ポリゴンモデルのデータ構造と位相操作

1. はじめに

CG で実写さながらの映像を作成するためには、レンダリングの技術が重要な役割を果たしますが、その前段階で行われる形の定義、つまりモデリングの技術も大切です。CG における形状モデリングでは、ものの形を忠実に、しかしながら少ないデータ量で効率的に再現することを目指します。そのため、立体の中身は無視してしまい、表面さえきちんと表現されていれば十分とすることがほとんどです[1]。人体の CT スキャン画像群のように、内部の情報を含むものをボリュームデータと呼ぶのに対して、表面の情報だけ持ったものをサーフェイスデータと呼び、この表面を三角形または四角形、ときには五角形以上の多角形の集合で表したものをポリゴンモデルと呼びます（図 1 左）。ポリゴンモデルはポリゴンメッシュと呼ばれることもありますが、小さな多角形の集合で滑らかな曲面を近似的に表現できるため、曲面の離散的表現と見なすことができます。NURBS 曲面やベジェ曲面のようなパラメトリック曲面[2]（図 1 右）と異なり、複雑な形を比較的容易に扱えることから、幅広く用いられています。

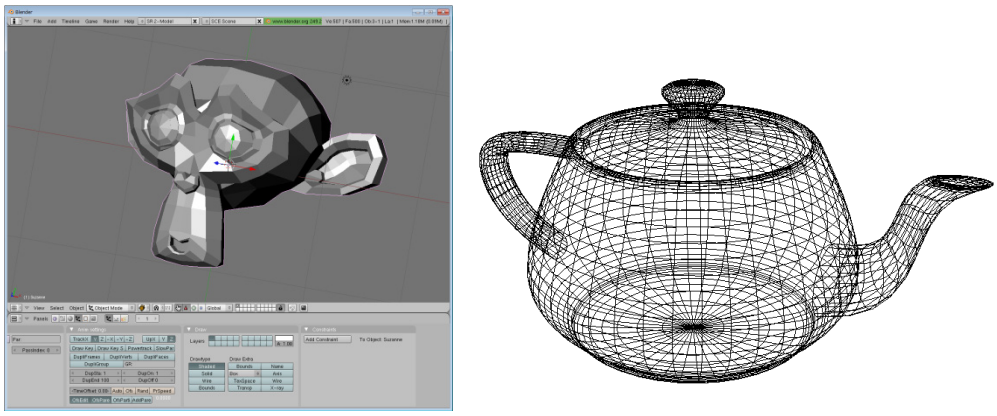

図 1. ポリゴンモデルで表現された Blender[3]のモンキー（左）と、ベジェ曲面で表現されたティーポット（GLUT ライブラリの glutWireTeapot 関数による描画）。

ポリゴンモデルでは、立体の表現に使用する多角形の数によって、データ量と詳細度を調整できるという利点があります。ハードウェアの性能向上に伴って、一度に扱える多角形の数が増え、時代とともに詳細な立体表現が可能となってきました。

ポリゴンモデルのなかでも、特に三角形だけを使って形を表したものを三角形メッシュ

---

1 立体の表面のことを、内部と外部を分ける「境界面」と呼びます
2 パラメータを持つ数式で表現された曲面
3 http://blender.jp/

モデルと呼びます（図 2)。三角形メッシュモデルは、すべての面が 3 つの頂点から構成されるため、単純なデータ構造で扱うことができる利点があり、形状表現の一般的手法として普及しています。

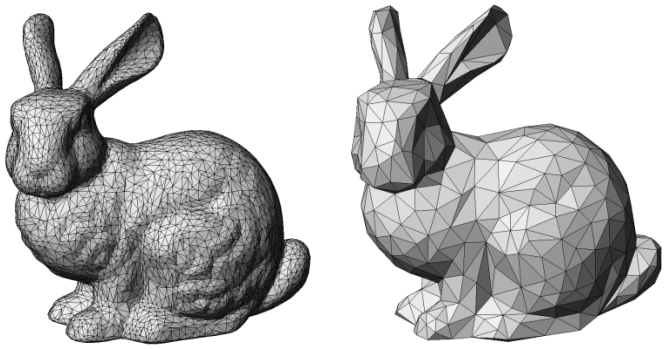

図 2. 三角形の集合で表現されたウサギのモデル（Stanford Bunny[4]）。三角形の数は左側が 10,000、右側は 1,000。

さて、自分で作成するプログラムでポリゴンモデルを扱う際には、その用途に応じたデータ構造を採用することになります。以降では、幾何学の分野の話を少し紹介したのち、ポリゴンモデルを扱う際に用いられる各種のデータ構造と、それぞれの特徴について説明します。その後、ポリゴンモデルを構成する面や頂点の追加・削除を行うアプリケーションの開発を前提として、ハーフエッジ（Halfedge）を基本とするデータ構造をメインに取りあげます。ポリゴンモデルの位相操作は、メッシュ再分割やメッシュ簡略化をはじめ、さまざまな形状編集で必要となるため、CG/CAD の分野のアプリケーションを開発するうえで重要となる処理です。本章の中では、C++言語によるハーフエッジ構造の構築および、位相操作の実装例を紹介します。また、ハーフエッジ構造を活用した事例として、著者が開発したペーパークラフト用の展開図を生成するアプリケーションで採用しているデータ構造の紹介を行います。

2. ポリゴンモデルで表現される立体の幾何

まずはじめに、一見わかりきったもののように思うかもしれませんが、ポリゴンモデルとは何であるのか、ということを幾何学の視点から眺めてみたいと思います。多面体は古くから多くの数学者に関心をもたれ、様々な研究がされています。現場ですぐ役立つ知識ではないですが、少しだけ寄り道してみましょう。

