Contents

1 Surface Representations

1.1 Polygonal Meshes

Polygonal meshes are defined by a collection of **vertices, edges and faces**:

- **Vertex**: 3D position (x, y, z).
- **Edge**: connection between two vertices.
- **Face**: closed set of edges, *often are triangles.*

More specifically, a polygonal mesh is a finite set M of **closed**, **simple** polygons P_i:

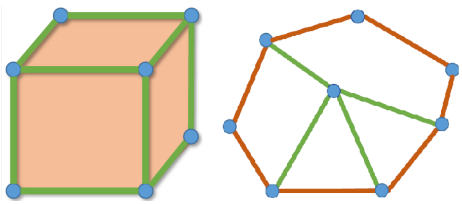
$$M =< V, E, F >$$

A polygon P is defined by an ordered set of vertices $V =< v_0, v_1, ..., v_{n-1} >$ and a set of edges $E =< (v_0, v_1), ..., (v_{n-2}, v_{n-1}) >$ and P is:

- **Closed** if $v_0 = v_{n-1}$
- **Planar** if all vertices on a plane
- **Simple** if it's not self-intersecting

Following are some basic properties of M:

- Intersection of two polygons in M is either empty, a vertex or an edge.
- Every edge belongs to at least one polygon.
- Each P_i defines a $face$ of the polygonal mesh.
- **Boundary** of M is defined as the set of all edges that belong to only one polygon. This set is either empty(left) or forms closed loops(right). If it's empty then the polygon mesh is **closed**.



Often polygonal mesh are constructed by **triangle faces** as:

- It simplifies data structure.
- It simplifies rendering.
- It simplifies computer graphic algorithms.
- Each face is planar and convex.
- Any polygon can be triangulated.

One more thing to add is that there are many ways to describe the same surface as the surface can be decomposed into different triangle faces. So in practise we need to find a way to determine how to form faces.

As a general conclusion this representation achieves a piecewise linear approximation of a surface and the error is bounded by $O(h^2)$, with h denoting the maximal edge length. It has many applications such as in creating template models or content creation for movies, games or mixed reality, and it **can represent arbitrary topology and can be edited/manipulated and rendered efficiently**.

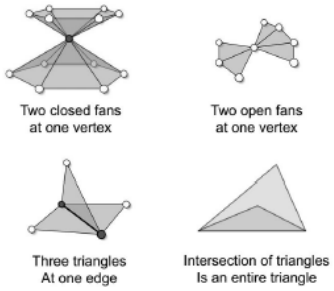
1.1.1 Manifold

A triangle mesh is called **manifold** if:

- The intersection of two triangles is either:
  - Empty
  - A common vertex
  - A common edge
- Edges have:
  - One adjacent triangle → border edge
  - Two adjacent triangles → inner edge
- For a vertex the adjacent triangles:
  - Build a single open fan → border vertex

  - Build a single closed fan → inner vertex

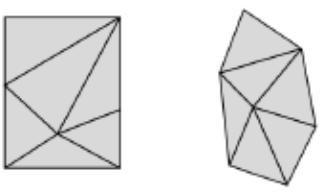
Following are examples of non-manifold triangle meshes:



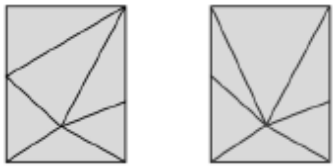
1.1.2 Topology vs. Geometry

Geometry studies the sizes, shapes, positions, angles, and dimensions of things while Topology is concerned with the properties of a geometric object that are preserved under continuous deformations, such as **connectivity**.

