Convex Hulls in Three Dimensions

Polyhedra

Polyhedron

1. A **polyhedron** is the generalization of a 2-D polygon to 3-D
   - A finite number of flat polygonal faces
   - The boundary or surface of a polyhedron
     - Zero-dimensional vertices (points)
     - One-dimensional edges (segments)
     - Two-dimensional faces (polygons)

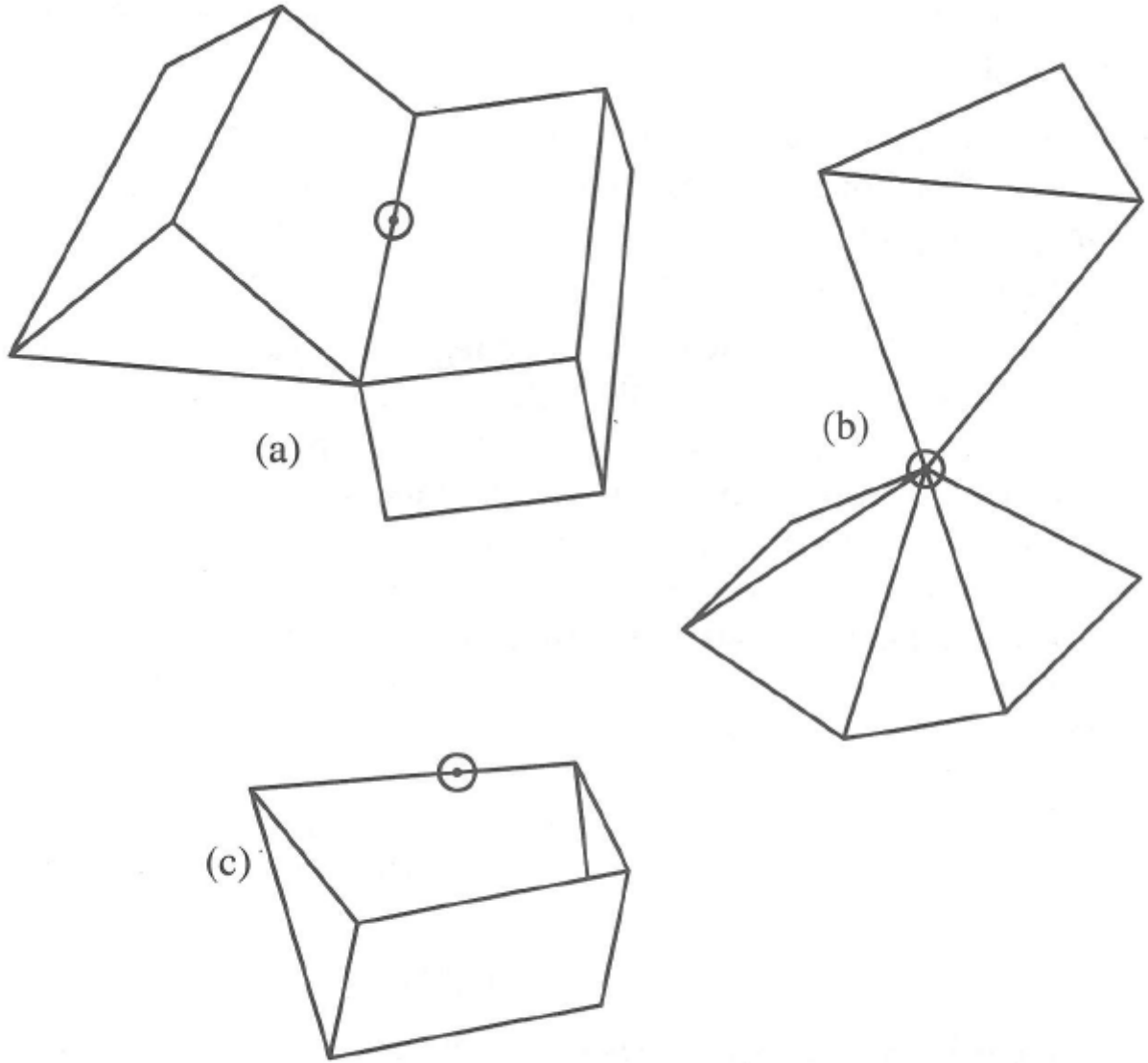


FIGURE 4.2 Three objects that are not polyhedra. In all three cases, a neighborhood of the circled point is not homeomorphic to an open disk. In (a) the point lies both on the top surface shown and on a similar surface underneath. Object (c) is not closed, so the indicated point's neighborhood is a half-disk.

Polyhedron

1. In summary, the boundary of a polyhedron is a finite collection of planar, bounded convex polygonal faces such that
    1. The faces intersect properly;
    2. The neighborhood of every point is topologically an open disk, or the link of every vertex is a simple polygonal chain; and
    3. The surface is connected, or the 1-skeleton is connected

Polyhedron

1. Convex polyhedra are called **polytopes**
2. Polytopes can be specified locally by requiring that all **dihedral** angles be convex.
3. Dihedral angles are the internal angles in space at an edge between the planes containing its two incident faces

Regular Polytopes

1. **Regular polyhedra**
    1. Called regular polytopes
    2. All faces are **regular polygons**
    3. The number of faces incident to each vertex is the same for all vertices
    4. These conditions imply **equal dihedral angles**
2. There are **only five** distinct types of regular polytopes, known as the Platonic solids

Regular Polytopes

1. We have v angles, each $\pi(1 - 2/p)$, which must sum to less than 2π

$$v\pi(1-2/p) < 2\pi, \tag{4.1}$$

$$1-2/p < 2/v,$$

$$pv < 2v+2p,$$

$$pv-2v-2p+4 < 4,$$

$$(p-2)(v-2) < 4. \tag{4.2}$$

2. Both p and v are of course integers and obviously $p \geq 3$, $v \geq 3$

Regular Polytopes

1. These constraints suffice to limit the possibilities to those listed in Table 4.1
2. For example, $p = 4$ and $v = 4$ leads $(p - 2)(v - 2) = 4$, violating the inequality

Table 4.1. Legal p/v values.

p	v	$(p-2)(v-2)$	Name	Description
3	3	1	Tetrahedron	3 triangles at each vertex
4	3	2	Cube	3 squares at each vertex
3	4	2	Octahedron	4 triangles at each vertex
5	3	3	Dodecahedron	3 pentagons at each vertex
3	5	3	Icosahedron	5 triangles at each vertex

Table 4.2. Number of Vertices, Edges, and Faces of the five regular polytopes.

Name	$\{p, v\}$	V	E	F
Tetrahedron	{3, 3}	4	6	4
Cube	{4, 3}	8	12	6
Octahedron	{3, 4}	6	12	8
Dodecahedron	{3, 5}	20	30	12
Icosahedron	{5, 3}	12	30	20

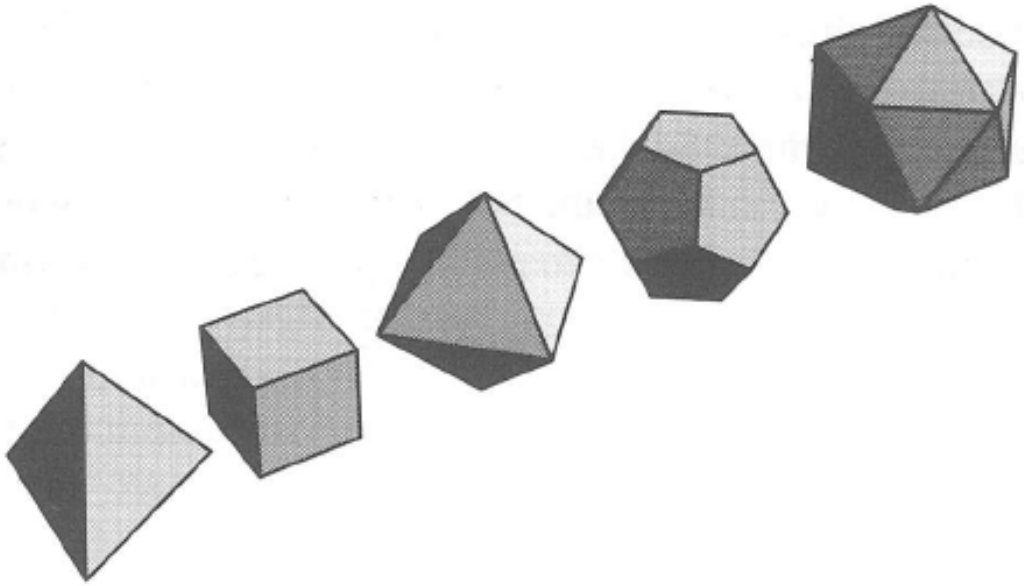


FIGURE 4.3 The five Platonic solids (left to right): tetrahedron, cube, octahedron, dodecahedron, and icosahedron.