ポリゴンモデルは、頂点（Vertex)、稜線（Edge)、面（Face）を構成要素として持ちま

[4] http://graphics.stanford.edu/data/3Dscanrep/

す。面の周囲は複数の稜線で構成され、稜線は始点と終点に位置する 2 つの頂点で構成されます（図 3)。

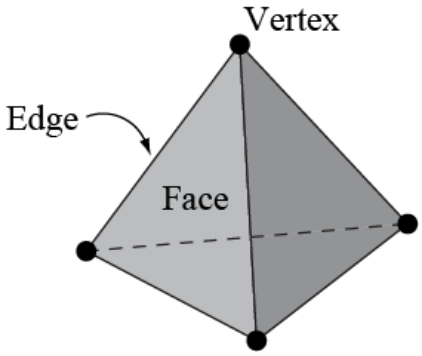

図 3. ポリゴンモデルの構成要素

表面に穴やすき間が無く、立体の内部と外部を明確に区別できるものをソリッドモデルと呼びます。ソリッドモデルの表面は、幾何学の分野では 2 次元多様体（2-manifold）と呼びます[5]。この 2 次元多様体を構成するポリゴンモデルでは、すべての稜線に対して、その稜線に接続する面が 2 つ存在します。球のように閉じた 1 つの立体のモデルに対しては、頂点の数を V、稜線の数を E、面の数を F とすると、V-E+F=2 の関係式が成り立ちます。例えば、図 3 の四面体では、頂点数が 4、稜線の数が 6、面の数が 4 ですから、V-E+F=4-6+4=2 が成り立ちます。

トーラスのように、物体を貫通する穴がある場合、その穴の数を g とすると（これを種数(genus)と呼びます)、V-E+F=2(1-g)の関係式が成り立ち、さらに、面の上に穴（Hole）を持たせることを許容すると、V-E+F-H=2(1-g)の関係式が成り立ちます。図 4 に示す形を左から順に確認してみましょう。左側の立方体は V=8, E=12, F=6, H=0, g=0 です。中央の立体は、トーラスと同じで物体を貫通する穴が 1 つ、そして面の上の穴が 2 つあります。そのため、V=16, E=24, F=10, H=2, g=1 です。右側の立体は、貫通する穴が 2 つ、面の上の穴が 4 つあるので、V=24, E=36, F=14, H=4, g=2 です。いずれも、V-E+F-H=2(1-g)の関係を満たすことが確認できます。

[5] 2 次元多様体は、幾何学の分野では「面上のあらゆる点で円板と同相の近傍を持つ」ものであると表現されます

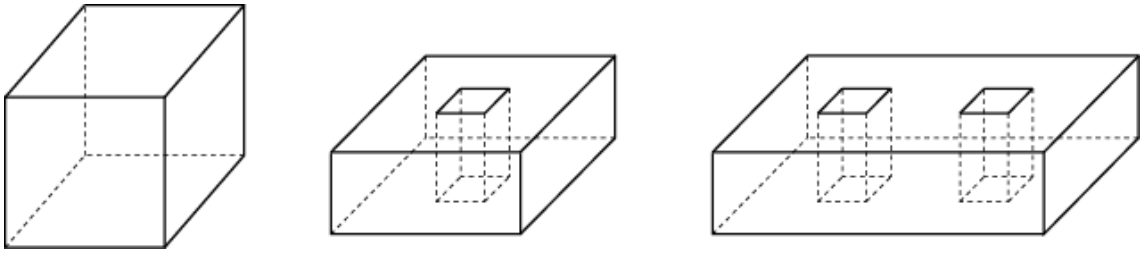

図 4. 立方体と、穴を含む面を持つポリゴンモデル。

この関係式を、オイラー・ポアンカレの式と呼び、多面体の幾何学を学習するときに登場する重要な式になります。ソリッドモデルを表すポリゴンモデルは、すべてこの式を満たします。5 節で紹介する、面や稜線の削除処理の前後でも、この関係は常に維持されます。もし、この式を満たさないポリゴンモデルができてしまったら、それは妥当なソリッドモデルではないと言えます。

たとえば図 5 のようにポリゴンモデルの面が閉じておらず、内部と外部を区別できない場合、これはソリッドモデルと呼ぶことはできません。図 5(左)のように、稜線に接続する面が 1 つまた 2 つだけである場合、それを境界付き 2 次元多様体(2-manifold with boundary)と呼びます。もしくは単にサーフェスモデルと呼びます。機械部品などに使われる CAD と異なり、3 次元キャラクターを扱うことが多い CG では、立体が閉じている必要性が無いので、このようなサーフェスモデルを扱うことが一般的です。また、図 5(右)の中央部のように、稜線に 3 つ以上の面が接続するモデルは、非多様体（non-manifold）モデルと呼びます。面と面の接続関係を見ながら形状処理を行う場合、このような非多様体モデルは処理中に問題を起こすことが多いため、扱いには注意が必要になります[6]。

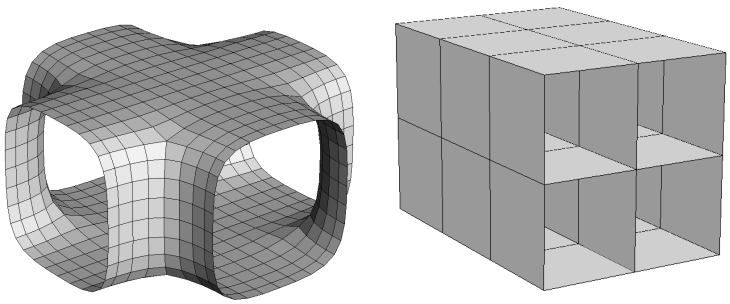

図 5. 境界付き 2 次元多様体モデル（左）と非多様体モデル（右）の例

さて、上記のようなポリゴンモデルを実際にプログラムで扱う場合、面、稜線、頂点の各

[6]本章で紹介するデータ構造は、いずれも非多様体モデルでの面と面の接続関係を扱えません。

情報を、なにかしらのデータ構造でメモリ上に保持する必要があります。頂点の位置を x, y, z の座標値で表した情報が必要であることはすぐにわかりますが、それ以外にも、各面を表示するためには、その面を構成する頂点を参照するための情報も必要になります。頂点の座標値の情報を「幾何情報」と呼び、面、稜線、頂点間の接続関係を表したものを「位相情報」と呼びます。位相情報には、面、稜線、頂点という 3 種類の要素間の組み合わせで 9 種類の関係がありますが、これらの情報を全て保持する必要はなく、単に面を表示するだけであれば、面→頂点の関係さえわかれば十分です。また、この情報から、他の接続関係の情報を復元することができます。