Two objects which are topologically equivalent are not necessarily to be at same time geometrically equivalent. On the following picture we can see that **connectivity between vertices are preserved after deformation, which implies they are topologically equivalent, but the shape has changed which means that they are not geometrically equivalent**:



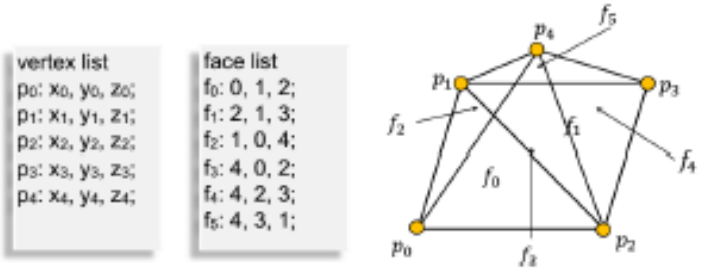
The following picture illustrates the case when two objects are geometrically equivalent but not topologically equivalent. These two figures has the same shape and same area but the connectivity between vertices has changed:



1.1.3 Data structure

As geometrical equivalence doesn't implies topological equivalence that means we can have many ways to represent the same object via triangle meshes, which means a standard data structure for storing vertices, faces and edges matters.

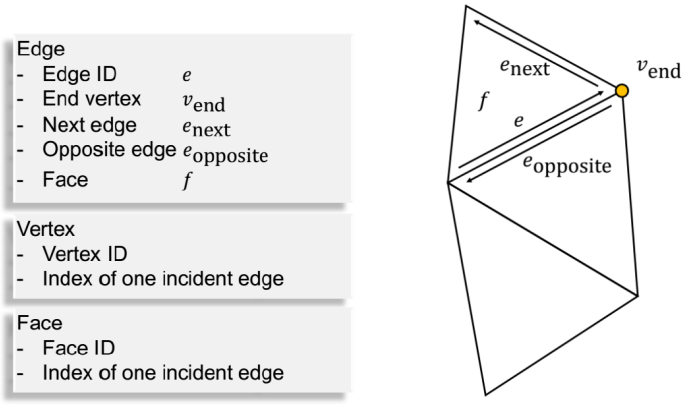
One common standard is to use the **shared vertex data structure**. In this standard every vertex is indexed and every face is constructed by specifying the index of vertices needed to form this face.



But this **format stores the neighborhood information implicitly** as in order to know which triangles share the same vertices we need to check the entire dataset, which could be very inefficient.

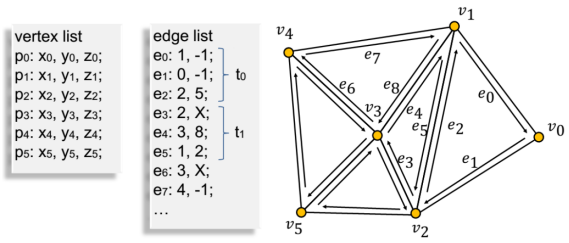
Another option is to use the **half-edge data structure**, but this format is restricted for manifold meshes only. This is an edge-centered data structure capable of maintaining incidence information of vertices, edges and faces. Each edge is decomposed into two halfedges with opposite orientations. One incident face and one incident vertex are stored in each halfedge. For each face and each vertex, one incident halfedge is stored.

This standard provides easy geometric queries and is widely used in geometric computations. For example, given an edge e if we wish to find the adjacent face of this edge then we just need to call $e.opposite.face$.



Half-edge format has a lot of variants, one of them is the **directed edge**. In directed edge format we will have a vertex list and a edge list. Every edge e in the edge list are composed by:

- **The id of the incident vertex**.
- **The id of the opposite edge**. If e is a **border edge** then the opposite edge of e is e inverted and we represent this by negating the id of e. (or we just store a -1).

