Surface Representation & Mesh Data Structures

Geometry Processing

Geometry processing is a field of computer science that deals with the manipulation and analysis of geometric objects, such as points, curves, surfaces, and volumes. The goal of geometry processing is to develop efficient algorithms and data structures for representing, computing, and manipulating geometric data in a variety of applications, including computer graphics, robotics, virtual reality, and engineering.

Geometry processing encompasses a wide range of techniques, including geometric modeling, mesh generation and optimization, **surface** reconstruction, shape analysis and matching, and geometric data compression. Some common operations in geometry processing include geometric transformations, such as translation, rotation, and scaling, as well as geometric queries, such as intersection testing, distance computation, and surface sampling.

One of the main challenges in geometry processing is to develop algorithms that are robust, efficient, and scalable to handle large-scale geometric data sets. This requires a deep understanding of the underlying mathematical principles, as well as expertise in software engineering and optimization techniques.

Some key applications of geometry processing include computer-aided design (CAD), computer-aided manufacturing (CAM), 3D printing, medical imaging, and computer graphics.

We will focus on **Surface Representations** and **Mesh Data Structures** where Surface representations describe the geometry of 3D objects using simplified approximations, while mesh data structures provide a way to efficiently represent and manipulate the connectivity of surfaces using vertices, edges, and faces.

Preface

Surface Representation Introduction

Choosing the right mesh data structure for a given application depends on a variety of factors, including the complexity of the mesh, the desired level of detail, and the computational resources available. Each type of mesh data structure has its own advantages and disadvantages, so it is important to choose the one that best meets the specific needs of the application.

Surface representations:

• Parametric surface representations:

Vector-valued parameterization function $\boldsymbol{f}$

---> $\boldsymbol{f}: \Omega \rightarrow \mathcal{S}$ maps "$\Omega \subset \mathbb{R}^2$" to $\mathcal{S} = \boldsymbol{f}(\Omega) \subset \mathbb{R}^3$

//Range of a function, surface patch

• Implicit (Volumetric) surface representations:

Zero set of a scalar-valued function $\boldsymbol{F}$

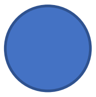
---> $\boldsymbol{F}: \mathbb{R}^3 \rightarrow \mathbb{R}$ which is $\mathcal{S} = \{\boldsymbol{x} \in \mathbb{R}^3 | \boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}\}$

//kernel of a function, level set

Defining curves: A simple 2D example --"Unit Circle"--

• Parametric:

$$\boldsymbol{f}: [0,2\pi] \mapsto \mathbb{R}^2,\ t \mapsto \begin{pmatrix} \cos t \\ \sin t \end{pmatrix}$$



• Implicit:

$$\boldsymbol{F}: \mathbb{R}^2 \rightarrow \mathbb{R},\ (x,y) \mapsto \sqrt{x^2+y^2} - 1$$

The two primary classes of surface representations highlighted are parametric and implicit representations. Parametric surfaces are defined by a function that maps a 2D parameter domain to the surface, while implicit surfaces are defined as the zero set of a scalar-valued function. An example of parametric representation is a function defining a curve, and the implicit equivalent would be for planar curves only.

For more complex shapes(like how Bezier surfaces or B-spline surfaces are used in computer-aided geometric design (CAGD)), an explicit formulation may not be feasible, leading to a piecewise definition where the function domain is split into smaller sub-regions, with an individual function defined for each segment. This allows the function to approximate the given shape locally, with the global approximation tolerance controlled by the size and number of segments. The challenge here is ensuring a consistent transition between patches (referred to a portion of a surface or an object).

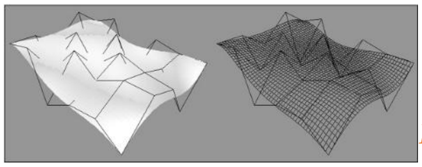
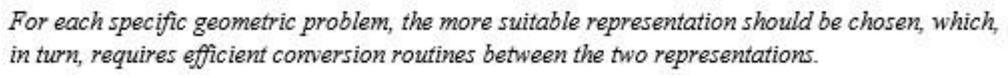


Figure. Bezier Surface

The strengths and weaknesses of parametric and implicit representations are complementary. Therefore, it is necessary to choose the most suitable representation for each geometric problem. We classify the operations performed on these representations into three categories:

1. Evaluation: Sampling of the surface geometry or other surface attributes, such as the surface normal field, commonly used in surface rendering.
2. Query: Determining the position of a point relative to a surface or calculating the distance to the surface.
3. Modification: Changing the surface either in terms of geometry (surface deformation) or topology (merging, cutting, or deleting parts of the surface).



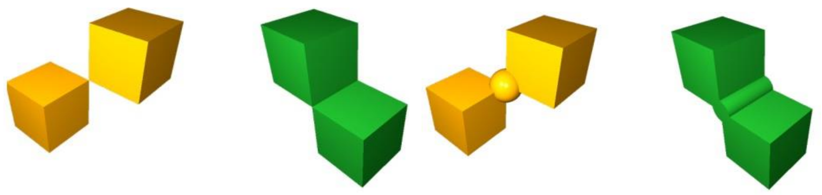
For each specific geometric problem, the more suitable representation should be chosen, which, in turn, requires efficient conversion routines between the two representations.

Surface Definition and Properties

Manifold: A geometric entity that is locally like a Euclidean space. Ex.: each point has a neighborhood resembling a 2D flat surface.
Non-degenerate: In the context of 3D solids, it refers to objects without any features that are infinitely thin, ensuring clear separation of interior and exterior.

In computer graphics, *"an orientable continuous 2D manifold embedded in* $\mathbb{R}^3$*"*, a surface is often described as 2D shape embedded in 3D space. Intuitively, this can be understood as the boundary surface of a non-degenerate 3D solid, it represents the boundary of a 3D object without infinitely thin features. If it has boundaries, they can be filled to make a manifold surface, which mean that if we have a surface that almost behaves like a manifold but has a few gaps, we can "repair" it by filling in those gaps, making it a proper manifold surface.