ポリゴンモデルを構成する要素の数や接続関係がアプリケーションの中で変化しない場合（位相が変化しない場合)、データ構造は表示のために必要な最低限のものでかまいませんが、一般に、スムースシェーディングなどのようにある点の近傍の情報を用いる処理では、面と面の接続関係の情報が必要となります。また一方で、頂点や面の追加・削除を行うときには、「ある頂点の周囲にある面をすべて取得する」というような、隣接関係に基づく特定の要素の取得が頻繁に発生します。このような操作を局所化して計算時間を軽減するためには、互いの接続情報も保持できるデータ構造が必要になります。一般に、次のような要件を満たすデータ構造が望まれます。

- ある面の外周を成す稜線を時計回りまたは反時計回りに取得できること。
- ある頂点の周囲にある稜線を時計回りまたは反時計回りに取得できること。
- ある稜線の両端にある頂点および両隣りにある面を取得できること。

このような要件を満たすデータ構造として、次節で挙げるハーフエッジ構造が広く使用されています。

3. ポリゴンモデル表現に用いられる各種データ構造

自作のプログラムでポリゴンモデルを扱う際、どのようなデータ構造を採用するかはその後の処理速度やメモリ使用量を左右する重要事項です。データが小規模で手早く開発することが大事であるならば、実装の効率も考慮すべきでしょう。実際のプログラムで必要となる操作や情報に応じて、データ構造を設計することになりますが、大きく分けると、面を基準としたデータ構造、稜線を基準としたデータ構造、およびハーフエッジという新しい概念を導入したデータ構造の 3 つに分類することができます。必要となるデータ量、処理に要する計算時間、実装の手間などは、それぞれのデータ構造で異なるため、総合的に判断して採用するものを決定することになります。以降では、これらの具体的な構造の説明を行いますが、テクスチャ用の UV 座標や、面の法線ベクトル、その他アプリケーションに固有な追加情報は、必要に応じて各データ構造に付加することになります。

面を基準としたデータ構造

最も単純なデータ構造は、表 1 のように三角形を構成する 3 つの頂点の座標値（x, y, z 値をそれぞれ 3 つ）をそのまま Triangle クラスに持たせたものになります。ポリゴンモデルを表す Model クラスは三角形のリストだけを保持し、それぞれの三角形がどの三角形に隣接するか、という情報は持ちません。つまり、単に三角形が集まっているだけの状態を表します。このようなデータ構造で表される立体は三角形スープ（triangle soup）、またはポリゴンスープ（polygon soup）と呼ばれることがあります。

表 1 最も単純なデータ構造

Triangle （各点が 3 次元座標値を持つ三角形）		
double	x1, y1, z1, x2, y2, z2, x3, y3, z3	3 つの頂点の 3 次元座標

Model （三角形の集合で表されるポリゴンモデル）		
Triangle のリスト	Triangles	三角形の集合

表 1 で示されるデータ構造を C++言語で記述すると、次のようになります[7]。

プログラムコード 1

```
class Triangle {
public:
   double x1, y1, z1, x2, y2, z2, x3, y3, z3;
};

class Model {
public:
   std::list<Triangle*> triangles;
};
```

実装の際には、三角形の集合を保持するのに配列を使っても構いませんが、動的にサイズ変更可能な std::list を使用するのも現実的な方法です。以降では、見やすさを優先して、データ構造を表 1 のような形式で示しますが、適宜、使用する言語に沿ったプログラムコードに読み替えるようにしてください。

[7]本章で紹介するプログラムコードは理解のしやすさを優先しています。本来のオブジェクト指向的な実装方法としては適切でない場合があることをご承知置きください。

さて、このようなシンプルなデータ構造が採用されることは、実際にはあまり無いでしょうが、CAD データから 3D プリンタ出力用などの目的で使われる STL 形式のファイルは、上記のデータ構造で示されるようなファイルフォーマットになっています。STL 形式のファイルには、頂点座標が格納されているだけであって、9 つの数値で 1 つ三角形を定義するようになっています。

通常のポリゴンモデルでは、複数の面が 1 つの頂点を共有します。表 1 のデータ構造では、それぞれの三角形が独自に 3 つの頂点の座標値を持つため、情報の重複があります。そこで、ポリゴンモデル中に存在する頂点の数だけ座標値を保持するようにし、その頂点へのリンクを持つ Face クラスを定義したものが次の表になります。面は三角形に限らず、保持する頂点の数によって四角形、またはそれ以上の多角形に柔軟に対応できるため、クラス名を Triangle とせずに Face としています。三角形に限定するのであれば、頂点のリストは要素数 3 の固定長の配列にします。

表 2 面に頂点へのリンクを持たせたデータ構造

Vertex （頂点）		
double	x, y, z	3 次元の頂点座標

Face （面）		
Vertex のリスト	Vertices	反時計回りに外周を巡回する頂点列

Model （ポリゴンモデル）		
Face のリスト	Faces	面の集合

このデータ構造では、Face クラスが面の外周を構成する頂点へのリンクを保持しています。頂点の並びは、面の表から見た時に反時計回りになる順番で保持するのが一般的です。この順番で面の表と裏（立体の中と外）の区別を行います（図 6)。 OBJ や VRML など、多くの 3DCG ファイルフォーマットで採用されている、面が頂点のインデックス番号の列で定義される方法に対応しています。

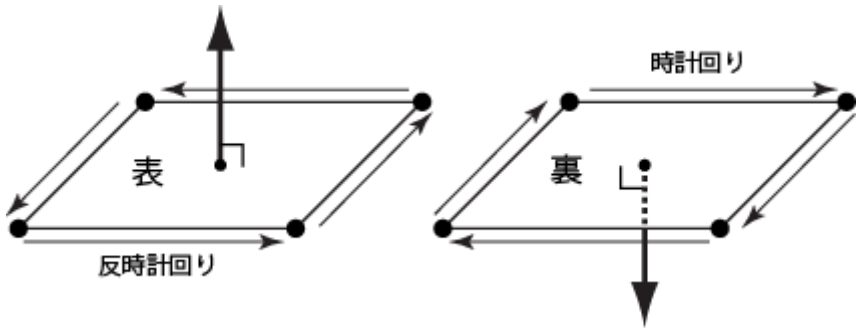

図 6. 頂点の巡回方向と面の法線

実際のプログラムでは、頂点における法線の計算などで、「頂点の周囲にある面」の取得が必要となる場合が多くあります（このような、互いに接続する関係にある要素のことを「1近傍」と表現します）。このような時、表 2 のようなデータ構造だと、隣接要素の取得に全ての面を探索する必要があり、要素数に比例した時間（O(n)の処理時間[8]）がかかってしまいます。

表 3 に示すデータ構造では、[頂点→面]と、[面→隣接面]のリンクを持たせています。[頂点→面]のリンクは 1 つしかありませんが、Face クラスが隣接する面のリンクを保持しているため、これを巡回することで頂点の周りにある面（頂点の 1 近傍と言います）を取得できます。そのため、ある頂点の周囲にある面を全探索なしに定数時間（O(1)の処理時間）で取得できます。

表 3 頂点から面へのリンク、面から隣接面へのリンクを含ませた構造

Vertex （頂点）		
double	x, y, z	3 次元の頂点座標
Face	Face	属する面（複数ある場合はその 1 つ）

Face （面）		
Vertex のリスト	vertices	反時計回りに巡回する頂点列
Face のリスト	faces	隣接する面の集合

Model （ポリゴンモデル）		
Face のリスト	faces	面の集合
Vertex のリスト	vertices	頂点の集合

稜線を基準としたデータ構造

表 4 に示すデータ構造は、稜線を基準に考えられたデータ構造で、図 7 に示すようなリンク関係を持っています。図 7 の中央に位置する Edge が、両側に翼を広げているように見えることから winged-edge 構造と呼ばれます。Edge は、両端の頂点および両側の面へのリンクを持ち、さらに、その面に含まれて自分自身に接続する稜線 4 本のリンクを持ちま

[8] O(n)は O-記法と呼ばれる表記方法でアルゴリズムの処理にかかる計算時間を表します。ここでは、面の数に比例した処理時間がかかることを表します

す。この稜線のリンクをぐるり一周辿ることで、面の外周の稜線と頂点を取得できます。

面に含まれる頂点だけでなく、その逆の「ある頂点の周囲の面」、または「ある頂点に接続する稜線」のような隣接情報を定数時間で取得できるため、このような情報に頻繁にアクセスする場合に用いられます。しかしながら、同様のことは次節で紹介するハーフエッジ構造でも行えるため、winged-edge 構造を採用するケースはあまり見られなくなっています。

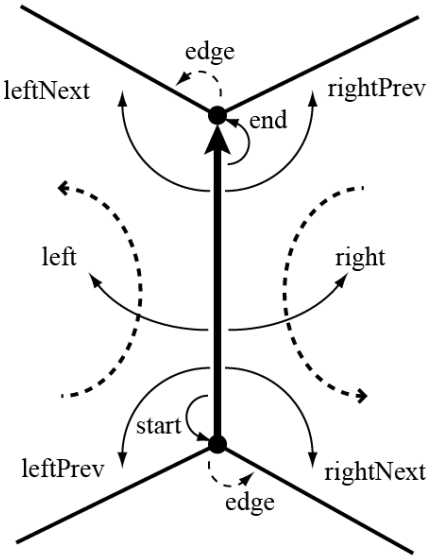

図 7. Winged-edge 構造

表 4 Winged-edge 構造

Vertex （頂点）		
Double	x, y, z	3 次元の頂点座標
Edge	edge	属する稜線のうちの 1 つ

Edge （稜線）		
Vertex	start, end	始点と終点にあたる頂点
Face	left, right	左側と右側に接続する面
Edge	leftNext, leftPrev, rightNext, rightPrev	左側と右側それぞれ、前後に接続する稜線

Face （面）		
Edge	edge	周囲の稜線のうちの 1 つ

Model （ポリゴンモデル）

Face のリスト	faces	面の集合
Edge のリスト	edges	稜線の集合
Vertex のリスト	vertices	頂点の集合

ハーフエッジを基準としたデータ構造

ここでは、面、稜線、頂点という基本要素とは異なる、ハーフエッジ（Halfedge）という要素を新しく導入します。稜線の両側には面があるので、それぞれの面に、稜線を半分ずつ割り当てたものがハーフエッジであると考えるとわかりやすいでしょう（例外的に、サーフェスの境界に位置して面が 1 つしかない稜線にはハーフエッジが 1 つだけ存在することになります[9]）。図 8 に示した矢印がハーフエッジになります。ハーフエッジには向きがあり、各面の外周で反時計回りのループを構成します。また、互いに異なる向きのハーフエッジが 2 つ 1 組で 1 つの稜線を表します。

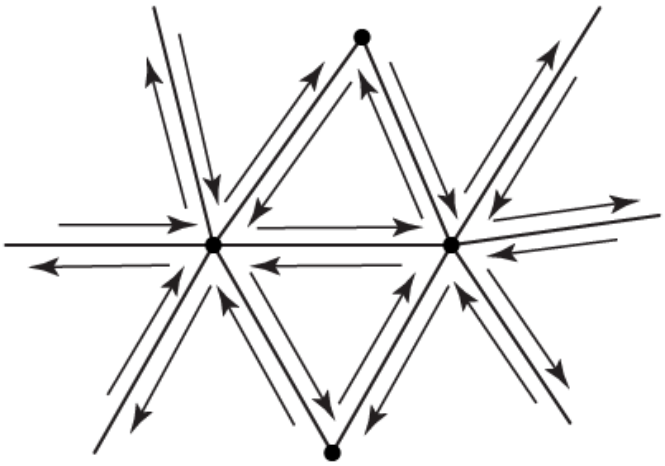

図 8 ハーフエッジのイメージ。稜線の両側に、向きをもった矢印で示される。

このようなハーフエッジを導入することで、表 5 に示すデータ構造によって各要素の接続情報（位相情報）を効率的に保持できます。位相情報は主にハーフエッジに保持されることになり、このようなデータ構造全体をハーフエッジ構造と呼びます。各要素を効率的に参照できるため、多くの現場で採用されています。

ハーフエッジは、面の外周をぐるり 1 周するように配置し、図 9 に示すように互いに next, prev という変数で前後のハーフエッジへのリンクを保持します。また、稜線を表すペアを

[9]面へのリンクを持たないハーフエッジを境界に配置する実装方法もあります。

pair という変数で互いにリンクします。ハーフエッジを含む面および、ハーフエッジの起点となる頂点へのリンクも保持します[10]。頂点には、その頂点を起点とするハーフエッジへのリンクを 1 つだけ持たせます。どれか一つのハーフエッジがわかれば、そこから巡回して、周囲の面や頂点を取得できます。

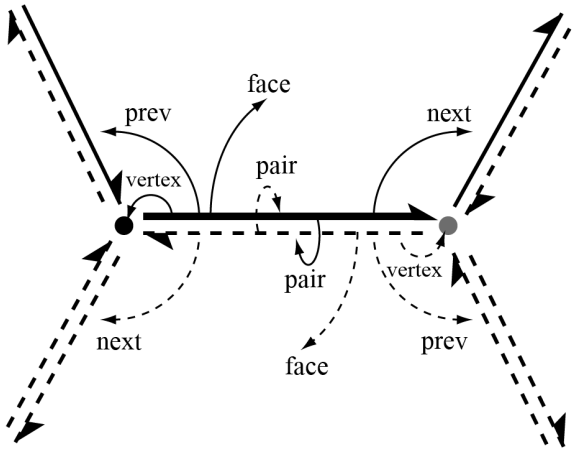

図 9 ハーフエッジ構造での各要素のリンク関係

表 5 ハーフエッジ構造

Vertex （頂点）		
double	x, y, z	3 次元の頂点座標
Halfedge	halfedge	この頂点を始点にもつハーフエッジの 1 つ

Halfedge （ハーフエッジ）		
Vertex	vertex	始点となる頂点
Face	face	このハーフエッジを含む面
Halfedge	pair	稜線を挟んで反対側のハーフエッジ
Halfedge	next	次のハーフエッジ
Halfedge	prev	前のハーフエッジ

Face （面）		
Halfedge	halfedge	含むハーフエッジのうちの 1 つ

[10] ハーフエッジの終点となる頂点へのリンクを保持させる実装も存在します。

Model （ポリゴンモデル）		
Face のリスト	faces	面の集合
Vertex のリスト	vertices	頂点の集合

表 5 に示すハーフエッジ構造には稜線（Edge）のデータが含まれませんが、例えば、稜線に色や角度などの情報を持たせたい場合や、稜線だけを描画することがある場合は、図 10 および表 6 に示すように、稜線を表す Edge クラスを定義します。Edge クラスにハーフエッジへのリンクを持たせる一方で、ハーフエッジには稜線へのリンクを追加します。

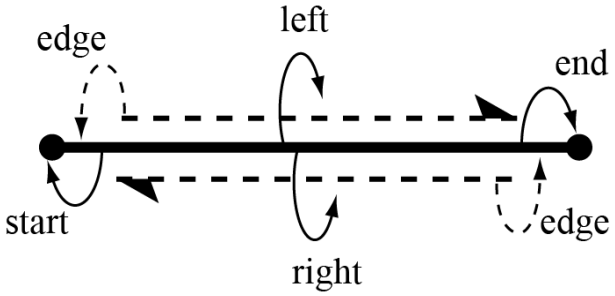

図 10. Edge 要素からのリンク

表 6 Edge クラスを追加する場合

Edge （稜線）		
Vertex	start, end	両端点の頂点
Halfedge	left	始点から終点に向かって左側のハーフエッジ
Halfedge	right	始点から終点に向かって右側のハーフエッジ

4. ハーフエッジ構造の構築例

ハーフエッジ構造は隣接情報の取得に便利な反面、リンク情報が多いのでファイルを読み込んで、そのデータ構造を構築するのに少し手間がかかります。ここでは、OBJ や VRML など、一般的なファイルフォーマットが採用する、[面→頂点]の関係を頂点インデックスで格納したファイルを読み込んで、ハーフエッジ構造のデータを構築する例を紹介します。

例を単純にするために、図 11 に示すようなファイル形式で保存されたテキストファイルを読み込むこととしましょう。このファイルは、最初の 2 つの数字で頂点数と面数を表し、それ以降に、頂点の座標値および、面を構成する頂点のインデックス番号が並びます。また、

面はすべて三角形であるとします。

448 896	頂点数 面数
-40.000000 0.000000 0.000000	1 番目の頂点の(x, y, z)座標
-42.283615 11.480503 0.000000	2 番目の頂点の(x, y, z)座標
-48.786797 21.213203 0.000000	3 番目の頂点の(x, y, z)座標
（中略）	
2 18 17	1 番目の面を構成する頂点のインデックス番号
2 17 1	2 番目の面を構成する頂点のインデックス番号
3 19 18	3 番目の面を構成する頂点のインデックス番号
（後略）	

図 11. OBJ フォーマットに似た形式で、三角形メッシュを表すデータの例

このような形式で、頂点座標と[面→頂点]のインデックスが記述されたファイルを読み込み、ハーフエッジ構造をしたモデルデータを構築するには、次の手順を行うことになります。

```
FOR 頂点の数だけ繰り返し
  頂点座標をファイルから読み込む // (1)
  読みこんだ x, y, z 座標に基づいて新しい Vertex を生成する // (2)
  生成した Vertex を Model に追加する // (3)
END FOR

FOR 面の数だけ繰り返し
  面を構成する頂点インデックスをファイルから読み込む // (4)
  読みこんだ頂点インデックスに基づいて、その頂点を起点とする Halfedge を生成する // (5)
  Halfedge の next と prev のリンクを構築する // (6)
  作成した Halfedge のうちの 1 つにリンクする Face を作成する // (7)
  作成した Face を Model に追加する // (8)
  Halfedge から Face へのリンクを構築する // (9)
  Model から Halfedge の pair を探して互いにリンクさせる // (10)
END FOR
```