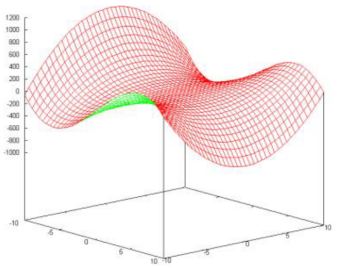


1.2 Explicit Surfaces

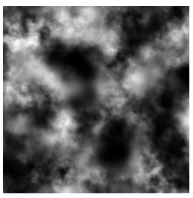
Explicit surfaces are defined by explicit functions. The calculation of surface points of this type of surfaces is very easy as is directly the output of the explicit function. Suppose we have a bunch of 2D points and we like to define a surface in 3D world with these 2D points. Then we just need to compute:

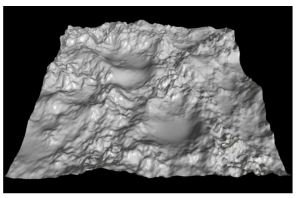
$$S(x, y) = (x, y, f(x, y))$$

Which assigns a corresponding "height" value to each (x, y) pair:



The main disadvantage of this type of surface is that it has restricted shapes, e.g, only one height value per (x,y) pair. But it's simplicity makes it very useful when it's used to model non-complex shapes such as ground.

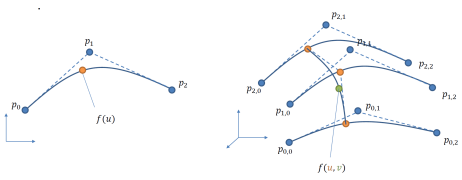




1.3 Parametric Surfaces

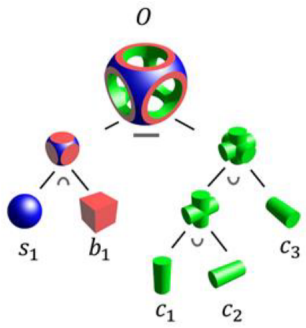
This type of surface representation is heavily used in design and construction as it allows to easily create freeform surfaces. The most well known parametric functions are the **Bezier Curves (2D)** and **Tensor Product Surface (3D)**.

Bezier curve is just an interpolation between a set of control points and the **Tensor Product Surface** is just the replication of 2D **Bezier curve** in 3D space.



1.4 Constructive Solid Geometry

In this type of representation the surface is defined as the boundary of a solid object that was created by Boolean operations on primitive solids. It's commonly used in design and construction but is hard to model "organic" shapes.



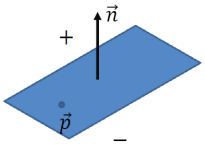
1.5 Implicit Surfaces

In this type of representation the surface is defined implicitly by a function f such that:

- if $f(x, y, z) = 0$ then the point is on surface.
- if $f(x, y, z) < 0$ then the point is inside.
- if $f(x, y, z) > 0$ then the point is outside.

f is also called as **Signed Distance Function** (SDF) as the sign indicates whether the point lies inside or outside the surface and the value indicates how far is the point from the surface. One example of such a function is the **Hesse normal form**:

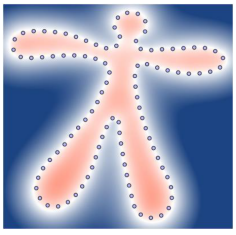
$$f(x, y, z) = (\begin{pmatrix} x \\ y \\ z \end{pmatrix} - \vec{p}) \cdot \vec{n} = 0$$



1.5.1 Calculation of f

Now given a set of sample points, how could we find a scalar function that fulfils:

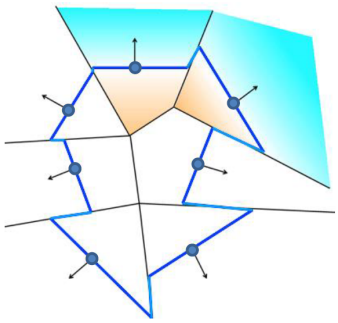
- $f = 0$ at the sample points
- $f < 0$ inside the object
- $f > 0$ outside the object



1.5.1.1 Hoppe's method

Hoppe's method is based on the idea of hesse normal norm that we have seen before. Given a set of sample points, P, and an unknown point, $\vec{x}$, the *location* of x is defined as follow:

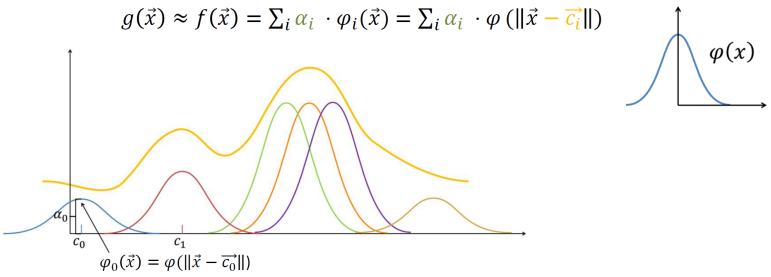
$$f(\vec{x}) = (\vec{x} - \vec{p}) \cdot \vec{n_p},\ \vec{p} \text{ is the closest sample point to } \vec{x}$$



We can see that the resulting function f is a piecewise linear function defined on the Voronoi diagram (partition of a plane into regions close to each of a given set of objects) of the input and it's discontinuous along Voronoi edges. **From this view this method is heavily dependent on the input density in order to produce a smooth surface**.

1.5.1.2 Radial Basis Function Based

We want our estimated f to be smooth, thus, we can use the fact that each complex function can be approximated as the sum of simple **scaled** and **translated** kernel functions $\phi(x)$:



A **Radial Basis Function** (RBF), also called a **Radial Basis Function Network**, is defined as the sum of translated and scaled kernels and a linear polynomial:

$$f(\vec{x}) = \sum_i \alpha_i * \phi_i(\vec{x}) + \vec{b} \cdot \vec{x} + d$$

It can be used to approximate functions of arbitrary complexity and if basis functions $\phi_i(x)$ are smooth, then $f(\vec{x})$ is smooth as well. In our case in order to approximate f we will use the biharmonic radial basis function:

$$\phi_i(\vec{x}) = \|\vec{p_i} - \vec{x}\|^3$$

Where $\vec{p_i}$ are the input sample points.

But $\alpha_i, \vec{b},$ and d are unknown, how could we find these parameters? We have no information about $\alpha_i, \vec{b},$ and d but we do know that $f(\vec{p}) = 0$. Thus, we can construct a system of n (n is the number of sample points) linear equations and solve this to find $\alpha_i, \vec{b},$ and d. But now the problem is that the system of equation that we have just constructed is **undetermined** as we have more unknown parameters (n + 4) than equations (n), and one trivial solution is to set both $\alpha_i, \vec{b},$ and d to 0.

This problem can be solved by adding additional constraints where f is non-zero. **For each sample point we can add off-surface points by moving the points a little in ±normal direction and set the target distance value of these points to** $\pm\epsilon$.

The final system of equation would like the following:

$$f(\vec{x}) = \sum_i \boldsymbol{\alpha_i} \cdot \|\vec{p_i} - \vec{x}\|^3 + \vec{\mathbf{b}} \cdot \vec{x} + \boldsymbol{d}$$

$$\begin{bmatrix} \varphi_{1,1} & \cdots & \varphi_{1,n} & p_{1,x} & p_{1,y} & p_{1,z} & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \varphi_{n,1} & \cdots & \varphi_{n,n} & p_{n,x} & p_{n,y} & p_{n,z} & 1 \\ \hat{\varphi}_{1,1} & \cdots & \hat{\varphi}_{1,n} & q_{1,x} & q_{1,y} & q_{1,z} & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hat{\varphi}_{n,1} & \cdots & \hat{\varphi}_{n,n} & q_{n,x} & q_{n,y} & q_{n,z} & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \\ b_1 \\ b_2 \\ b_3 \\ d \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \pm\varepsilon \\ \vdots \\ \pm\varepsilon \end{bmatrix}$$