Euler's Formula

1. In 1758 Leonard Euler noticed a remarkable regularity in the numbers of vertices, edges, and faces of a polyhedron
2. The number of vertices and faces together is always two more than the number of edges
3. Let V, E, and F be the number of vertices, edges, and faces respectively
4. Euler's formula is:
5. $V - E + F = 2$

Proof of Euler's Formula

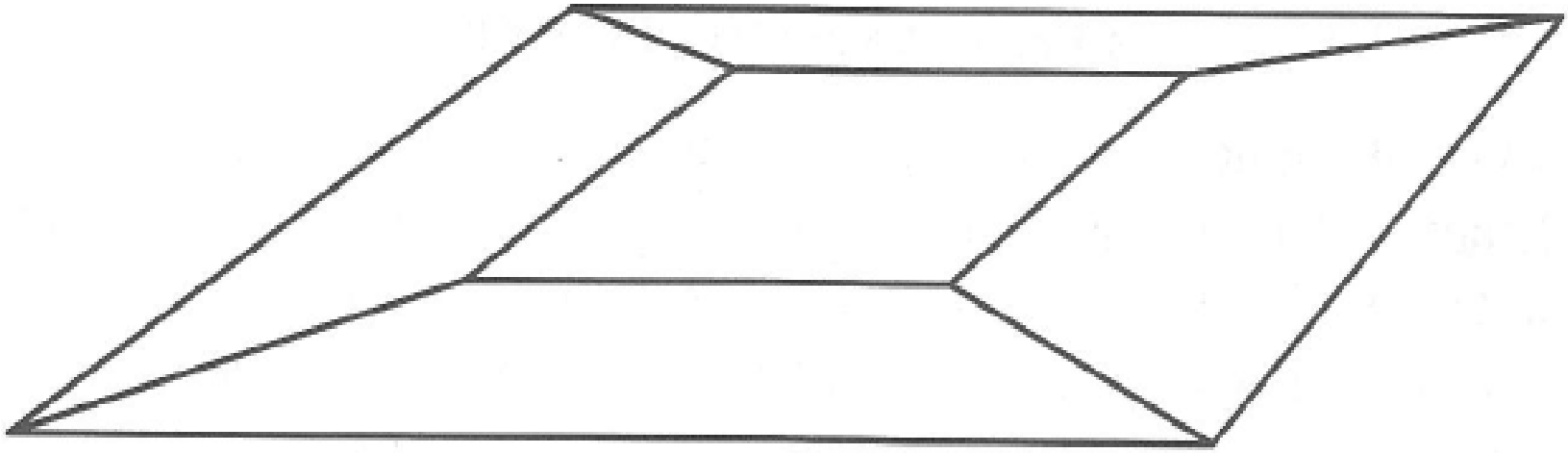


FIGURE 4.4 The 1-skeleton of a cube, obtained by flattening to a plane.

Consequence: Linearity

$$V - E + F = 2,$$

$$V - E + 2E/3 = 2,$$

$$V - 2 = E/3,$$

$$E = 3V - 6 < 3V = 3n = O(n), \tag{4.4}$$

$$F = 2E/3 = 2V - 4 < 2V = 2n = O(n)\ . \tag{4.5}$$

1. Theorem 4.1.1

   1. For a polyhedron with $V = n$, E, and F vertices, edges, and faces respectively; $V - E + F = 2$, and both E and F are O(n)

Convex Hulls in Three Dimensions

Hull Algorithms

Gift Wrapping

1. A careful implementation can achieve $O(n^2)$ time complexity
2. $O(n)$ work per face, and the number of faces is $O(n)$
3. As in two dimensions, this algorithm has the advantage of being output-size sensitive:
4. $O(nF)$ for a hull of F faces

Divide and Conquer

1. The only lower bound for constructing the hull in three dimensions is the same as for two dimensions: $\Omega(n \log n)$
2. The paradigm is the same as in two dimensions:
    1. Sort the points by their x coordinate
    2. Divide into two sets
    3. Recursively construct the hull of each half
    4. Merge
3. All the work is in the merge

Divide and Conquer

1. Let A and B be the two hulls to be merged
2. The hull of $A \cup B$ will add a single “band” of faces with the topology of a cylinder without endcaps
3. See Figure 4.5(b)
4. The number of these faces will be linear in the size of the two polytopes
5. Each face uses at least one edge of either A or B, so the number of faces is no more than the total number of edges
6. Thus it is feasible to perform the merge in linear time

Divide and Conquer

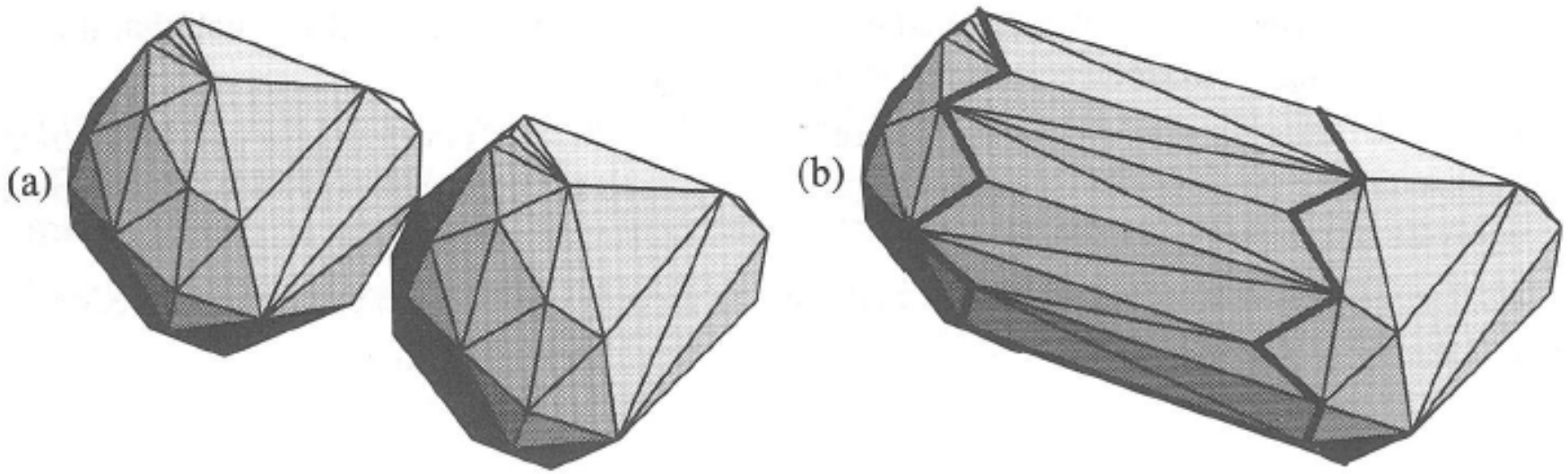


FIGURE 4.5 (a) Polytopes prior to merge. In this example, A and B are congruent, although that will not be true in general. (b) conv $\{A \cup B\}$. The dark edges show the "shadow boundary": the boundary of the newly added faces.