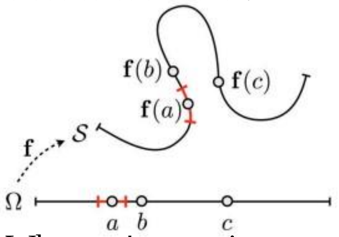
An orientable continuous 2-manifold describes the surface of a non-degenerate solid. A degenerate/non-manifold vertex and a solid with a degenerate/non-manifold edge (left yellow and green), Those solids are fixed in the right part.



Information about these surfaces is typically obtained through discrete sampling. This raw data needs continuity, **to ensure a smooth and continuous surface,** a neighborhood relation between these samples is established. Which can be achieved by creating consistent neighborhood relationships between samples, establishing what's called a 'geodesic neighborhood relation'. This means two points on the surface are 'geographically' close if their corresponding pre-images in the parameter domain are close. (*Here, Pre-images are original data points in the parameter domain that correspond to points on the surface in the spatial domain*.).

For each point on the surface, there's a 'local manifold' condition: if we draw a tiny sphere around a surface point, the surface patch within it should resemble a disk. This holds for both parametric and implicit surface representations. In many applications, the initial data about the surface we want to model comes from discrete samples, whether that's by assessing a pre-existing digital model or measuring a physical object. The first step in creating a mathematical representation of this surface is to ensure continuity, or seamless connection, among these samples. This is done by forming a consistent system of neighborhoods among the samples, which infers the existence of a continuous surface, or manifold, that the samples originate from.

This neighborhood relationship is determined by 'geodesic' closeness, which refers to closeness following the surface's contours, instead of just physical distance. This can be harder to deduce from implicit surface representations but is straightforward in parametric representations. In parametric models, two points are considered geodesically close if their corresponding initial points in the parameter domain, or pre-images, are close to each other.



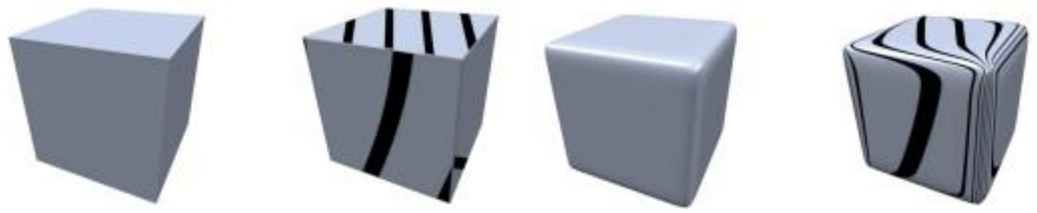
A manifold curve. While the points f(a), f(b), and f(c) are all in close spatial proximity, only f(a) and f(b) are geodesic neighbors since their pre-images a and b are neighbors, too. In red: The pre-image of a sufficiently small δ neighborhood around f(a) in $\mathbb{R}^2$ lies in an ε neighborhood of a in $\mathbb{R}$.

When we're creating a continuous surface using a set of separate data points, we have two main options. We can make the surface pass exactly through each data point, which is known as interpolation. Alternatively, we can allow the surface to be an approximation that fits closely to the data points within a certain allowed deviation. This second option is usually preferred in real-world applications, as data points often have some noise or errors and the true surface that exists between these points is likely to be an approximation itself.

Smoothness

Generally, we want our surfaces to be smooth, except at certain well-defined features like sharp curves and corners. Notice that this analytical definition of C^k smoothness coincides with the intuitive geometrical understanding of smoothness only if the partial derivatives of f or the gradient of F, respectively, do not vanish locally (regularity).

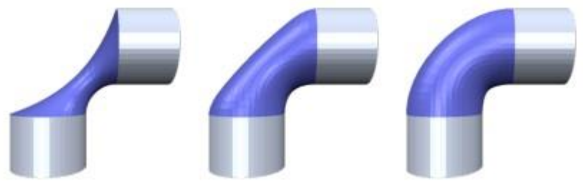
- position continuity : C^0
- tangent continuity : C^1
- curvature continuity : C^2



Fairness

Fairness in surfaces goes beyond just having continuous derivatives; it also takes into account the magnitudes and variations of those derivatives. While there isn't a precise formal definition for the aesthetic concept of fairness, a surface is typically regarded as fair if measures like curvature or its variation are minimized on a global scale.

- minimum surface area
- minimum curvature
- minimum curvature variation



Three examples of fair surfaces, which define a blend between two cylinders: a membrane surface that minimizes the surface area (left), a thin-plate surface that minimizes total curvature (center), and a surface that minimizes the variation of mean curvature (right).

Approximation Power

Mathematical modeling of real objects or boundaries is often complex, so digital surface representations are approximations. To simplify the approximation, the domain is divided into small segments, and a function (patch) is defined for each segment to locally approximate the input.

Polynomials are commonly chosen for surface representations due to their efficient processing and the Weierstrass theorem, which guarantees that smooth functions can be approximated by polynomials with any desired precision, approximation accuracy can be improved by increasing the degree of the polynomial (p-refinement) or reducing the segment size (h-refinement). In geometry processing, h-refinement is preferred over p-refinement due to limitations in assuming bounded higher-order derivatives and the difficulty in satisfying smoothness conditions between segments. From calculus we know that a C^∞ function g with bounded derivatives can be approximated over an interval of length h by a polynomial of degree p such that the approximation error behaves like $O(h^{p+1})$ (e.g., Taylor's theorem or generalized mean value theorem). Consequently, there are, in principle, two possibilities to improve the accuracy of an ap-proximation with piecewise polynomials.