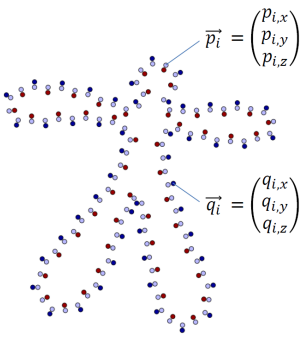
on surface points

off surface points

$$A \qquad \cdot \vec{x} \quad = \vec{b}$$

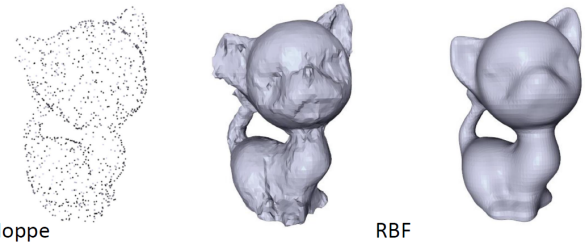
$$\varphi_{i,j} = \|\vec{p_i} - \vec{p_j}\|^3$$

$$\hat{\varphi}_{i,j} = \|\vec{q_i} - \vec{p_j}\|^3$$



Now the system is **overdetermined** and we can just use the least squares solution. The following picture illustrates the difference between the Hoppe method and the RBF method:

1.5.1.3 Comparison between Hoppe method and RBF method



Hoppe
- Local method
- Fast and easy to implement
- Cannot handle noise, outliers, large holes

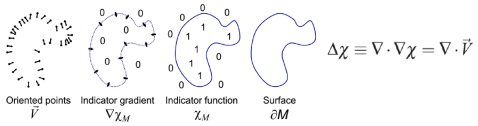
RBF
- Global method
- Requires solving a linear system (slow)
- Can only handle small point sets
- Can handle noise, outliers

⇒ Find a compromise between local and global fitting
- Fit point cloud in a coarse-to-fine manner
- Segment point cloud into parts, fit individually ….

1.5.1.4 Other methods

– RBF with floating centers
- Reduce number of RBF centers
- Also optimize for the center positions → non-linear
- [Süßmuth'10] J.Süßmuth "Surface Reconstruction based on Hierarchical Floating Radial Basis Functions"

– Poisson Reconstruction
- [Kazhdan'06] M.Kazhdan "Poisson Surface Reconstruction"



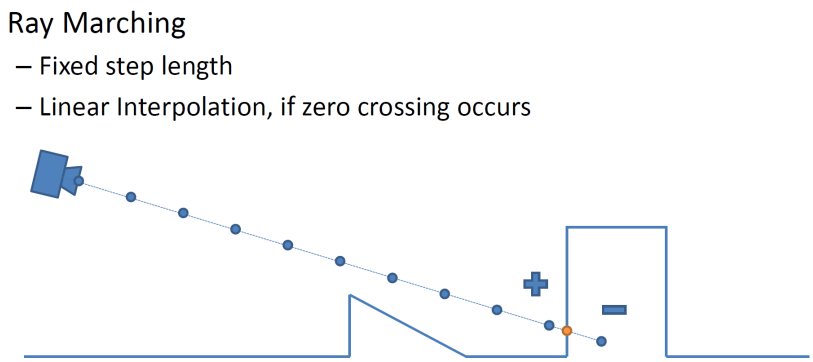
In Poisson reconstruction we'd like to find an indicator function the gradient of which is a vector field that is zero everywhere, except at points near the surface, which should be equal to the inward surface normal. Thus, the oriented point samples can be viewed as samples of the gradient of the model's indicator function.