簡単なプログラムコードで記述すると次のようになります。プログラムコード中のカッコ付きの番号は、上に記した処理の番号に対応します。

プログラムコード 2

```
#include <cstdio>
#include <list>
#include <vector>
#include <fstream>

class Vertex;
class Halfedge;
class Face;
class Model;

class Vertex {
public:
	double x, y, z;
	Halfedge *halfedge;

	Vertex(double _x, double _y, double _z) {
		x = _x;
		y = _y;
		z = _z;
		halfedge = NULL;
	}
};

class Halfedge {
public:
	Vertex *vertex;
	Face *face;
	Halfedge *pair;
	Halfedge *next;
	Halfedge *prev;

	Halfedge(Vertex *v) {
		vertex = v; // Halfedge → Vertex のリンク
		if(v->halfedge == NULL) {
			v->halfedge = this; // Vertex → Halfedge のリンク
		}
	}
};

class Face {
public:
	Halfedge *halfedge;

	Face(Halfedge *he) {
		halfedge = he; // Face → Halfedge のリンク
	}
};

class Model {
public:
	std::list<Face*> faces;
	std::list<Vertex*> vertices;

private:
	// he と pair となるハーフエッジを探して登録する
	void setHalfedgePair( Halfedge *he ) {
		std::list<Face*>::iterator it_f;
		// すべてのFaceを巡回する
		for( it_f = faces.begin(); it_f != faces.end(); it_f++ ) {
			// 現在のFaceに含まれるハーフエッジを巡回する
			Halfedge *halfedge_in_face = (*it_f)->halfedge;
```

```
                                do {
                                        if(he->vertex == halfedge_in_face->next->vertex &&
                                                he->next->vertex == halfedge_in_face->vertex) {
                                                // 両方の端点が共通だったらpairに登録する
                                                he->pair = halfedge_in_face;
                                                halfedge_in_face->pair = he;
                                                return;
                                        }
                                        halfedge_in_face = halfedge_in_face->next;
                                } while(halfedge_in_face != (*it_f)->halfedge);
                        }
                }


public:
        void Model::addFace( Vertex *v0, Vertex *v1, Vertex *v2 ) {
                Halfedge* he0 = new Halfedge( v0 );     // (5)
                Halfedge* he1 = new Halfedge( v1 );     // (5)
                Halfedge* he2 = new Halfedge( v2 );     // (5)

                he0->next = he1;     he0->prev = he2;    // (6)
                he1->next = he2;     he1->prev = he0;    // (6)
                he2->next = he0;     he2->prev = he1;    // (6)

                Face *face = new Face( he0 );    // (7)
                faces.push_back( face );    // (8)
                he0->face = face;   // (9)
                he1->face = face;   // (9)
                he2->face = face;   // (9)
                setHalfedgePair( he0 );   // (10)
                setHalfedgePair( he1 );   // (10)
                setHalfedgePair( he2 );   // (10)
        }
};

Model* readFile(const char* filename) {

        std::ifstream datafile(filename);

        int vertexNum, faceNum;
        datafile >> vertexNum >> faceNum;

        Model *model = new Model();

        std::vector<Vertex*> vertices; // インデックスで頂点を参照するためのvector

        // 頂点情報の読み込み
        for(int i = 0; i < vertexNum; i++) {
                double x, y, z;
                datafile >> x >> y >> z;             // (1)
                Vertex *v = new Vertex(x, y, z);     // (2)
                model->vertices.push_back(v);        // (3)
                vertices.push_back(v);
        }

        // 面を構成する頂点インデックス情報の読み込み
        for(int i = 0; i < faceNum; i++) {
                int index0, index1, index2;
                datafile >> index0 >> index1 >> index2;     // (4)

                // 3つの頂点を引数にして面の登録を行う
                model->addFace( vertices[index0-1], vertices[index1-1], vertices[index2-1] );
        }
```

```
        return model;
}
```

実装上で難しいのは Halfedge オブジェクトにおける pair の登録です。pair になるのは、一方の始点と終点が他方の終点と始点に一致するハーフエッジの組で、このような組み合わせを見つけるには、基本的には全てのハーフエッジを調べないとなりません。例えばプログラムコード 2 では、Model クラスの setHalfedgePair 関数で、この処理を行っていますが、この実装では素直に全探索を行っているのでハーフエッジの数の 2 乗に比例した時間（$O(n^2)$の処理時間）がかかってしまうことになります。このような処理でも、面の数が数千程度のモデルなら、それほど問題にならないでしょう。実際に、このような実装方法を見かけることがあります。一方で、面の数が大きなモデルを扱うには、処理時間を短くするための工夫が必要になります。

ハーフエッジの pair を高速に見つけるための工夫として、次のような方法があります。

■ハッシュの使用

両端の頂点の ID をキーとしたハッシュでハーフエッジを格納しておきます。ハーフエッジの pair を探索する時は、この頂点の ID をキーとして探索します。もっとも現実的な対応ですが、両端の頂点から適切なキーを生成するための工夫が必要になります。キーを生成するための 1 つの例として、次のような方法があります（下の例では、Vertex に id という変数があることを前提としていますが、頂点のポインタ（アドレス）を ID に使うのもひとつの方法です）。

プログラムコード 3

```
unsigned int getHashKey(const Vertex& v1, const Vertex& v2) {
    unsigned int i1 = v1.id;
    unsigned int i2 = v2.id;
    if (i1 > i2) {
        swap(i1, i2);
    }
    return (i1 | (i2 << 16));
}
```

■頂点→ハーフエッジのリンクの利用

それぞれの頂点にリンクする全てハーフエッジを、その頂点にリストとして保持しておきます。こうすることで、ある頂点を端に持つハーフエッジをすぐに見つけることができま

す。データ構築後はデータを破棄してメモリを確保します。

■行列を使う

頂点の数が n であるとき、n × n の 2 次元行列を準備します。ハーフエッジの両端の頂点のインデックスが i, j であるとき、(i, j) 要素にその Halfedge のポインタを格納しておきます。このようにすることで、同じ 2 つの頂点を両端に持つハーフエッジを、すぐに見つけることができます。行列のサイズに対して、値を持つ要素数が非常に少ないので、疎行列ライブラリを活用することで、使用するメモリを抑えることができます。

上記で紹介した方法は、いずれもハーフエッジの pair の探索が固定時間で行えるので、全体ではハーフエッジの数に比例した時間（O(n)の処理時間）で処理を終えられることになります。実装の手間はかかりますが、処理時間の大幅な削減を実現できます。

5. ハーフエッジ構造での位相操作

ハーフエッジ構造の利点の 1 つに、位相操作を容易に行えることが挙げられます。図 12 に示す稜線削除（Edge Collapse）は、ポリゴン数の削減処理などで頻繁に行われます。また、図 13 に示す稜線交換（Edge Swap）も代表的な位相操作の一つです。これらの処理を行う際には、対象とする要素の近傍の要素を取得し、リンク関係を再構築する必要がありますが、ハーフエッジ構造ではこれらを局所的に行えます[11]。

ここでは、図 12 に示す稜線削除と図 13 に示す稜線交換の操作を例に、具体的な実装例をプログラムコード 4 に示します。それぞれ、Halfedge のポインタを引数にとる edgeCollapse 関数と、edgeSwap 関数で実現されます。図とプログラムコードを注意深く見比べて、どのようにして接続関係の更新が行われるかを確認しましょう。

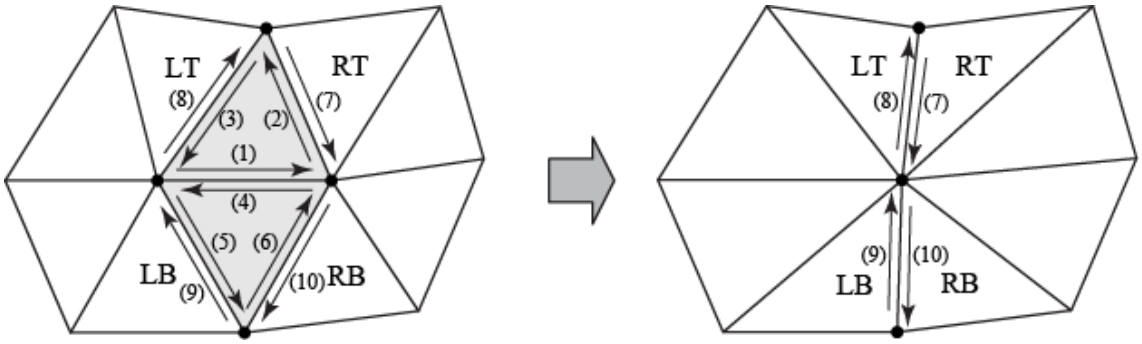

図 12. 稜線削除（Edge Collapse）操作。ハーフエッジ(1)と(4)から構成される稜線と、

[11] 稜線削減時の処理時間は厳密には使用するコレクションライブラリの実装に依存します。

その両側の面を削除する。

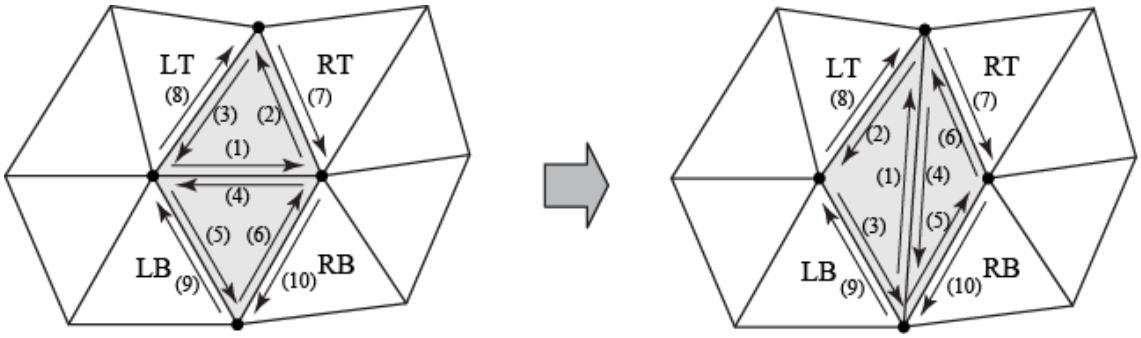

図 13. 稜線交換（Edge Swap）操作。ハーフエッジ(1)と(4)から構成される稜線を付け替える。

プログラムコード 4

```
class Model {
public:
	std::list<Face*> faces;
	std::list<Vertex*> vertices;

/ * 略 */

private:

	// 二つのハーフエッジを互いに pair とする
	void setHalfedgePair(Halfedge *he0, Halfedge *he1) {
		he0->pair = he1;
		he1->pair = he0;
	}

	// Vertex のリストから vertex を削除する
	void deleteVertex( Vertex *vertex ) {
		vertices.remove(vertex);
		delete vertex;
	}

	// Face のリストから face を削除する
	void deleteFace( Face* face ) {
		faces.remove(face);
		delete face->halfedge->next;
		delete face->halfedge->prev;
		delete face->halfedge;
		delete face;
	}

public:
	void edgeCollapse( Halfedge *he ) {
		// 後で図に示した記号で参照できるようにするため
		Halfedge *heLT = he->prev->pair;
		Halfedge *heRT = he->next->pair;
```

```
		Halfedge *heLB = he->pair->next->pair;
		Halfedge *heRB = he->pair->prev->pair;

		// 移動後の頂点の座標値を設定
		he->next->vertex->x = ( he->vertex->x + he->next->vertex->x ) / 2;
		he->next->vertex->y = ( he->vertex->y + he->next->vertex->y ) / 2;
		he->next->vertex->z = ( he->vertex->z + he->next->vertex->z ) / 2;

		// 頂点→ハーフエッジの再設定
		// 頂点が参照するハーフエッジが削除されてしまうかもしれないので設定し直す
		he->next->vertex->halfedge = heRB;  // 右側の頂点から(10)を参照
		he->prev->vertex->halfedge = heRT;  // 上の頂点から(7)を参照
		he->pair->prev->vertex->halfedge = heLB;  // 下の頂点から(9)を参照