We can either raise the degree of the polynomial (p-refinement) or we can reduce the size of the individual segments and use more segments for the approximation (h-refinement). In geometry processing applications, h-refinement is usually preferred over p-refinement since, for a discretely sampled input surface, we can- not make reasonable assumptions about the boundedness of higher-order derivatives.

Moreover, for piecewise polynomials with higher degree, the C^k smoothness conditions between segments are sometimes quite difficult to satisfy. Finally, with today's computer architectures, processing many very simple objects is often much more efficient than processing a smaller number of more complex ones. This is why the extremal choice of C^0 piecewise linear surface representations, i.e., polygonal meshes, have become the widely established standard in geometry processing.

While, for parametric surfaces, the $O(h^{p+1})$ approximation error estimate follows from the mean value theorem in a straightforward manner, a more careful consideration is necessary for implicit representations. The generalized mean value theorem states that if a sufficiently smooth function g over an interval [a, a + h] is interpolated at the abscissae $t_0, \ldots, t_p$ by a polynomial f of degree p, then the approximation error is bounded by:

$$|f(t) - g(t)| \;\leq\; \frac{1}{(p+1)!} \max f^{(p+1)} \prod_{i=0}^{p} (t_i - t) \;=\; O(h^{p+1}).$$

For an implicit representation G: IR → IR and the corresponding polynomial approximant F , this theorem is still valid; however, here the actual surface geometry is not defined by the function values G(x), for which this theorem gives an error estimate, but by the zero-level set of G, i.e., by: $S = \{x \in \mathbb{R}^3 \mid G(x) = 0\}$.

Consider a point x on the implicit surface defined by the approximating polynomial F , i.e., F (x) = 0 within some voxel. We can find a corresponding point x + d on the implicit surface defined by G, ex: G(x + d) = 0 by shooting a ray in normal direction to F , i.e., d = d ∇F/||∇F||. For a sufficiently small voxel size h, we obtain:

$|F(\mathbf{x}+\mathbf{d})| \;\approx\; |d| \, \|\nabla F(\mathbf{x})\| \quad \Rightarrow \quad |d| \;\approx\; \frac{|F(\mathbf{x}+\mathbf{d})|}{\|\nabla F(\mathbf{x})\|}$ from the mean value theorem, we get: $|F(\mathbf{x}+\mathbf{d}) - G(\mathbf{x}+\mathbf{d})| \;=\; |F(\mathbf{x}+\mathbf{d})| \;=\; O(h^{p+1})$

which yields $|d| = O(h^{p+1})$ if the magnitude of the gradient ||∇F|| is bounded from below by some ε > 0. In practice one tries to find an approximating polynomial F with low variation of the gradient magnitude in order to have a uniform distribution of the approximation error.

In summary, the combination of the generalized mean value theorem, ray shooting, and gradient control is used to estimate and minimize the approximation error in implicit surface representations.

Parametric Surface Representations

Parametric surfaces simplify 3D problems by reducing them to 2D issues within a parameter domain. For example, generating sample points or identifying geodesic neighborhoods on a surface becomes easier. However, creating a parametric surface can be complex due to the need to align the parameter domain with the surface's structure. Topological modifications to the surface may also require adjustments to the parameter domain, which can be challenging. Additionally, certain operations like inside/outside spatial queries or detecting self-collisions can be computationally expensive on parametric surfaces, for this reason topological modification and spatial queries are the weak points of parametric surfaces.

Tensor product spline surface of bi-degree n

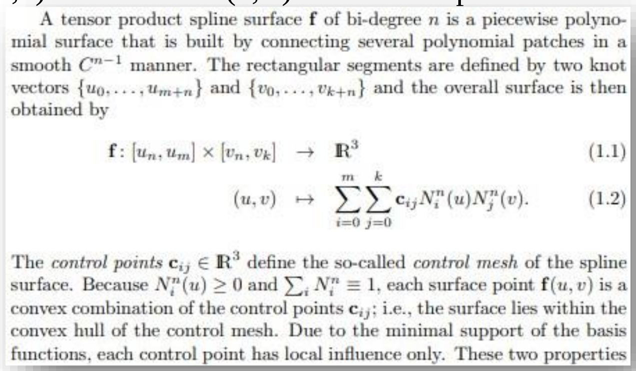
A tensor product spline surface of bi-degree n is a 2D surface expressed as a grid of n-degree spline curves. Imagine a quilt, each patch is a smooth curve, seamlessly joined to form a larger surface.

This surface is mathematically represented by two sets of parameters (u,v). The surface f(u, v) is the tensor product of B-spline basis functions and control points:

f(u,v) = ∑ ∑ Ni,n(u) * Mj,n(v) * Pij

Where:

- f(u,v) is a point on the surface,
- Ni,n(u) and Mj,n(v) are B-spline basis functions,
- Pij are the control points.



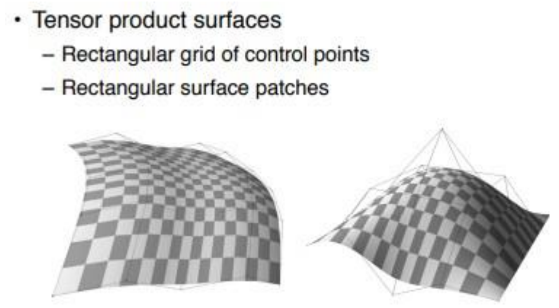
A tensor product spline surface **f** of bi-degree n is a piecewise polynomial surface that is built by connecting several polynomial patches in a smooth C^{n-1} manner. The rectangular segments are defined by two knot vectors $\{u_0, \ldots, u_{m+n}\}$ and $\{v_0, \ldots, v_{k+n}\}$ and the overall surface is then obtained by

$$\mathbf{f}\colon [u_n, u_m] \times [v_n, v_k] \to \mathbb{R}^3 \quad (1.1)$$

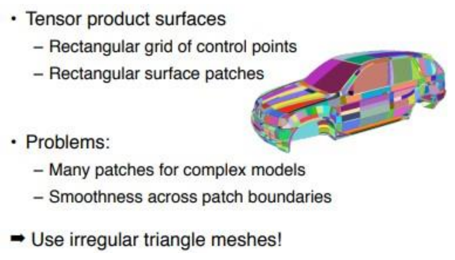
$$(u,v) \mapsto \sum_{i=0}^{m} \sum_{j=0}^{k} \mathbf{c}_{ij} N_i^n(u) N_j^n(v). \quad (1.2)$$

