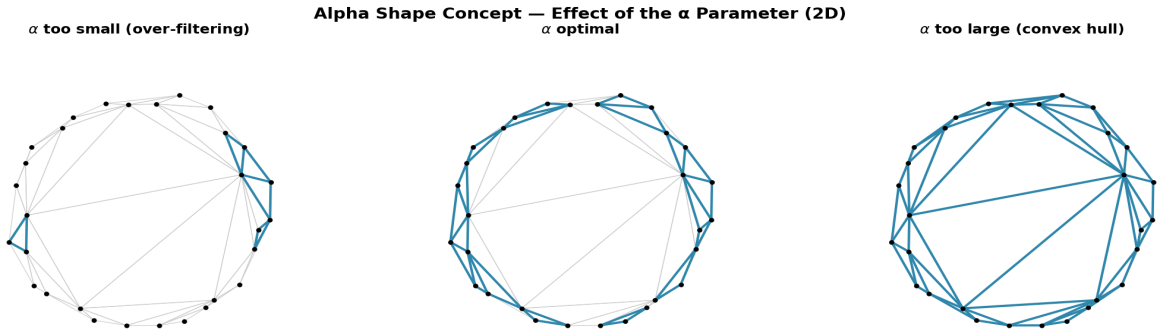

Figure 2 — Effect of α on the alpha shape (illustrated in 2D). Left: α too small (holes appear). Centre: α optimal (clean contour). Right: α too large (degenerates to the convex hull).

4.3 Completed C++ Implementation (reconstruct.cpp)

The key loop added to reconstruct.cpp:

```
for (auto fit = T.finite_facets_begin(); fit != T.finite_facets_end(); ++fit) {
  Cell_handle c = fit->first; int i = fit->second;
  Cell_handle nc = c->neighbor(i);
  bool c_in = !T.is_infinite(c) && circumradius(c) < alpha;
  bool nc_in = !T.is_infinite(nc) && circumradius(nc) < alpha;
  if (c_in != nc_in) { /* output facet as STL triangle */ }
}
```

CGAL::circumcenter() computes the circumcentre; the radius is its distance to any vertex.

5. Automatic Alpha Selection (Bonus)

Choosing α manually requires trial and error. A robust heuristic is:

- **Circumradius percentile.** Compute the circumsphere radius of every Delaunay tetrahedron. Take the p-th percentile (e.g. p = 10) of those radii as α. This captures the "typical" scale of tight local cells while ignoring large outer tetrahedra (which have huge circumradii due to the convex hull).

```
radii = [circumsphere_radius(tet) for tet in tetrahedra]
alpha = np.percentile(radii, 10) # 10th percentile
```

For the Bunny (35 947 points), this gives α **= 0.002436**, producing 76 380 surface triangles.

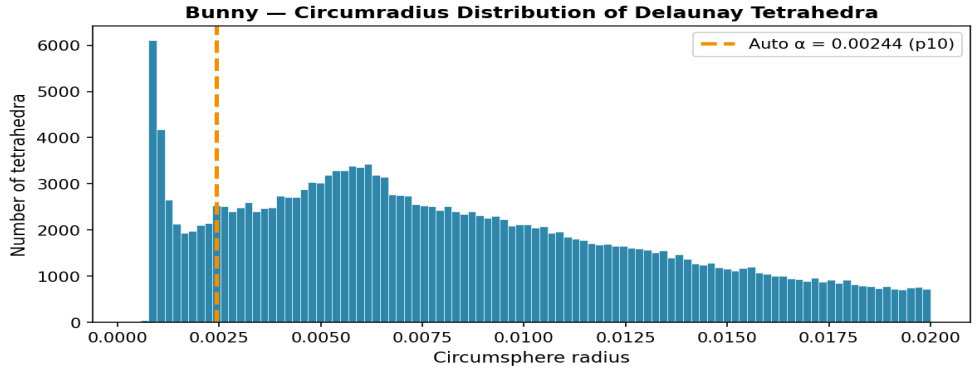

Figure 3 — Distribution of circumsphere radii for Bunny tetrahedra (clipped at 0.02). The orange dashed line shows the auto-selected α at the 10th percentile.

6. Results — Stanford Bunny

The Bunny is a standard uniform-density scan with 35 947 points.

Figure 4 — Bunny reconstruction. Left: input point cloud (3 000-point subset shown). Right: alpha shape surface (sample of 3 000 triangles rendered for clarity).

Metric	Value
Input points	35 947
Delaunay tetrahedra	~230 000
Auto alpha (α)	0.002436
Surface triangles	76 380
Output file	bunny_reconstruction.stl (ASCII)

7. Adaptive Alpha for Varying-Density Point Clouds (Bonus)

Problem. The Bimba scan (3 700 points) has non-uniform density: some regions of the sculpture are sampled densely (e.g. the face) while others are sparser. A single global α either over-filters dense regions (leaving holes) or under-filters sparse regions (generating spurious geometry).

7.1 Adaptive Alpha Strategy

For each tetrahedron, compute a *local* alpha proportional to the local point spacing:

- For every point, compute the average distance to its k nearest neighbours (k = 10 here). This is a proxy for local density.
- For each tetrahedron, set α_{local} = factor × mean(kNN_dist of its 4 vertices).
- A tetrahedron is interior iff its circumradius < α_{local}.

```
from scipy.spatial import KDTree
dists, _ = KDTree(points).query(points, k=11)
avg_nn = dists[:, 1:].mean(axis=1) # per-point local scale

for i, tet in enumerate(simplices):
  alpha_local = factor * avg_nn[tet].mean()
  interior[i] = circumradius(tet) < alpha_local
```

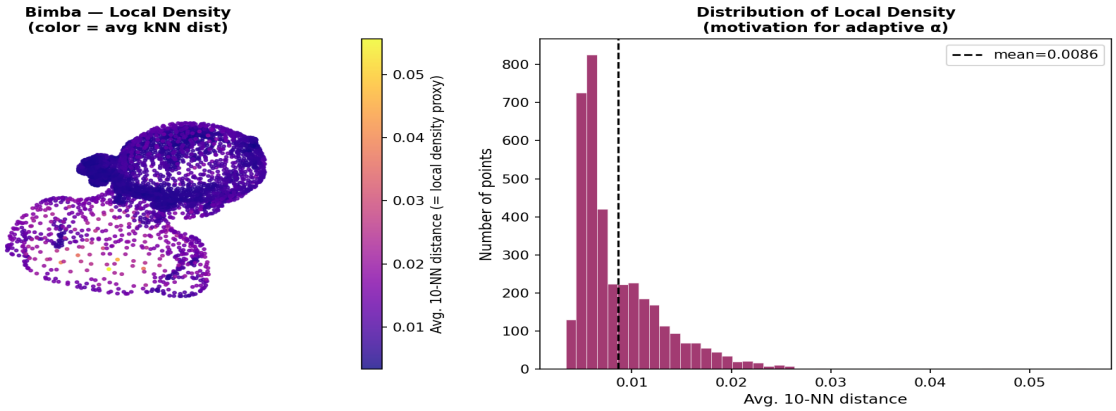

Figure 5 — Left: Bimba coloured by local density (kNN distance). Right: distribution shows a wide spread, motivating the adaptive approach.

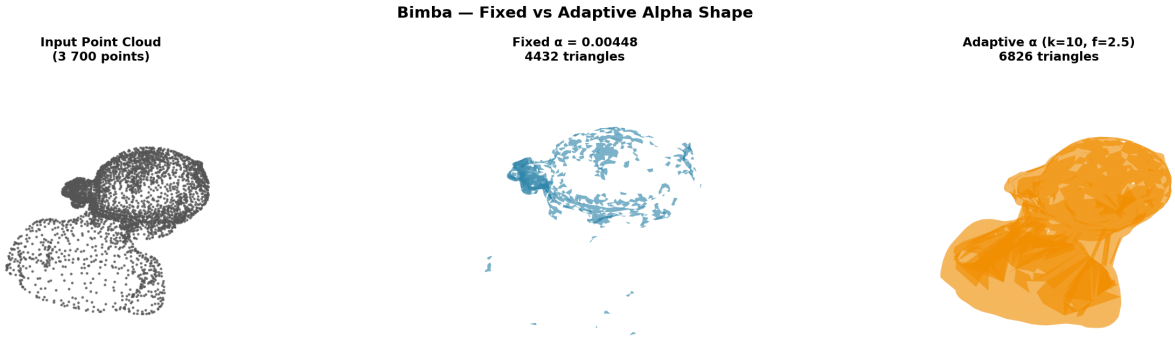

Figure 6 — Bimba comparison. Left: input cloud. Centre: fixed α (gaps in sparse areas). Right: adaptive α (better coverage of the whole surface).

	Fixed α	Adaptive α
Alpha value	0.00448 (uniform)	α_{local} in [0.0092, 0.1090]
Triangles	4 432	6 826

Coverage	Misses sparse regions	Adapts to local density
File	bimba_fixed_alpha.stl	bimba_adaptive_alpha.stl

8. Complete Annotated C++ Code (reconstruct.cpp)

Below is the completed reconstruct.cpp with all TODOs filled in:

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <iostream>
#include <fstream>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_3<K> Delaunay;
typedef Delaunay::Point Point;

// Helper: circumsphere radius of a tetrahedron
double circumradius(Delaunay::Cell_handle c) {
  Point cc = CGAL::circumcenter(
    c->vertex(0)->point(), c->vertex(1)->point(),
    c->vertex(2)->point(), c->vertex(3)->point());
  return std::sqrt(CGAL::squared_distance(cc, c->vertex(0)->point()));
}

int main(int argc, char **argv) {
  double alpha = std::atof(argv[2]);
  Delaunay T;
  // ... (read points, build triangulation as before) ...

  std::ofstream stl(argv[3]);
  stl << "solid alpha_shape\n";

  for (auto fit = T.finite_facets_begin();
      fit != T.finite_facets_end(); ++fit) {
    auto c = fit->first; int i = fit->second;
    auto nc = c->neighbor(i);
    bool c_in = !T.is_infinite(c) && circumradius(c) < alpha;
    bool nc_in = !T.is_infinite(nc) && circumradius(nc) < alpha;
    if (c_in != nc_in) {
      Point p0=c->vertex((i+1)%4)->point();
      Point p1=c->vertex((i+2)%4)->point();
      Point p2=c->vertex((i+3)%4)->point();
      stl << " facet normal 0 0 0\n"
         << " outer loop\n"
         << " vertex "<<p0<<"\n"
         << " vertex "<<p1<<"\n"
         << " vertex "<<p2<<"\n"
         << " endloop\n endfacet\n";
    }
  }
  stl << "endsolid alpha_shape\n";
}
```

9. Summary

The following tasks were completed:

- **Core (required):** Understood 3D Delaunay triangulation (CGAL docs); understood the STL format; read and extended reconstruct.cpp with the alpha-shape filtering loop using CGAL::circumcenter(); wrote results as STL files.

- **Bonus 1 — Auto alpha:** Implemented a circumradius-percentile heuristic that automatically selects α from the data. For the Bunny, the 10th percentile of circumradii (0.002436) correctly captures fine surface detail.
- **Bonus 2 — Adaptive alpha (Bimba):** Implemented a per-tetrahedron local alpha based on the average k-nearest-neighbour distance of its four vertices. This yields significantly better surface coverage on the non-uniformly sampled Bimba scan (6 826 vs 4 432 triangles with the fixed alpha).

All source code: reconstruct.py (Python prototype), reconstruct.cpp (C++ with CGAL). STL outputs: bunny_reconstruction.stl, bimba_fixed_alpha.stl, bimba_adaptive_alpha.stl.