		// ハーフエッジ→頂点の更新
		// (1)の始点の頂点周りのハーフエッジを巡回しながら参照頂点を付け替える
		Halfedge *he1 = he->prev->pair;
		do {
			he1->vertex = he->next->vertex;
			he1 = he1->prev->pair;
		} while(he1 != he);

		// pair 関係の構築
		setHalfedgePair(heRT, heLT); // (7)と(8)をpairに登録する
		setHalfedgePair(heLB, heRB); // (9)と(10)をpairに登録する

		// 不要になった要素の削除
		deleteVertex(he->vertex);     // 頂点を削除する
		deleteFace(he->pair->face);   // 一方の面を削除する
		deleteFace(he->face);         // 他方の面を削除する
}

void edgeSwap( Halfedge *he ) {
		Halfedge *heLT = he->prev->pair;
		Halfedge *heRT = he->next->pair;
		Halfedge *heLB = he->pair->next->pair;
		Halfedge *heRB = he->pair->prev->pair;

		// 頂点→ハーフエッジの再設定
		he->next->vertex->halfedge = heRB; // 右側の頂点から(10)を参照
		he->prev->vertex->halfedge = heRT; // 上の頂点から(7)を参照
		he->vertex->halfedge = heLT; // 左の頂点から(8)を参照
		he->pair->prev->vertex->halfedge = heLB;  // 下の頂点から(9)を参照

		// ハーフエッジ→頂点の更新
		he->vertex = heLB->vertex;
		he->next->vertex = heRT->vertex;
		he->prev->vertex = heLT->vertex;
		he->pair->vertex = heRT->vertex;
		he->pair->next->vertex = heLB->vertex;
		he->pair->prev->vertex = heRB->vertex;

		// pair 関係の更新
		setHalfedgePair(heLT, he->next);
```

```
                    setHalfedgePair(heLB, he->prev);
                    setHalfedgePair(heRT, he->pair->prev);
                    setHalfedgePair(heRB, he->pair->next);
        }
};
```

位相操作の処理は、頂点、面、ハーフエッジ間の接続関係が変わるので、注意して実装する必要があります。

稜線削除と稜線交換の処理はどちらも、場合によっては問題を引き起こすことがあり、あらゆる稜線にも適用できるわけではありません。例えば、図 14 で×記号の付いた稜線に稜線削除の処理を施すと、1 つの稜線にぶら下がる三角形ができてしまいます（このような状態を「縮退」と呼びます)。そのため、実際に稜線削除を行う前に、問題が生じないかチェックする仕組みが必要になります。具体的には、次のようにして判定できます。

稜線削除の対象となる稜線の両端点を V0, V1 としたとき、V0 の 1 近傍であり、かつ V1 の 1 近傍である頂点が 3 つ以上存在する場合は削除できない。

稜線交換も同様に、図 15 に示したケースでは 1 つの稜線にぶら下がる三角形ができてしまいます。こちらも事前にチェックする仕組みが必要になります。具体的には、次のようにして判定できます。

稜線交換をした後に稜線の両端点となる予定の 2 頂点が、すでに他の稜線の両端点であるときは稜線交換を実行できない。

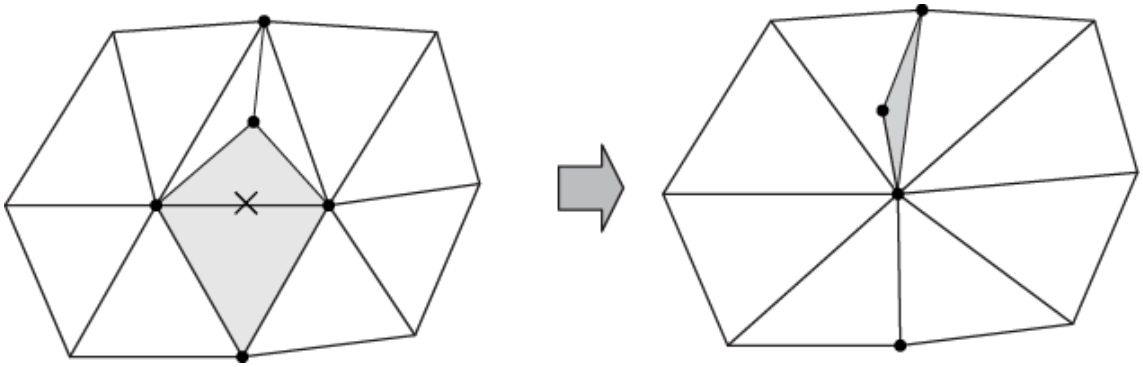

図 14.　稜線削除で問題が生じるケース

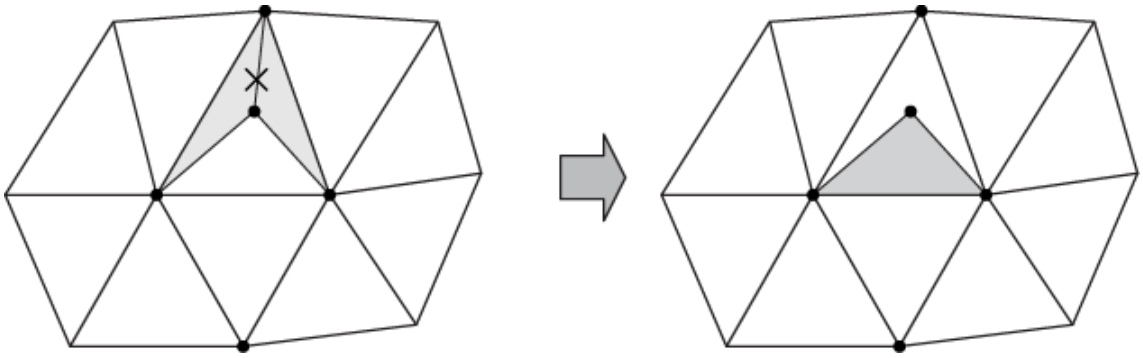

図 15.　稜線交換で問題が生じるケース