The *control points* $\mathbf{c}_{ij} \in \mathbb{R}^3$ define the so-called *control mesh* of the spline surface. Because $N_i^n(u) \geq 0$ and $\sum_i N_i^n \equiv 1$, each surface point $\mathbf{f}(u,v)$ is a convex combination of the control points $\mathbf{c}_{ij}$; i.e., the surface lies within the convex hull of the control mesh. Due to the minimal support of the basis functions, each control point has local influence only. These two properties

To understand, observe the image and consider: "The grid's control points dictate the surface, which is smoothly interpolated between these points."



- Tensor product surfaces
  - Rectangular grid of control points
  - Rectangular surface patches

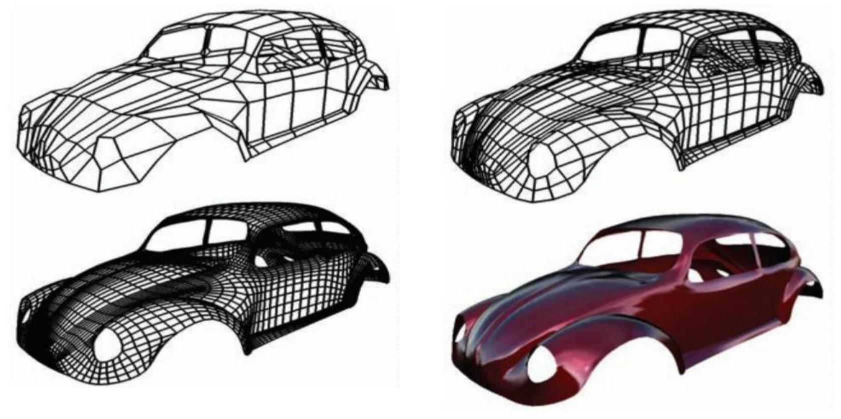


- Tensor product surfaces
  - Rectangular grid of control points
  - Rectangular surface patches
- Problems:
  - Many patches for complex models
  - Smoothness across patch boundaries

➡ Use irregular triangle meshes!

Subdivision Surfaces

Subdivision surfaces can be considered a generalization of spline surfaces since they are also controlled by a coarse control mesh, but in contrast to spline surfaces, they can represent surfaces of arbitrary topology. Subdivision surfaces are generated by repeated refinement of control meshes: *after each topological refinement step, the positions of the (old and new) vertices are adjusted based on a set of local averaging rules,*



Subdivision surfaces are generated by iterative uniform refinement of a coarse control mesh.

Triangle Meshes

Triangle Meshes and Barycentric Parameterization

Triangle meshes are widely used in geometry processing. They are essentially a collection of triangles where each triangle, using barycentric coordinates, defines a part of a piecewise linear surface representation. This means that each point inside a triangle can be uniquely represented as a weighted sum of the triangle's vertices.

Every point inside a triangle can be uniquely represented by a combination of the corner points of the triangle. This is the essence of barycentric coordinates.

In the equation, $p = \alpha a + \beta b + \gamma c$, 'p' is a point inside the triangle, 'a', 'b', and 'c' are the corners of the triangle, and 'α', 'β', and 'γ' are the barycentric coordinates.

The barycentric coordinates represent the weights or contributions of the corners to the point 'p'. These weights must always add up to 1, and they should be greater than or equal to 0. This ensures that the point 'p' lies inside the triangle.

Linear Mapping and Global Parameterization

By selecting an arbitrary triangle in the parameter domain, a linear mapping from the 2D domain to 3D space can be established. This mapping provides the base for defining a 2D position for each vertex, which is enough to derive a global parameterization for the entire mesh. This global parameterization is important because it enables the entire mesh to be understood and manipulated in the context of this 2D-3D mapping.

By choosing a triangle in the 2D parameter domain, we can define a mapping from this 2D triangle to the 3D triangle. This mapping is a direct transfer of the barycentric weights.

In the mapping equation, $\alpha u + \beta v + \gamma w 7\!\!\longrightarrow \alpha a + \beta b + \gamma c$, 'u', 'v', and 'w' are points in the 2D domain and 'a', 'b', and 'c' are points in the 3D domain. 'α', 'β', and 'γ' are the same barycentric coordinates in both domains.

This mapping is useful because once we define the 2D positions for each vertex of the mesh, we can derive the 3D positions for the entire mesh, which is a global parameterization.

Triangle Mesh Components

A triangle mesh is made up of a geometric part and a topological part.

The topological part is like the blueprint of the mesh, represented by a graph structure. In this graph, the vertices are the points in space and the faces are the triangles formed by these points.

In some cases, it can be more efficient to represent the mesh connectivity in terms of the edges connecting the vertices instead of the triangular faces.

Geometric Embedding

The geometric part of the mesh is its placement in 3D space. Each vertex from the topological graph is given a specific 3D position. These positions are denoted by 'P' and are in the form of (x, y, z) coordinates. These 3D positions define the actual shape of the mesh in space.

Approximation and Error

Triangle meshes are used to approximate smooth surfaces. The approximation error, which is the difference between the actual surface and the approximated surface, is inversely proportional to the number of faces in the mesh. Hence, the more triangles, the better the approximation.

2-Manifold and Non-Manifold

A triangle mesh is a 2-manifold if it is locally like a disc around every point. Non-manifold edges and vertices break this property and can cause problems in algorithms.

Euler's Formula and Mesh Statistics

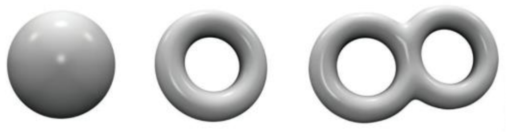
Euler's formula states a relationship between the numbers of vertices (V), edges (E), and faces (F) in a mesh, where g is the genus of the surface. Based on this, there are some important mesh statistics that can be derived:

$V - E + F = 2(1 - g)$

- The number of triangles is approximately twice the number of vertices: $F \approx 2V$
- The number of edges is approximately three times the number of vertices: $E \approx 3V$
- The average vertex valence (number of edges incident to a vertex) is 6.

'g' represents the genus of the surface. The genus of a surface is a measure of its complexity, and it's essentially the number of "holes" or "handles" it has. For instance:

A sphere, which has no holes, has a genus of 0.



A torus (doughnut shape) has one hole, and therefore a genus of 1.

A double torus (two doughnuts stuck together) has two holes, and so has a genus of 2.

These statistics are useful when estimating the complexity of mesh processing algorithms and analyzing data structures or file formats for triangle meshes.

Implicit Surface Representations

An implicit surface representation uses a function to assign each point in a three-dimensional space a value. This value signifies whether the point is inside, outside, or on the boundary surface (S) of a geometric model.

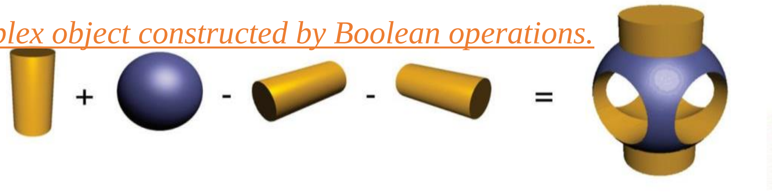
Several different types of implicit functions can be used for such representation, including algebraic surfaces, radial basis functions, and discrete voxelizations. In each case, the surface S of the object is defined as the zero-level isosurface(*level set of a continuous function whose domain is 3-space.*) of the scalar-valued function F. This means that points with a function value of zero are on the surface, those with a positive value are outside the object, and those with a negative value are inside the object.

The primary advantages of this kind of representation are its continuity (no holes in the surface) and its robustness against geometric self-intersections. This robustness makes implicit representations useful for applications like mesh repair.

Another advantage is the simplicity with which one can answer geometric inside/outside queries. Since these are resolved by simply evaluating the function at a point and checking the sign of the result, implicit representations are ideal for use in constructive solid geometry (CSG). This is a method for constructing complex objects by applying Boolean operations (union, intersection, and difference) to simpler "primitive" geometric shapes.

Despite these advantages, implicit surface representations have certain limitations. These include the difficulty of generating sample points on an implicit surface, finding geodesic neighborhoods, and rendering the surface. Also, implicit surfaces do not provide any form of parameterization, which can complicate the task of consistently applying textures to evolving surfaces.

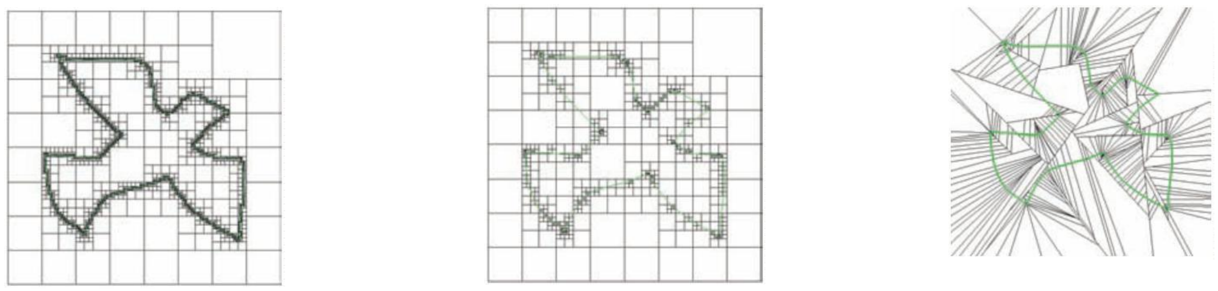
A complex object constructed by Boolean operations.



Regular Grids

In order to efficiently process implicit representations, the continuous scalar field F is typically discretized in some bounding box around the object using a sufficiently dense grid with nodes $g_{igk} \in \mathbb{R}^3$. The most basic representation, therefore, is a uniform scalar grid of sampled values $F_{ijk} := F(g_{igk})$, and function values within voxels are derived by trilinear interpolation, thus providing quadratic approximation order. However, the memory consumption of this naive data structure grows cubically if the precision is increased by reducing the edge length of grid voxels.

Adaptive Data Structures



Different adaptive approximations of a signed distance field with the same accuracy: three-color quadtree (left, 12040 cells), adaptively sampled distance fields (ADF) (center, 895 cells), and binary space partitioning (BSP) (right, 254 cells).

Conversion Methods

In order to exploit the specific advantages of parametric and implicit surface representations, efficient conversion methods between the two are necessary. However, notice that both kinds of representations are usually finite samplings (e.g., triangle meshes in the parametric case, uniform/adaptive grids in the implicit case) and that each conversion corresponds to a resampling step. Hence, special care has to be taken in order to minimize loss of information during these conversion routines.

Parametric to Implicit

The conversion of a parametric surface representation to an implicit one amounts to the computation or approximation of its signed distance field.

This can be done very efficiently by voxelization or 3D scan-conversion techniques, but the resulting approximation is piecewise constant only. As a surface's distance field is, in general, not smooth everywhere, a piecewise linear or piecewise trilinear approximation seems to be the best compromise between approximation accuracy and computational efficiency.

Since we focus on polygonal meshes as parametric representation in this book, the conversion to an implicit representation requires the computation of signed distances to the triangle mesh at the nodes of a (Uniform or adaptive) 3D grid.

Computing the exact distance of a grid node to a given mesh requires to calculate the distance to the closest triangle, which can be found efficiently by using spatial data structures, e.g., kd-trees. Notice that, in order to compute a signed distance field, one additionally has to determine whether a grid node lies inside or outside the object. If g denotes the grid node and c its closest point on the surface, then the sign can be derived from the angle between the vector $g - c$ and the outer normal n(c): The point g is defined to be inside if $(g-c)^T n(c) < 0$. The robustness and reliability of this test strongly depends on the way the normal n(c) is computed. Using angle-weighted pseudo-normal for faces, edges, and vertices can be shown to yield correct results.

Computing the distances on the entire grid can be accelerated by fast marching methods. In a first step, the exact signed distance values are computed for all grid nodes in the immediate vicinity of the triangle mesh.

After this initialization, the fast-marching method propagates distances to the remaining grid nodes with unknown distance value in a breadth-first manner. Let's divide them in step for a better understanding, we are describing the process of converting a parametric surface representation to an implicit one. In this case, it specifically focuses on the conversion from a polygonal mesh to an implicit representation.

1. **Step 1: Compute the Signed Distance Field** The first step in this conversion involves the computation or approximation of the surface's signed distance field. This could be done efficiently using voxelization or 3D scan-conversion techniques. However, these result in a piecewise constant approximation. Since the distance field of a surface is not smooth everywhere, a piecewise linear or piecewise trilinear approximation is often a good compromise between approximation accuracy and computational efficiency.
2. **Step 2: Compute Signed Distances to the Triangle Mesh** The conversion process then requires the calculation of signed distances to the triangle mesh at the nodes of a 3D grid (which can be either uniform or adaptive). The exact distance from a grid node to the mesh involves calculating the distance to the closest triangle. This can be done efficiently using spatial data structures such as kd-trees.
3. **Step 3: Determine Inside or Outside Nodes** In order to compute a signed distance field, it's also necessary to determine whether a grid node lies inside or outside the object. This can be determined by

looking at the angle between the vector from the closest point on the surface to the grid node, and the outer normal at the closest point. If the dot product of this vector with the outer normal is negative, the point is inside the object. The reliability of this test is influenced by how the normal is computed. The text suggests using angle-weighted pseudo-normal for faces, edges, and vertices for best results.

4. **Step 4: Fast Marching Methods** Last step involves the use of fast marching methods to accelerate the computation of distances on the entire grid. The method starts by calculating the exact signed distance values for all grid nodes in the immediate vicinity of the triangle mesh. After this initialization, the fast-marching method propagates distances to the remaining grid nodes with unknown distance values in a breadth-first manner.

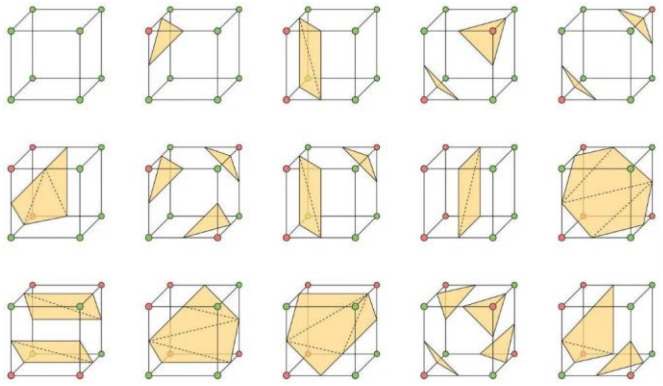
To visualize this process, we could imagine a 3D grid enveloping the mesh, and the steps listed above applied to each node on the grid to convert the parametric mesh to an implicit representation.

Implicit to Parametric

Marching Cubes: An algorithm for extracting a polygonal mesh of an isosurface from a 3D scalar field.

Isosurface extraction: The process of creating a 3D surface from a 3D scalar field based on a defined threshold value.

The conversion from an implicit or volumetric representation to a triangle mesh, the so called isosurface extraction, occurs for instance in CSG modeling and in medical applications, ex: to extract the skull surface from a CT head scan. The de-facto standard algorithm for isosurface extraction is marching cubes.



The 15 base configurations of the marching cubes triangulation table. The other 241 cases can be found by rotation, reflection, or inversion.

This grid-based method samples the implicit function on a regular grid and processes ach cell of the discrete distance field separately, thereby allowing for trivial parallelization. For each cell that is intersected by the isosurface S, a surface patch is generated based on local criteria. The collection of all these small pieces eventually yields a triangle mesh approximation of the complete isosurface S.

For each grid edge intersecting the surface S, the marching cubes algorithm computes a sample point that approximates this intersection. In terms of the scalar field F, this means that the sign of F differs at the grid edge's endpoints p1 and p2 . Since the trilinear approximation F is actually linear along the grid edges, the intersection point s can be found by linear interpolation of the distance values d1 := F(p1) and d2 := F(p2) at the edge's endpoints: $$\mathbf{s} = \frac{|d_2|}{|d_1| + |d_2|}\mathbf{p}_1 + \frac{|d_1|}{|d_1| + |d_2|}\mathbf{p}_2.$$

The resulting sample points of each cell are then connected to a triangulated surface patch based on a triangulation look-up table holding all possible configurations of edge intersections. Since the possible combinatorial configurations are determined by the signs at a cell's corners, their number, and hence the size of the table, is $2^8 = 256$.

We notice that a few cell configurations are ambiguous, which might lead to cracks in the extracted surface. A properly modified look-up table yields a simple and efficient solution, however, at the price of sacrificing the symmetry with regard to sign inversion of F. The resulting isosurfaces then are watertight 2-manifolds, which is exploited by many mesh repair techniques.

Extended Marching Cubes is an improved algorithm that preserves sharp features of a 3D model by adding extra sample points. It can work directly on adaptively refined octrees to manage the triangle complexity of the output mesh. However, it may generate non-manifold meshes that need further fixing.

Finally, an alternative to marching cubes and its variants consists of refining and filtering a 3D Delaunay triangulation. The resulting surface mesh can contain only well-shaped triangles and faithfully approximates the input surface in terms of both topology and geometry.

Summary

In this chapter, we discussed about various mathematical surface representations, namely parametric and implicit surface representations, their strengths, and their weaknesses. Parametric surfaces, while excellent at capturing detail and easy to sample and modify, struggle with distance queries and require significant restructuring for topological changes. Implicit surfaces, on the other hand, handle topological changes and distance queries with ease, but face challenges with sampling, shape editing, and their detail resolution is dependent on voxel size.

We also explored hybrid representations that merge the benefits of both parametric and implicit representations. There are many other conversion techniques for further reading. We have seen an approach which converts a polygon soup into implicit surfaces, which can range from interpolating to approximating with adjustable smoothness and tolerance. Besides the approaches described here, which are most relevant for the techniques presented in this book, there are many other representations suitable for efficient geometry processing. Radial basis functions are a prominent example, as are partition of unity implicit and point-based representations just to mention a few.

Mesh Data Structures Introduction

The efficiency and memory usage of geometric modeling algorithms heavily rely on the underlying surface mesh data structures. This chapter provides an overview of the most commonly used data structures found in literature. When choosing a mesh data structure, both topological and algorithmic considerations need to be taken into account:

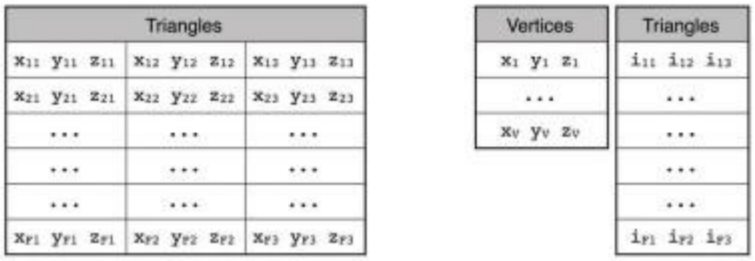
1. Topological requirements: It is important to determine the types of meshes that need to be represented by the data structure. Do we only need to handle 2-manifold meshes, or should the data structure also support complex edges and singular vertices? Is it sufficient to represent pure triangle meshes, or is there a need for arbitrary polygonal meshes? Are the meshes regular, semi-regular, or irregular? Additionally, it may be necessary to build a hierarchy of meshes with different levels of refinement.
2. Algorithmic requirements: The choice of data structure depends on the algorithms that will operate on it. Will the mesh be used solely for rendering purposes, or does efficient access to local neighborhoods of vertices, edges, and faces matter? Will the mesh be static, or will its geometry and connectivity change over time? Is there a need to associate additional data with vertices, edges, or faces? Special memory consumption requirements may also be a consideration, especially when dealing with massive data sets.

Evaluating a data structure involves measuring various criteria, including:
a) Preprocessing time required to construct the data structure,
b) Query time for answering specific queries,
c) Operation time for performing specific operations,
d) Memory consumption and redundancy.

While it may not sound incredibly exciting, the design and selection of mesh data structures play a crucial role in the efficiency and effectiveness of geometric modeling algorithms. It's common practice to create specialized data structures that cater to specific algorithms, but there are also widely used data structures that serve multiple geometry processing algorithms. In this chapter, we delve into these commonly employed data structures, examining their strengths and weaknesses to help you make informed choices. By considering the topological and algorithmic requirements and carefully evaluating data structures based on factors such as construction time, query time, operation time, and memory usage, covering the ideal mesh data structure that perfectly aligns with the needs of geometric modeling algorithms. It's an essential foundation for creating efficient and powerful algorithms.

How to store geometry & connectivity? *To store geometry and connectivity efficiently, use compact storage techniques like efficient file formats. Optimize algorithms for time-critical operations, such as accessing all vertices/edges of a face or all incident vertices/edges/faces of a vertex. Use adjacency lists or half-edge data structures for connectivity storage, ensuring fast access and traversal. These strategies minimize memory usage, improve performance, and facilitate effective manipulation of meshes.*

Face-Based Data Structures

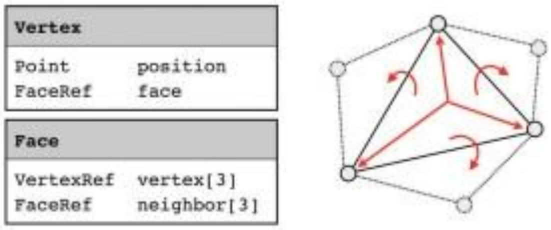


Triangles		
x_{11} y_{11} z_{11}	x_{12} y_{12} z_{12}	x_{13} y_{13} z_{13}
x_{21} y_{21} z_{21}	x_{22} y_{22} z_{22}	x_{23} y_{23} z_{23}
...	...	...
...	...	...
...	...	...
x_{F1} y_{F1} z_{F1}	x_{F2} y_{F2} z_{F2}	x_{F3} y_{F3} z_{F3}

Vertices
x_1 y_1 z_1
...
x_V y_V z_V

Triangles
i_{11} i_{12} i_{13}
...
...
...
...
i_{F1} i_{F2} i_{F3}

Face-set data structure (left) and indexed face-set data structure (right) for triangle meshes.

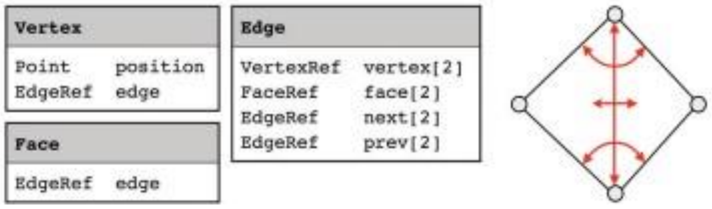


```
Vertex
Point        position
FaceRef      face

Face
VertexRef    vertex[3]
FaceRef      neighbor[3]
```

Connectivity information stored in a face-based data structure.

Edge-Based Data Structures



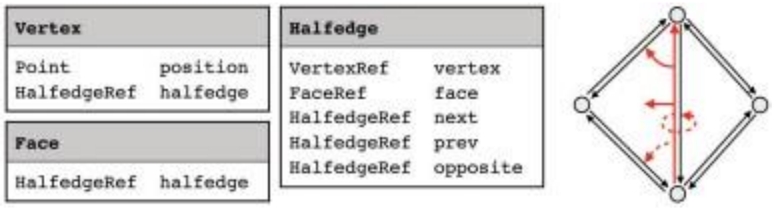
```
Vertex
Point       position
EdgeRef     edge

Face
EdgeRef     edge

Edge
VertexRef   vertex[2]
FaceRef     face[2]
EdgeRef     next[2]
EdgeRef     prev[2]
```

Connectivity information stored in an edge-based data structure.

Halfedge-Based Data Structures

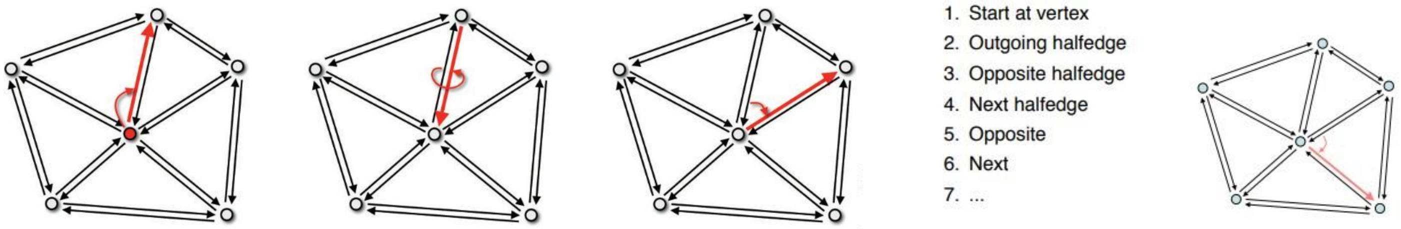


```
Vertex
Point         position
HalfedgeRef   halfedge

Face
HalfedgeRef   halfedge

Halfedge
VertexRef     vertex
FaceRef       face
HalfedgeRef   next
HalfedgeRef   prev
HalfedgeRef   opposite
```

Connectivity information stored in an halfedge-based data structure.



1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge
5. Opposite
6. Next
7. ...

The one-ring neighbors of the center vertex can be enumerated by starting with an outgoing halfedge of the center (left), and then repeatedly rotating clockwise by stepping to the opposite halfedge (center) and next halfedge (right) until the first halfedge is reached again.

Directed-Edge Data Structures

The directed-edge data structure is a memory-efficient variant of the halfedge data structure that is particularly designed for triangle meshes. It is based on indices that reference each element in the mesh (vertex, face, or halfedge). The indexing follows certain rules that implicitly encode some of the connectivity information of the triangle mesh. Instead of pairing opposite halfedges (as proposed in the previous section), this data structure groups the three halfedges belonging to a common triangle. To be more precise, let f be the index of a face.

Then, the indices of its three halfedges are given as halfedge (f, i) = 3f + i, i = 0, 1, 2. Now let h be the index of a halfedge. Then, the index of its adjacent face and its index within that face are simply given by
face(h) = h/3, face index(h) = h mod 3. The index of h's next halfedge can be computed as (h + 1) mod 3. The remaining parts of the connectivity have to be stored explicitly: each vertex stores its position and the index to an outgoing halfedge; each halfedge stores the index of its opposite halfedge and the index of its vertex. This leads to a memory consumption of only 16 bytes/vertex + 8 bytes/halfedge = 64 bytes/vertex, which is just as much as the simple face-based structure of Section 2.1, although the directed edges data structure offers much more functionality. Directed edges can represent all triangle meshes that can be represented by a general halfedge data structure. Note, however, that the boundaries are handled by special (e.g., negative) indices indicating that the opposite halfedge is invalid. Traversing boundary loops is more expensive, since there is no atomic operation to enumerate the next boundary edge. For a general halfedge structure, this can efficiently be accessed by the next halfedge along the boundary.
Although we have here described the directed-edge data structure for pure triangle meshes, an adaption to pure quad meshes is straightforward.
However, it is not possible to mix triangles and quads, or to represent general polygonal meshes. The main benefit of directed edges is its memory efficiency. Its drawbacks are (a) the restriction to pure triangle/quad meshes and (b) the lack of an explicit representation of edges.

Summary and Further Reading

Carefully designed data structures are central for geometry processing algorithms based on polygonal meshes. For most algorithms presented in this book we recommend halfedge data structures, or directed-edge data structures as a special case for triangle meshes. While implementing such datastructures may look like a simple programming exercise at a first glance, it is actually much harder to achieve a good balance between versatility, memory consumption, and computational efficiency. For those reasons we recommend using existing implementations that provide a number of generic features and that have been matured over time. Some of the publicly available implementations include CGAL, OpenMesh, and MeshLab.
Finally, we point the reader to data structures that offer a trade-off between low memory consumption and full access.