Divide and Conquer

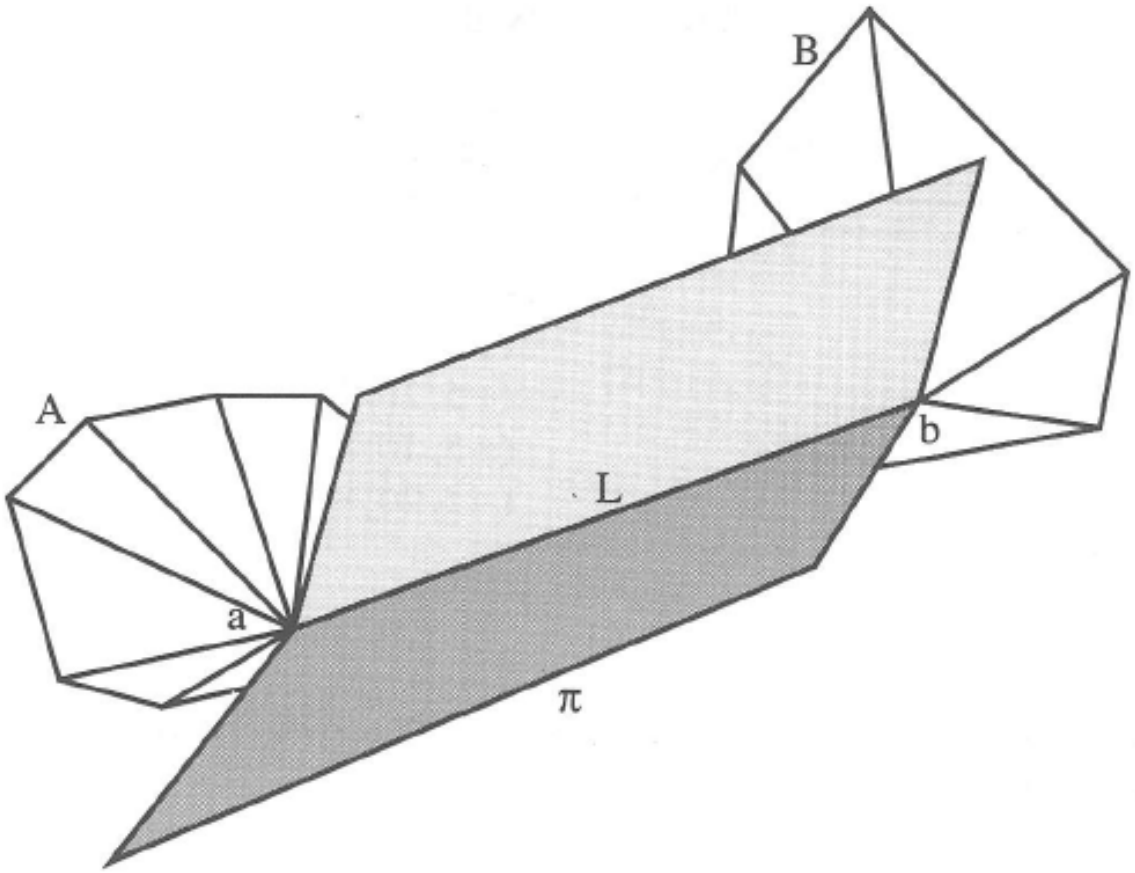


FIGURE 4.6 Plane π is creased along L and bent toward the polytopes A and B (only the faces incident to a and b are shown).

Divide and Conquer

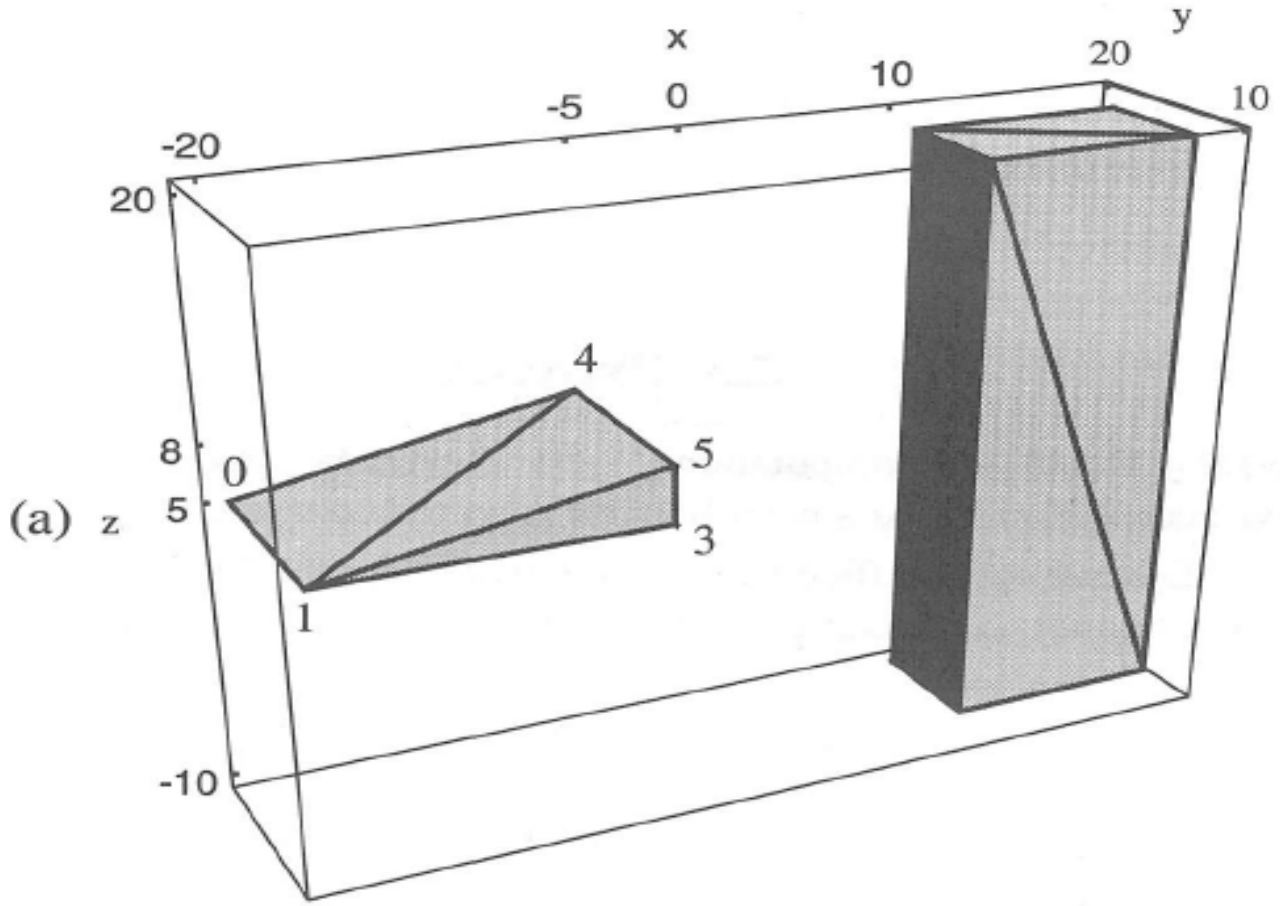


FIGURE 4.7 (a) Wedge and block prior to merging

Divide and Conquer

(b)

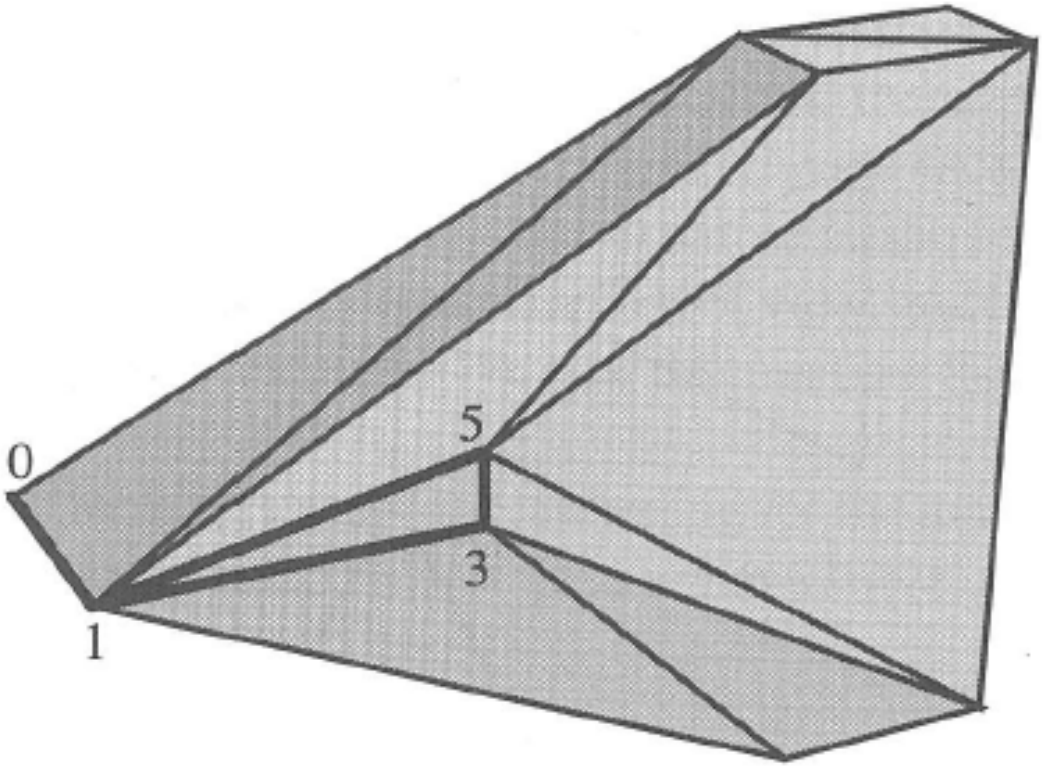


FIGURE 4.7 (b) hull of wedge and block.

Divide and Conquer

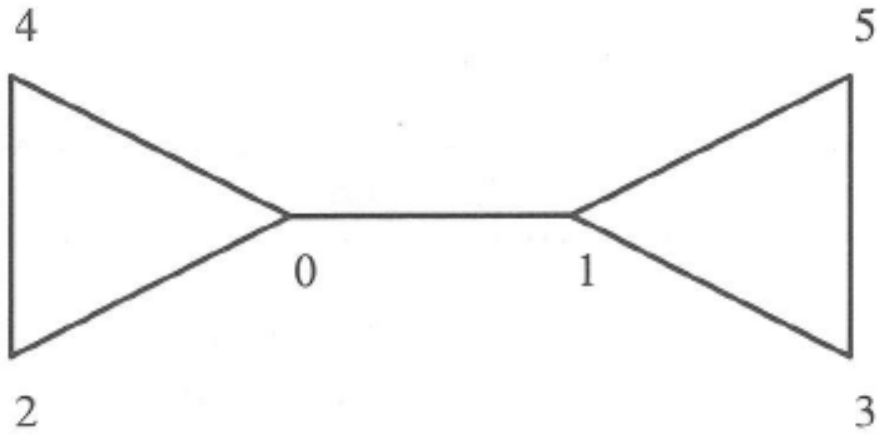


FIGURE 4.8 Topology of shadow boundary edges for Figure 4.7(b).

Incremental Algorithm

1. The overall structure of the three-dimensional incremental algorithm is identical to that of the two-dimensional version
2. At the ith iteration, compute H_i <- conv($H_{i-1} \cup p_i$)
3. Let $p = p_i$ and $Q = H_{i-1}$
4. Two cases
   1. If $p \in Q$, discard p
   2. If not, compute the cone tangent to Q whose apex is p, and construct the new hull

Incremental Algorithm

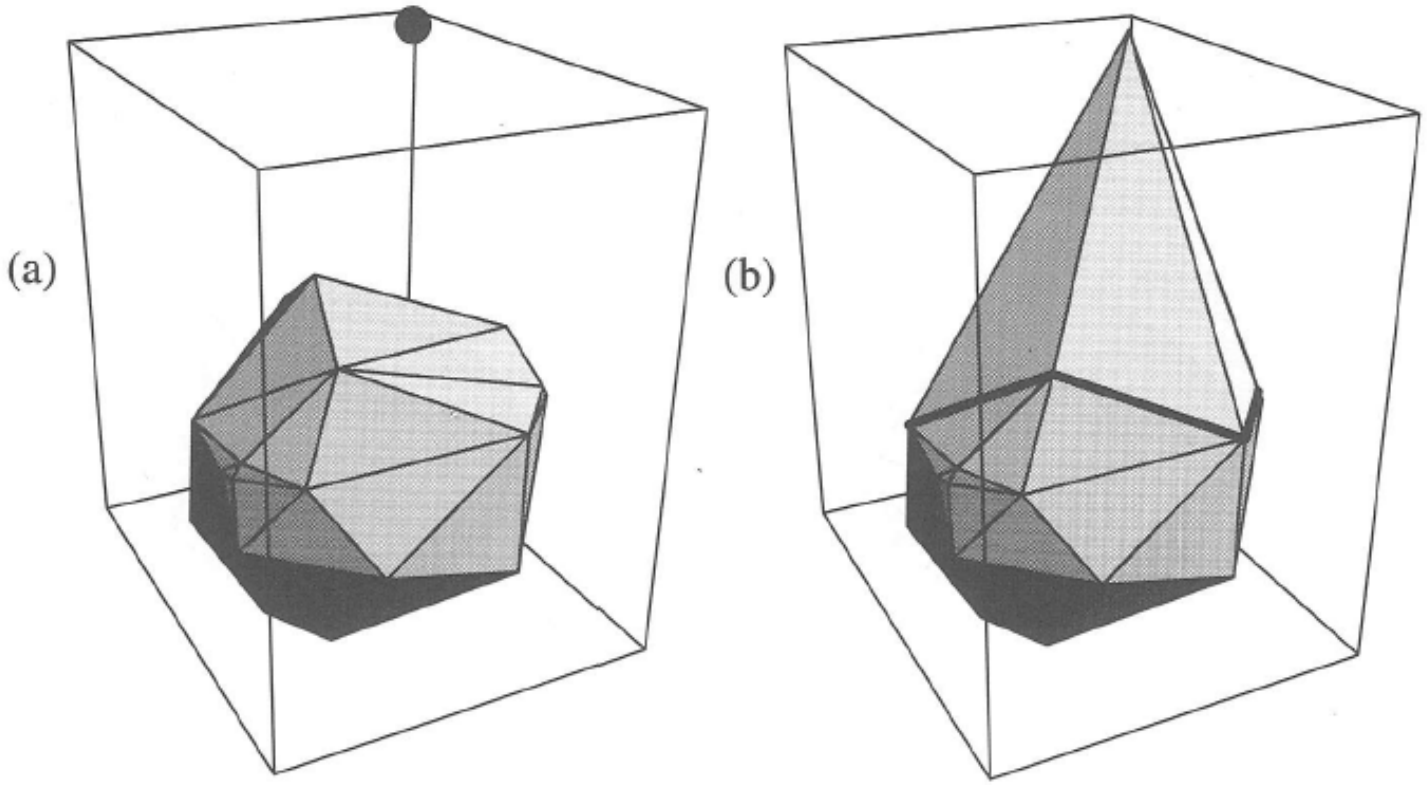


FIGURE 4.10 Viewpoint one: (a) H_{i-1} before adding point in corner; (b) after: H_i.

Incremental Algorithm

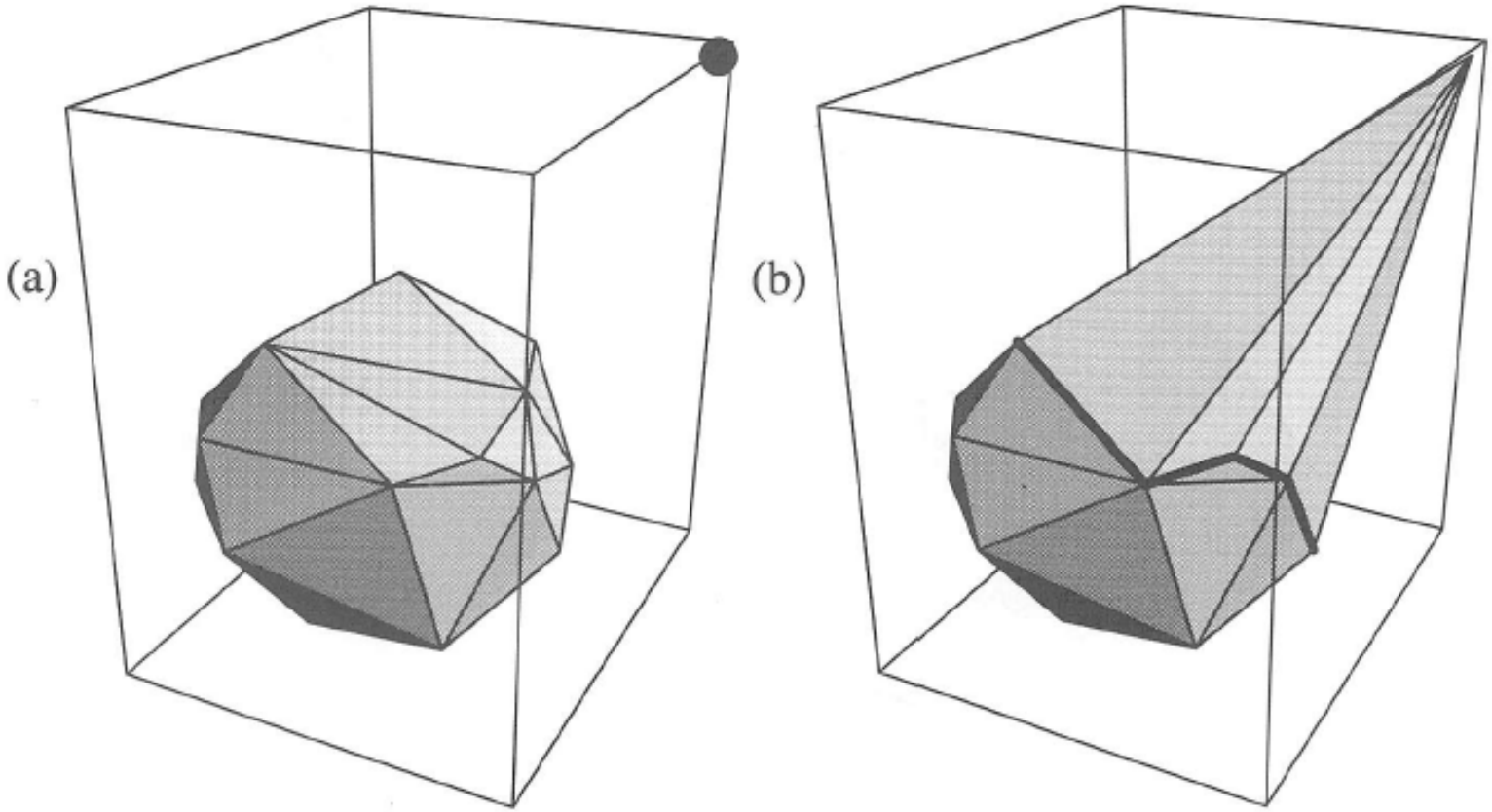


FIGURE 4.11 Viewpoint two: (a) H_{i-1} before adding point in corner; (b) after: H_i.

Incremental Algorithm

Algorithm: 3D INCREMENTAL ALGORITHM
Initialize H_3 to tetrahedron (p_0, p_1, p_2, p_3).
    for $i = 4, \ldots, n-1$ do
    for each face f of H_{i-1} do
        Compute volume of tetrahedron determined by f and p_i.
        Mark f visible iff volume < 0.
    if no faces are visible
        then Discard p_i (it is inside H_{i-1}).
        else
            for each border edge e of H_{i-1} do
                Construct cone face determined by e and p_i.
            for each visible face f do
                Delete f.
            Update H_i.

Algorithm 4.1 Incremental algorithm, three dimensions.

Incremental Algorithm

1. Complexity Analysis

1. $F = O(n)$ and $E = O(n)$, where n is the number of vertices of the polytope
2. So the loops over faces and edges are linear
3. Since these loops are embedded inside a loop that iterates n times, the total complexity is quadratic: $O(n^2)$

Convex Hulls in Three Dimensions

Implementation of Incremental Algorithm

Data Structure

1. Assume the surface of our polytope is triangulated: Every face is a triangle
2. **Structure Definitions**
    1. Three primary data types: vertices, edges, and faces
    2. Doubly lined into a circular list
    3. The ordering has no significance; so these lists should be thought more as sets than as lists
    4. Vertex structure contains the coordinates of the vertex
    5. It contains no pointer to its incident edges nor its incident faces
    6. The edge structure contains pointers to the two vertices and two adjacent faces
    7. The face structure contains pointers to the three vertices and to the three edges
3. See Code 4.1 and 4.2

Data Structure

```
struct tVertexStructure {
   int      v[3];
   int      vnum;
   tVertex next, prev;
};

struct tEdgeStructure {
   tFace    adjface[2];
   tVertex endpts[2];
   tEdge    next, prev;
};

struct tFaceStructure {
   tEdge    edge[3];
   tVertex vertex[3];
   tFace    next, prev;
};
```

Code 4.1 Three primary `structs`.

Data Structure

```
typedef struct tVertexStructure tsVertex;
typedef struct tVertexStructure tsVertex;

typedef struct tEdgeStructure tsEdge;
typedef tsEdge *tEdge;

typedef struct tFaceStructure tsFace;
typedef tsFace *tFace;
```

Code 4.2 Structure typedefs.

Data Structure

1.Example of Data Structure

- Polytope P_5 Consist of 9 edges and 6 faces
- The three lists in Tables 4.5-4.7 are shown exactly as they are constructed by the code
- A view of the polytope is shown in Figure 4.12
- Faces f_2, f_5 , and f_6 are visible; f_0 is on the xy-plane.
- The two “black” faces, f_3 and f_7 are coplanar, forming a square face of the cube, (v_0, v_1, v_5, v_4)

Data Structure

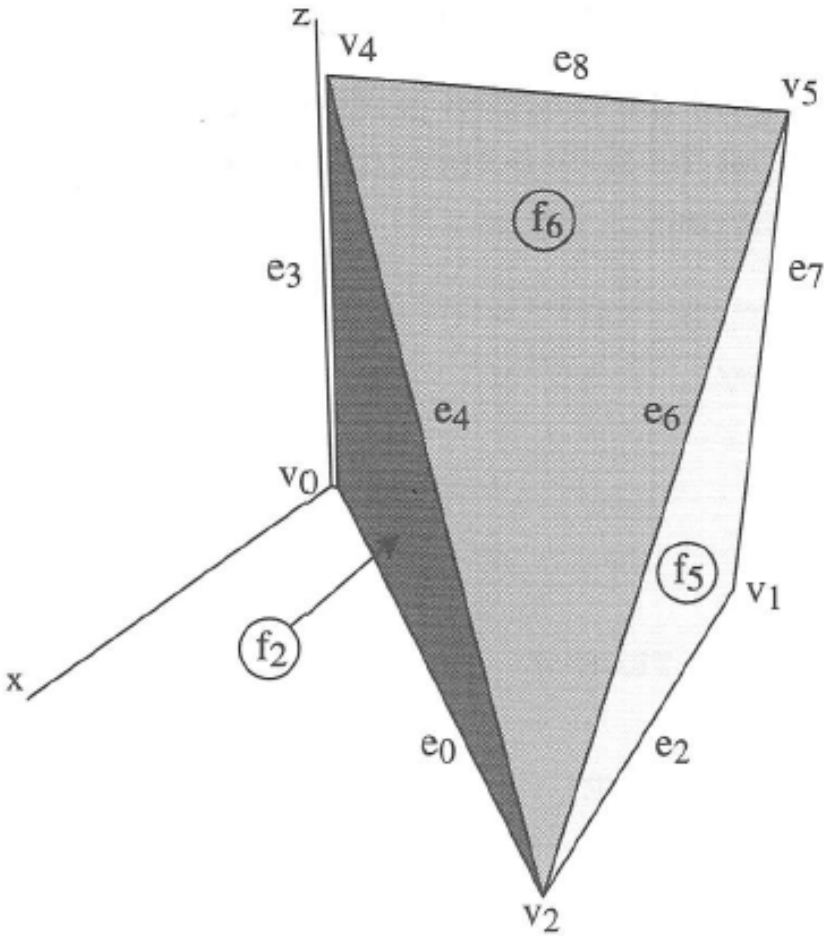


FIGURE 4.12 A view of P_5, with labels.

Data Structure

Table 4.5. Vertex list.

Vertex	Coordinates
v_0	(0, 0, 0)
v_1	(0, 10, 0)
v_2	(10, 10, 0)
v_3	(10, 0, 0)
v_4	(0, 0, 10)
v_5	(0, 10, 10)
v_6	(10, 10, 10)
v_7	(10, 0, 10)

Table 4.6. Edge list.

Edge	Endpoints	Adjacent faces
e_0	(v_0, v_2)	(f_2, f_0)
e_1	(v_1, v_0)	(f_3, f_0)
e_2	(v_2, v_1)	(f_5, f_0)
e_3	(v_0, v_4)	(f_2, f_3)
e_4	(v_2, v_4)	(f_2, f_6)
e_5	(v_1, v_4)	(f_3, f_7)
e_6	(v_2, v_5)	(f_5, f_6)
e_7	(v_1, v_5)	(f_5, f_7)
e_8	(v_4, v_5)	(f_6, f_7)

Table 4.7. Face list.

Face	Vertices	Edges
f_0	(v_0, v_1, v_2)	(e_0, e_1, e_2)
f_2	(v_0, v_2, v_4)	(e_0, e_4, e_3)
f_3	(v_1, v_0, v_4)	(e_1, e_3, e_5)
f_5	(v_2, v_1, v_5)	(e_2, e_6, e_6)
f_6	(v_4, v_2, v_5)	(e_4, e_6, e_8)
f_7	(v_1, v_4, v_5)	(e_5, e_8, e_7)

Data Structure

1. An important property of the face data structure that is maintained at all times is that the vertices in field vertex are ordered counterclockwise,

- so that the right-hand rule yields a vector normal to the face pointing exterior to the polytope
- Thus, f_6's vertices occur in the order (v_4, v_2, v_5)

2. The same counterclockwise ordering is maintained for the edge field

- Thus, the ordering of f_6's edges is (e_4, e_6, e_8)

Data Structure

1. Head Pointers

- At all times a global “head” pointer is maintained to each of the three lists, initialized to NULL

2. See Code 4.3

```
tVertex vertices   =   NULL;
tEdge edges        =   NULL;
tFace faces        =   NULL;
```

Code 4.3 Head Pointers.

Data Structure

1. Basic List Processing

- Four basic list processing routines for each of the three data structures:
  1. Allocation of a new element (NEW)
  2. Freeing an element's memory (FREE)
  3. Adding a new element to the list (ADD)
  4. And deleting an old element (DELETE)

Data Structure

```
#define DELETE( head, p ) if ( head ) {\
   if ( head == head->next ) \
       head = NULL; \
   else if ( p == head ) \
       head = head->next; \
   p->next->prev = p->prev; \
   p->prev->next = p->next; \
   FREE( p ); \
}
```

Code 4.4 DELETE macro.

```
struct tVertexStructure {
  int       v[3];
  int       vnum;
  tEdge     duplicate;    /* pointer to incident cone edge (or NULL) */
  bool      onhull;       /* T iff point on hull. */
  bool      mark;         /* T iff point already processed. */
  tVertex   next, prev;
};

struct tEdgeStructure {
  tFace     adjface[2];
  tVertex   endpts[2];
  tFace     newface;      /* pointer to incident cone face. */
  bool      delete;       /* T iff edge should be delete. */
  tEdge     next, prev;
};

struct tFaceStructure {
  tEdge     edge[3];
  tVertex   vertex[3];
  bool      visible;      /* T iff face visible from new point. */
  tFace     next, prev;
};
```

Code 4.6 Full vertex, edge, and face structures.

Data Structure

```
tVertex MakeNullVertex( void )
{
   tVertex v;

   NEW( v, tsVertex );
   v->duplicate = NULL;
   v->onhull = !ONHULL;
   v->mark = !PROCESSED;
   ADD( vertices, v );

   return v;
}
tEdge  MakeNullEdge( void )
{
   tEdge e;

   NEW( e, tsEdge );
   e->adjface[0] = e->adjface[1] = e->newface = NULL;
   e->endpts[0] = e->endpts[1] = NULL;
   e->delete = !REMOVED;
   ADD( edges, e );
   return e;
}
```

```
tFace  MakeNullFace( void )
{
  tFace f;
  int   i;

  NEW( f, tsFace);
  for ( i=0; i < 3; ++i ) {
     f->edge[i] = NULL;
     f->vertex[i] = NULL;
  }
  f->visible = !VISIBLE;
  ADD( faces, f );
  return f;
}
```

Code 4.7 Full vertex, edge, and face structures.

Example: Cube

1. **Main**
2. The work is separated into four sections at the top level (Code 4.8): read, create initial polytope, construct the hull, and print
3. Code 4.9 shows a list of which routine calls which, with a comment number indicating the order in which they are discussed

Example: Cube

```
main( int argc, char *argv[] )
{
     /* (Flags etc. not shown here.) */

     ReadVertices();
     DoubleTriangle();
     ConstructHull();
     Print();
}
```

Code 4.8 `main`.

Example: Cube

```
/* 1 */    ReadVertices()
                  MakeNullVertex()
/* 2 */    DoubleTriangle()
/* 3 */           Collinear()
/* 4 */           MakeFace()
                           MakeNullEdge()
                           MakeNullFace()
                  VolumeSign()
/* 5 */    ConstructHull()
/* 6 */           AddOne()
/* 7 */             VolumeSign()
/* 8 */           MakeConeFace()
                           MakeNullEdge()
                           MakeNullFace()
/* 9  */                   MakeCcw()
/* 10 */          Cleanup()
/* 12 */                   CleanEdges()
/* 11 */                   CleanFaces()
/* 13 */                   CleanVertices()
                  Print()
```

Code 4.9 Who calls whom. Comments indicate the order of discussion.

Example: Cube

1.ReadVertices

2.The input file for the cube example is:

0	0	0
0	10	0
10	10	0
10	0	0
0	0	10
0	10	10
10	10	10
10	0	10

- They are read in and formed into the vertex list with the straightforward procedures ReadVertices (Code 4.10) and MakeNullVertex (Code 4.7)

Example: Cube

```
void    ReadVertices( void )
{
  tVertex    v;
  int        x, y, z;
  int        vnum = 0;

  while ( scanf ("%d %d %d", &x, &y, &z ) != EOF ) {
    v = MakeNullVertex();
    v->v[X] = x;
    v->v[Y] = y;
    v->v[Z] = z;
    v->vnum = vnum++;
    }
  }
}
```

Code 4.10 ReadVertices.

Example: Cube

1. **DoubleTriangle**
   - To create the initial polytope
   - It is natural to start with a tetrahedron
   - But, use a doubly covered triangle
     - A polyhedron with three vertices and two faces identical except in the order of their vertices
   - The task is not trivial
     - First, it is not adequate to use simply the first three points as those points might be collinear
       - Find three noncollinear points
     - Second, the counterclockwise ordering of the vertices in each face record must be ensured.

Example: Cube

```
void DoubleTriangle( void )
{
  tVertex    v0, v1, v2, v3, t;
  tFace      f0, f1 = NULL;
  tEdge      e0, e1, e2, s;
  int        vol;

  /* Find 3 noncollinear points. */
  v0 = vertices;
  while ( Collinear( v0, v0->next, v0->next->next ) )
     if ( ( v0 = v0->next ) == vertices )
        printf("DoubleTriangle: All points are Collinear!\n"),
        exit(0);
  v1 = v0->next;   v2 = v1->next;

  /* Mark the vertices as processed. */
  v0->mark=PROCESSED; v1->mark=PROCESSED; v2->mark=PROCESSED;

  /* Create the two "twin" faces. */
  f0 = MakeFace( v0, v1, v2, f1 );
  f1 = MakeFace( v2, v1, v0, f0 );
```

```
    /* Link adjacent face fields. */
    f0->edge[0]->adjface[1] = f1;
    f0->edge[1]->adjface[1] = f1;
    f0->edge[2]->adjface[1] = f1;
    f1->edge[0]->adjface[1] = f0;
    f1->edge[1]->adjface[1] = f0;
    f1->edge[2]->adjface[1] = f0;

    /* Find a fourth, noncoplanar point to form tetrahedron. */
    v3 = v2->next;
    vol = VolumeSign( f0, v3 );
    while ( !vol ) {
        if ( ( v3 = v3->next ) == v0 )
            printf("DoubleTriangle: All points are coplanar!\n"),
            exit(0);
        vol = VolumeSign( f0, v3 );
    }

    /* Insure that v3 will be the first added. */
    vertices = v3;
}
```

Code 4.11 `DoubleTriangle`.

Example: Cube

```
bool  Collinear( tVertex a, tVertex b, tVertex c )
{  return
        ( c->v[Z] - a->v[Z] )*( b->v[Y] - a->v[Y] )-
        ( b->v[Z] - a->v[Z] )*( c->v[Y] - a->v[Y] )== 0
      &&( b->v[Z] - a->v[Z] )*( c->v[X] - a->v[X] )-
        ( b->v[X] - a->v[X] )*( c->v[Z] - a->v[Z] )== 0
      &&( b->v[X] - a->v[X] )*( c->v[Y] - a->v[Y] )-
        ( b->v[Y] - a->v[Y] )*( c->v[X] - a->v[X] )== 0 ;
}
```

Code 4.12 Collinear.

Example: Cube

1. **Face construction**
   - Each face is created by an ad hoc routine MakeFace, which takes three vertex pointers as input and one face pointer fold (Code 4.13)
   - If the face pointer fold is not NULL, it uses it to access the edge pointers
   - The goal is to fill the face record with three vertex pointers in the order passed, and with three edge pointers, either constructed de novo (for the first triangle) or
   - copied from fold (for the second triangle), and finally to link the adjface fields of each edge

```
tFace MakeFace( tVertex v0, tVertex v1, tVertex v2, tFace fold )
{
  tFace    f;
  tEdge    e0, e1, e2;

  /* Create edges of the initial triangle. */
  if( !fold ) {
    e0 = MakeNullEdge();
    e1 = MakeNullEdge();
    e2 = MakeNullEdge();
  }
  else { /* Copy from fold, in reverse order. */
    e0 = fold->edge[2];
    e1 = fold->edge[1];
    e2 = fold->edge[0];
  }
  e0->endpts[0] = v0;     e0->endpts[1] = v1;
  e1->endpts[0] = v1;     e1->endpts[1] = v2;
  e2->endpts[0] = v2;     e2->endpts[1] = v0;

  /* Create face for triangle. */
  f = MakeNullFace();
  f->edge[0]   = e0; f->edge[1]   = e1; f->edge[2]   = e2;
  f->vertex[0] = v0; f->vertex[1] = v1; f->vertex[2] = v2;

  /* Link edges to face. */
  e0->adjface[0] = e1->adjface[0] = e2->adjface[0] = f;

  return f;
}
```

Code 4.13 `MakeFace`.

Example: Cube

1. ConstructHull

1. See Code 4.14
2. It is called by `main` after the initial polytope is constructed
3. Simply adds each point one at a time with the function `AddOne`
4. `v->mark` to avoid points already processed
5. After each point is added to the previous hull, an important routine `CleanUp` is called
6. This deletes superfluous parts of the data structure

Example: Cube

```
void   ConstructHull( void )
{
  tVertex    v, vnext;
  int        vol;
  v = vertices;
  do {
     vnext = v->next;
     if ( !v->mark ) {
        v->mark = PROCESSED;
        AddOne( v );
        CleanUp();
     }
     v = vnext;
  } while ( v != vertices );
}
```

Code 4.14 ConstructHull.

Example: Cube

1. AddOne

1. The primary work is done here (Code 4.15)
2. Adds a single point p to the hull, constructing the new cone of faces if p is exterior
3. Two steps to this procedure
    1. Determine which faces of the previously constructed hull are “visible” to p. If no face is visible from p, then p must lie inside the hull and it is marked for subsequent deletion
    2. Add a cone of faces to p. Two adjacent faces are both marked visible are known to be interior to the visible region. Edges with just one adjacent visible face are known to be on the border of the visible region. Constructing this new face is handled by MakeConeFace

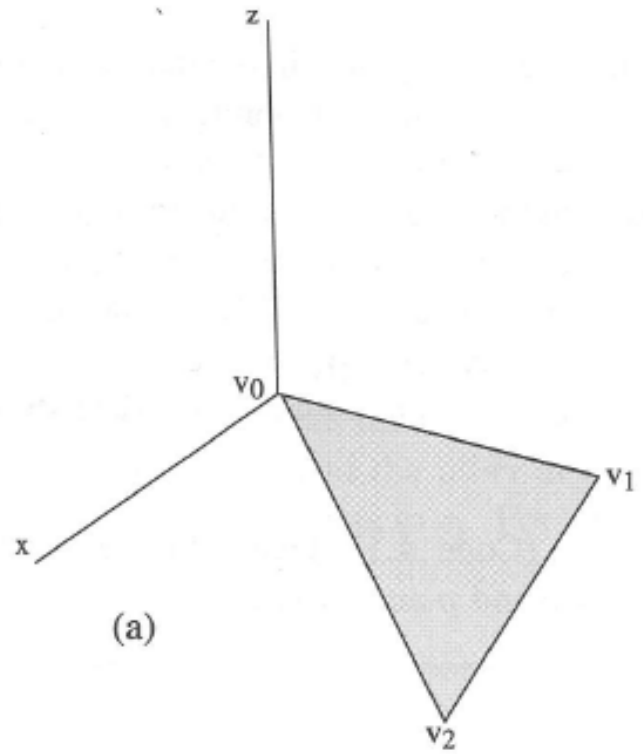
```
bool  AddOne( tVertex p )
{
  tFace    f;
  tEdge    e, temp;
  bool     vis = FALSE;

  /* Mark faces visible from p. */
  f = faces;
  do {
     if ( VolumeSign( f, p ) < 0 ) {
         f->visible = VISIBLE;
         vis = TRUE;
     }
     f = f->next;
  } while ( f != faces );

  /* If no faces are visible from p, then p is inside the hull. */
  if ( !vis ) {
     p->onhull = !ONHULL;
     return FALSE;
  }
```

```
    /* Mark edges in interior of visible region for deletion.
       Erect a newface based on each border edge. */
    e = edges;
    do {
        temp = e->next;
        if ( e->adjface[0]->visible && e->adjface[1]->visible )
            /* e interior: mark for deletion. */
            e->delete = REMOVED;
        else if ( e->adjface[0]->visible || e->adjface[1]->visible )
            /* e border: make a new face. */
            e->newface = MakeConeFace( e, p );
        e = temp;
    } while ( e != edges );
    return TRUE;
}
```

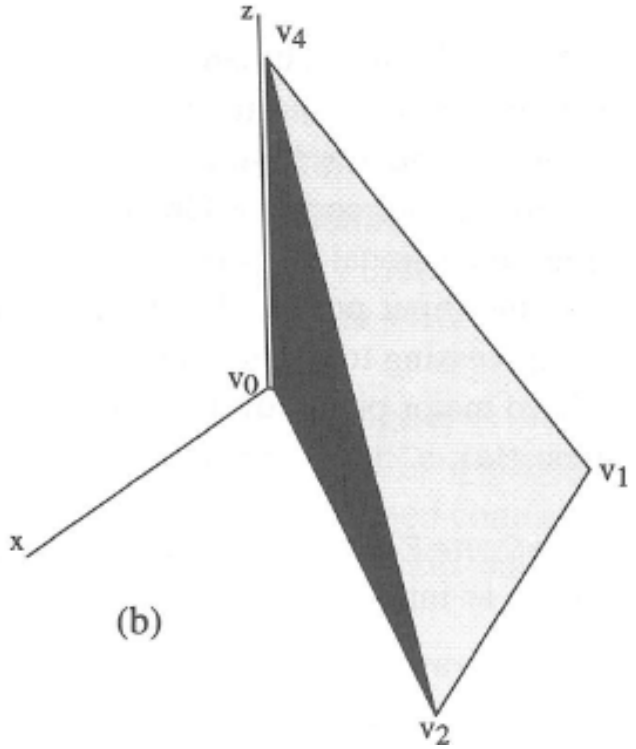
Code 4.15 AddOne.

Example: Cube

1. **AddOne:** Cube Example
    1. The first three vertices by DoubleTriangle: v_0, v_1, v_2
    2. As discussed, the head pointer is moved to v_4
    3. The vertices are then added in the order v_4, v_5, v_6, v_7, v_3
    4. Let P_i be the polytope after adding vertex v_i
    5. The polytopes are then produced in the order P_2, P_4, P_5, P_6, P_7, and P_3
    6. Figure 4.13



(a)



(b)