The problem of computing the indicator function thus reduces to find the scalar function $\mathcal{X}$ whose gradient best approximates a vector field $\vec{V}$ defined by the samples, i.e. $\min_{\mathcal{X}}||\nabla\mathcal{X} - \vec{V}||$. However, $\vec{V}$ is generally not integrable (i.e. it's not curl-free), so an exact solution does not generally exists. To find the best **least-squares approximate** solution, we apply the divergence operator to form the **Poisson equation**:

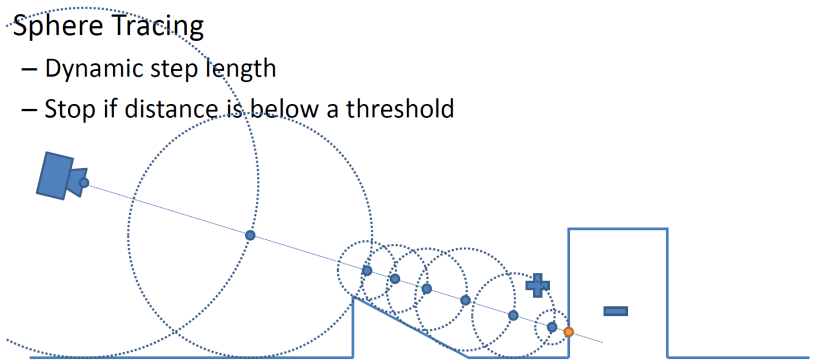
$$\Delta\mathcal{X} = \nabla \cdot \vec{V} \iff \nabla \cdot (\nabla\mathcal{X}) = \nabla \cdot \vec{V}$$

1.5.2 Rendering Signed Distance Functions

1.5.2.1 Ray Marching



1.5.2.2 Sphere Tracing



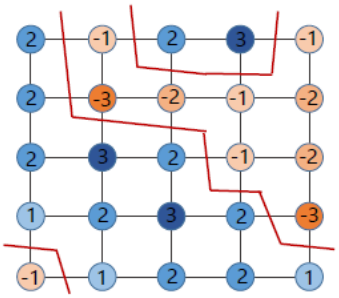
1.5.3 Convert iso-surface to polygonal mesh: Marching Cubes

Given **implicit representation** defined by a function f we would like to convert it to **explicit representation** defined by some type of polygonal meshes, i.e. extract surface at zero-set $\{x : f(x) = 0\}$.

This type of conversion is typically done through **Marching Cubes**. This algorithm is efficient and easy to understand, and before moving to 3D space let's first see how does it works in 2D space to get more intuition.

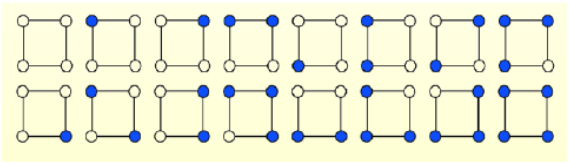
The key idea of this approach is based on **zero-crossing searching**. If we randomly sample points from our implicit function f then **between any pair of points** x, y **such that** $f(x) > 0$ **and** $f(y) < 0$ **there must exists a surface point** s **such that** $f(s) = 0$.

If we sample points in a random way then is hard to find **zero-crossings** as points are unstructured. So, in order to better handle them we will sample points following a **grid structure**:

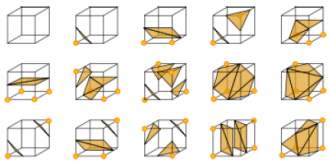


In the above picture we see that **red lines** defines the connection between surface points, thus, defines the surface. In order to create the red line **we will move a square over the grid, and depending on which pairs of vertices contains a zero-crossing we can find the surface point on that corresponding edge**. But, how can we determine where exactly does the zero-crossing happens? I.e., which t should we choose such that $f(x + t(y - x)) = 0$? The simple way is to assume that f is linear between x, y, thus $t = \frac{f(x)}{f(y)-f(x)}$.

In order to make the process more efficient we can create a **lookup table** where we defines all possible vertices combinations which contains a zero-crossing:



In this way we can quickly know which vertices will involve into the calculation of surface point. The same idea is also applied in 3D, but this time instead of moving a square we are moving a **cube** and hence comes the name **Marching cube**. The following is the **lookup table** for cube, in total we have 2^8 combinations but some of them are equivalent and can be reduced to 15 combinations:



It need to be clarified that the above mesh is just an illustration, in practice the size of the triangle mesh depends on the function value of vertices: