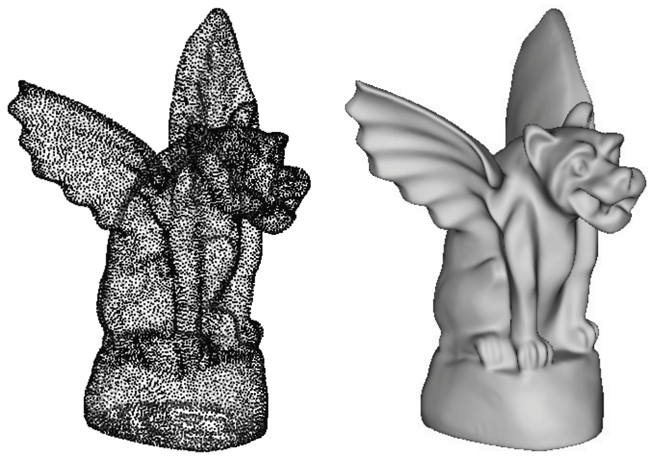
Projects for

Chapter 8: Domain-Modeling Techniques

1 PROJECT 1

Consider an unoriented 3D point cloud $P = \{\mathbf{p}_i \in \mathbb{R}^3\}$. Implement the *local triangulation* method used to reconstruct a triangle mesh passing through the points $\mathbf{p}_i$, described in Chapter 8, Section 8.3.2. For additional insight, see the reference Clarenz *et al.* mentioned in the text.


Left: 3D point cloud. Right: Surface reconstruction using local triangulations.

For this, proceed as follows:

1. To find all neighbor-points $\mathbf{p}^i_j$ of a given point $\mathbf{p}_i$ within a given radius ρ, use the fast spatial-search structures provided by the Approximate Nearest Neighbor (ANN) library.

The use of this library is demonstrated in the sample code for Chapters 5 and 7.

2. To estimate the tangent plane to a 3D point cloud, use Principal Component Analysis (PCA). This use of PCA is demonstrated in the sample *surface classification* provided for Chapter 7.

3. Compute the Delaunay triangulation of a 2D point cloud (obtained by projecting the neighbors $\mathbf{p}_j^i$ on the tangent plane computed above), use the Triangle library. The use of this library is demonstrated in the *Delaunay triangulation* sample for Chapter 3.

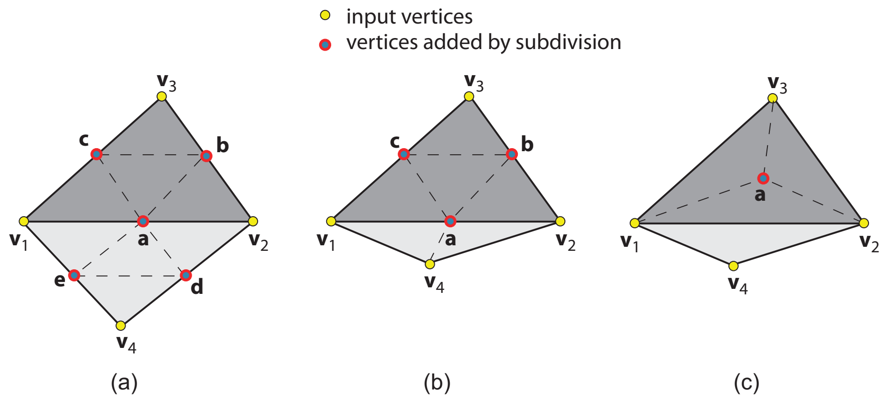
4. Construct the final 3D mesh by reprojecting the connectivity information, computed by the local Delaunay triangulations, onto the 3D points. Take care not to add the same triangle, *i.e.* using the same 3D vertices, twice.

Test and demonstrate your method by loading various meshes (*e.g.* in PLY format) and displaying side-by-side the loaded mesh and the surface reconstruction obtained from the mesh *vertices* using your method.

2 PROJECT 2

Implement a simple and computationally efficient 3D mesh refinement variation of the *Loop subdivision* algorithm (Chapter 8, Section 8.4.3). In contrast to classical Loop subdivision, however, your algorithm should incorporate a triangle size (area) criterion: In order to limit the generation of unnecessarily small triangles, no triangles smaller than a user-supplied triangle-size σ should be split, unless this is necessary to avoid the creation of T-vertices. The algorithm should have a single user-controlled parameter, being the above-mentioned triangle size σ. Test your implementation on a number of triangle meshes ranging from a few hundred to a few tens of thousands of triangles.

Hints: Consider the figure below, where we want to subdivide the triangle $\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$ (dark gray). In the classical Loop case (a), the neighbor triangle $\mathbf{v}_1\mathbf{v}_2\mathbf{v}_4$ is also split into four triangles, even though it may be very small. Alternative ways to split $\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$ are sketched in images (b) and (c), where we either split the small neighboring triangle differently (image b), or avoid splitting it at all by subdividing $\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$ using its barycenter $\mathbf{a}$ (image c).



A variation of the Loop subdivision algorithm.

3 PROJECT 3

Given a uniform 3D grid, *e.g.* consisting of cubic cells (voxels) that encode a 3D scalar field, an isosurface of this field extracted by the classical marching cubes method (Chapter 5, Section 5.3.2) will contain triangles having roughly the same size. Using this observation, imagine and describe a method that, given an unstructured 3D triangle mesh representing a closed surface, would produce a refinement or simplification of this mesh where all triangles have roughly the same size σ, where σ is a user-controlled parameter. Proceed as follows:

1. As input, your code should accept a 3D triangle mesh (having possibly triangles of highly different sizes) which describes an orientable closed (watertight) surface.

2. Consider the 3D distance function, or distance transform, of this meshed surface (see Chapter 9, Section 9.4.5). To compute this distance transform, first densely sample the triangles of the input mesh with additional points, so that no location in a triangle is at a larger distance from a sample-point or input-mesh point than a (small) user-given value. Next, use the 3D distance transform GPU code of Cao *et al.* or, alternatively, the CPU distance transform code of Roerdink *et al.* Both codes are provided in the samples for Chapter 9. This allows you to compute the 3D distance transform of your dense point cloud. The resolution of the 3D voxel volume used to compute and store the distance transform should be proportional with your desired degree of mesh refinement or simplification σ.

3. Finally, extract an isosurface from this volume (at the value 0) using marching cubes, to visualize your refined or simplified mesh.

Test your implementation by visualizing the original 3D mesh and several refinements and simplifications thereof using different $sigma$ values. For a better verification, use a highly non-uniformly sampled mesh.

4 PROJECT 4

Consider a 2D or 3D point cloud $C = \{\mathbf{p}_i \in R \subset \mathbb{R}^{n\in\{2,3\}}\}$. We assume that this cloud is highly non-uniformly sampled, *i.e.*, some areas show high point densities, where other areas have low densities. Such point clouds can become very large, *e.g.* tens of millions of points, which can create performance problems for further processing and rendering.

Design and implement a simple method for subsampling the dataset C. The method should have a single user-provided parameter ρ, which gives the *maximal* point density allowed in the subsampled dataset, expressed as the minimal allowed distance between any two sample points in C. Given C and ρ, construct a new point cloud C' that

- is a subset of C, *i.e.* $C' \subset C$;
- is as large as possible, *i.e.* eliminates only those points in C that violate the maximal density condition.
- obeys the maximal density imposed by the parameter ρ;
- can be computed efficiently. Your implementation should be able to subsample large clouds C, *e.g.* millions of 3D points, in a matter of seconds on a modern PC computer.

Test your method by *e.g.* loading various 3D meshes (in PLY format), constructing C by considering their vertices, and displaying the subsampled cloud C' side-by-side with the original cloud C, for various values of ρ.

Hints: Consider using the Approximate Nearest Neighbors (ANN) library to efficiently locate all neighbors of a 3D point within a given radius ρ. ANN is demonstrated in the code samples in Chapter 5. Iterate through the points in C and, for each point, erase all its neighbors within the radius ρ. The resulting cloud C' will consist in all non-erased points in C.

5 PROJECT 5

Consider a 2D point cloud $C = \{(\mathbf{p}_i \in R \subset \mathbb{R}^2, v_i \in \mathbb{R})\}$ where each point $\mathbf{p}_i$ has a scalar value v_i. We assume that C samples a given 2D spatial region R which has a simple shape, such as a rectangle. One way to construct a *gridless* interpolation $\tilde{v}$ of this scalar dataset over R is by using radial basis functions is to use the Shepard method, as described in Chapter 3, Section 3.9.2. However, a main problem of this method is that it does not guarantee that the reconstructed scalar signal $\tilde{v}$ passes precisely through the sample values at the sample points, *i.e.* that $\tilde{v}(\mathbf{p}_i) = v_i, \forall i$, the so-called 'partition of unity' problem of radial basis functions.

Implement an alternative interpolation method that guarantees both conditions below

- the reconstructed signal $\tilde{v}$ is smooth, of $\mathscr{C}^\infty$ continuity;
- the reconstructed signal truly interpolates the sample points, *i.e.* $\tilde{v}(\mathbf{p}_i) = v_i, \forall i$.

For this, proceed as follows:

1. Construct an uniform, dense, sampling $S = \{\mathbf{q}_i \in R\}$ of the 2D spatial domain R. The cloud points $\mathbf{p}_i$ should be *all* part of this sampling, *i.e.*, $C \subset S$. For this, use the area-constrained Delaunay triangulation code provided in the samples for Chapter 3.

2. Use S to construct an unstructured 2D grid G over R with triangle cells.

3. Create a scalar dataset $D = \{(\mathbf{q}_i, w_i \in \mathbf{R})\}$ over the above grid G. For all $\mathbf{q}_i \in C$, set $w_i = v_i$. For all other $\mathbf{q}_i \notin C$, set $w_i = 0$.

4. Apply a Laplacian smoothing operation to the values w_i over G, but only for the points $\mathbf{q}_i \notin C$. For this, study the *Laplacian smoothing* sample provided in Chapter 8. This will 'spread' the non-zero sample values over their neighbor points in G, but it will *not* modify the values w_i at the points $\mathbf{q}_i \in C$.

5. Repeat step 4 for a number of times, until the signal $\tilde{v} = \{w_i\}$ becomes as smooth as desired.

6. Visualize the resulting signal $\tilde{v}$ by *e.g.* colormapping or a height plot. For this, study the respective code samples provided in Chapter 6.

End of Projects for Chapter 8: Domain-Modeling Techniques