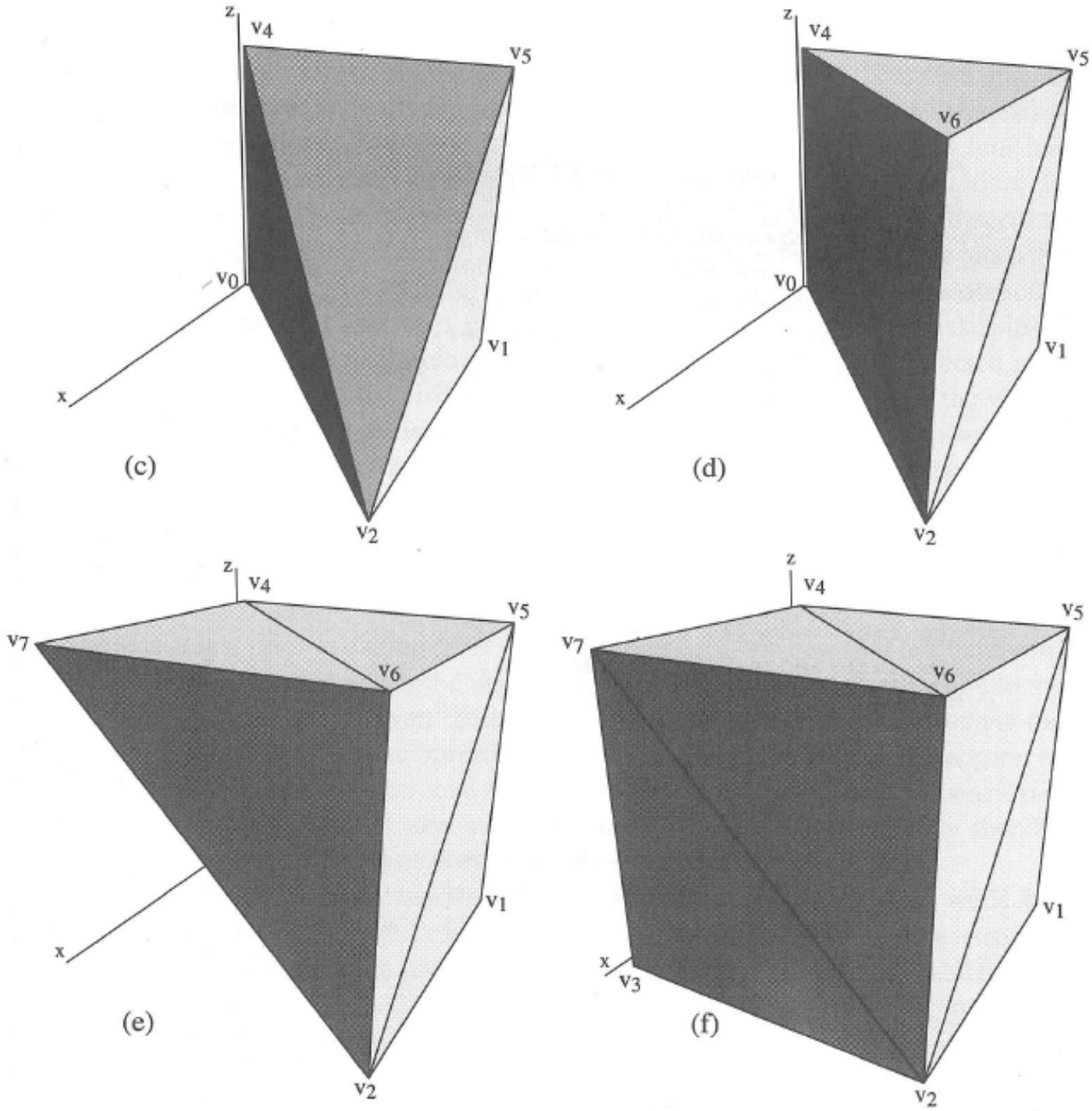


FIGURE 4.13 (a) P_2; (b) P_4; (c) P_5; (d) P_6; (e) P_7; (f) P_3.

Example: Cube

1. MakeConeFace

1. See Code 4.17
2. The routine takes an edge e and a point p as input and creates a new face spanned by e and p and two new edges between p and the endpoints of e.
3. A pointer to the face is returned, and the created structures are linked together properly
4. This is mostly straightforward, but there are two complications

```
tFace  MakeConeFace( tEdge e, tVertex p )
{
    tEdge    new_edge[2];
    tFace    new_face;
    int      i, j;

    /* Make two new edges (if they don't already exist). */
    for ( i=0; i < 2; ++i )
        /* If the edge exists, copy it into new_edge. */
        if ( !( new_edge[i] = e->endpts[i]->duplicate) ) {
            /* Otherwise (duplicate is NULL), MakeNullEdge. */
            new_edge[i] = MakeNullEdge();
            new_edge[i]->endpts[0] = e->endpts[i];
            new_edge[i]->endpts[1] = p;
            e->endpts[i]->duplicate = new_edge[i];
        }

    /* Make the new face. */
    new_face = MakeNullFace();
    new_face->edge[0] = e;
    new_face->edge[1] = new_edge[0];
    new_face->edge[2] = new_edge[1];
    MakeCcw( new_face, e, p );

    /* Set the adjacent face pointers. */
    for ( i=0; i < 2; ++i )
        for ( j=0; j < 2; ++j )
            /* Only one NULL link should be set to new_face. */
            if ( !new_edge[i]->adjface[j] ) {
                new_edge[i]->adjface[j] = new_face;
                break;
            }
    return new_face;
}
```

Code 4.17 `MakeConeFace`.

Example: Cube

1. **MakeConeFace**
    1. Two complications
    2. First, the creation of duplicate edges must be avoided
    2. Second, complication in MakeConeFace is the need to arrange the array elements in the vertex field of *f* in counterclockwise order

```
void    MakeCcw( tFace f, tEdge e, tVertex p )
{
  tFace    fv;   /* The visible face adjacent to e */
  int      i;    /* Index of e->endpoint[0] in fv. */
  tEdge    s;    /* Temporary, for swapping */

  if  ( e->adjface[0]->visible )
       fv = e->adjface[0];
  else fv = e->adjface[1];

  /* Set vertex[0] & [1] of f to have the same orientation
     as do the corresponding vertices of fv. */
  for ( i=0; fv->vertex[i] != e->endpts[0]; ++i )
    ;
  /* Orient f the same as fv. */
  if ( fv->vertex[ (i+1) % 3 ] != e->endpts[1] ) {
     f->vertex[0] = e->endpts[1];
     f->vertex[1] = e->endpts[0];
  }
  else {
     f->vertex[0] = e->endpts[0];
     f->vertex[1] = e->endpts[1];
      SWAP( s, f->edge[1], f->edge[2] );
  }

  f->vertex[2] = p;
}
#define SWAP(t,x,y) { t = x; x = y; y = t; }
```

Code 4.18 `MakeCcw`.

Example: Cube

1. CleanUp

1. The purpose of CleanUp is to “clean up” the three data structures to represent the new hull exactly and only, thereby preparing the structures for the next iteration
2. We partition the work into three natural groups: cleaning up the vertex, the edge, and the face lists
3. See Code 4.19
4. The order in which the three are processed is important

```
void    CleanUp( void )
{
   CleanEdges();
   CleanFaces();
   CleanVertices();
}
```

Code 4.19 `CleanUp`.

```
void    CleanFaces( void )
{
   tFace   f;      /* Primary pointer into face list. */
   tFace   t;      /* Temporary pointer, for deleting. */

  while ( faces && faces->visible ) {
     f = faces;
     DELETE( faces, f );
  }
  f = faces->next;
  do {
     if ( f->visible ) {
         t = f;
         f = f->next;
         DELETE( faces, t );
     }
     else f = f->next;
  } while ( f != faces );
}
```

Code 4.20 `CleanFaces`.

```
void    CleanEdges( void )
{
   tEdge e;    /* Primary index into edge list. */
   tEdge t;    /* Temporary edge pointer. */

   /* Integrate the newfaces into the data structure. */
   /* Check every edge. */
   e = edges;
   do {
      if ( e->newface ) {
         if ( e->adjface[0]->visible )
            e->adjface[0] = e->newface;
         else e->adjface[1] = e->newface;
         e->newface = NULL;
      }
      e = e->next;
   } while ( e != edges );
```

```
    /* Delete any edges marked for deletion. */
    while ( edges && edges->delete ) {
        e = edges;
        DELETE( edges, e );
    }
    e = edges->next;
    do {
        if ( e->delete ) {
            t = e;
            e = e->next;
            DELETE( edges, t );
        }
      else e = e->next;
    } while ( e != edges );
}
```

Code 4.21 `CleanEdges`.

Code 4.22

```
void    CleanVertices( void )
{
   tEdge      e;
   tVertex    v, t;

   /* Mark all vertices incident to some undeleted edge as on the hull. */
   e = edges;
   do {
      e->endpts[0]->onhull = e->endpts[1]->onhull = ONHULL;
      e = e->next;
   } while (e != edges);

   /* Delete all vertices that have been processed but are not on the hull. */
   while ( vertices && vertices->mark && !vertices->onhull ) {
      v = vertices;
      DELETE( vertices, v );
   }
   v = vertices->next;
   do {
      if ( v->mark && !v->onhull ) {
         t = v;
         v = v->next;
         DELETE( vertices, t )
      }
      else v = v->next;
   } while ( v != vertices );
```

Code 4.22 (con't)

```
    /* Reset flags. */
    v = vertices;
    do {
        v->duplicate = NULL;
        v->onhull = !ONHULL;
        v = v->next;
    } while ( v != vertices );
}
```

Code 4.22 CleanVertices.

Checks

1. Face orientations: Check that the endpoints of each edge occur in opposite orders in the two faces adjacent to that edge
2. Convexity: Check that each face of the hull forms a nonnegative volume with each vertex of the hull
3. Euler's relations: Check that $F = 2V - 4$ (Equation 4.5) and $2E = 3V$

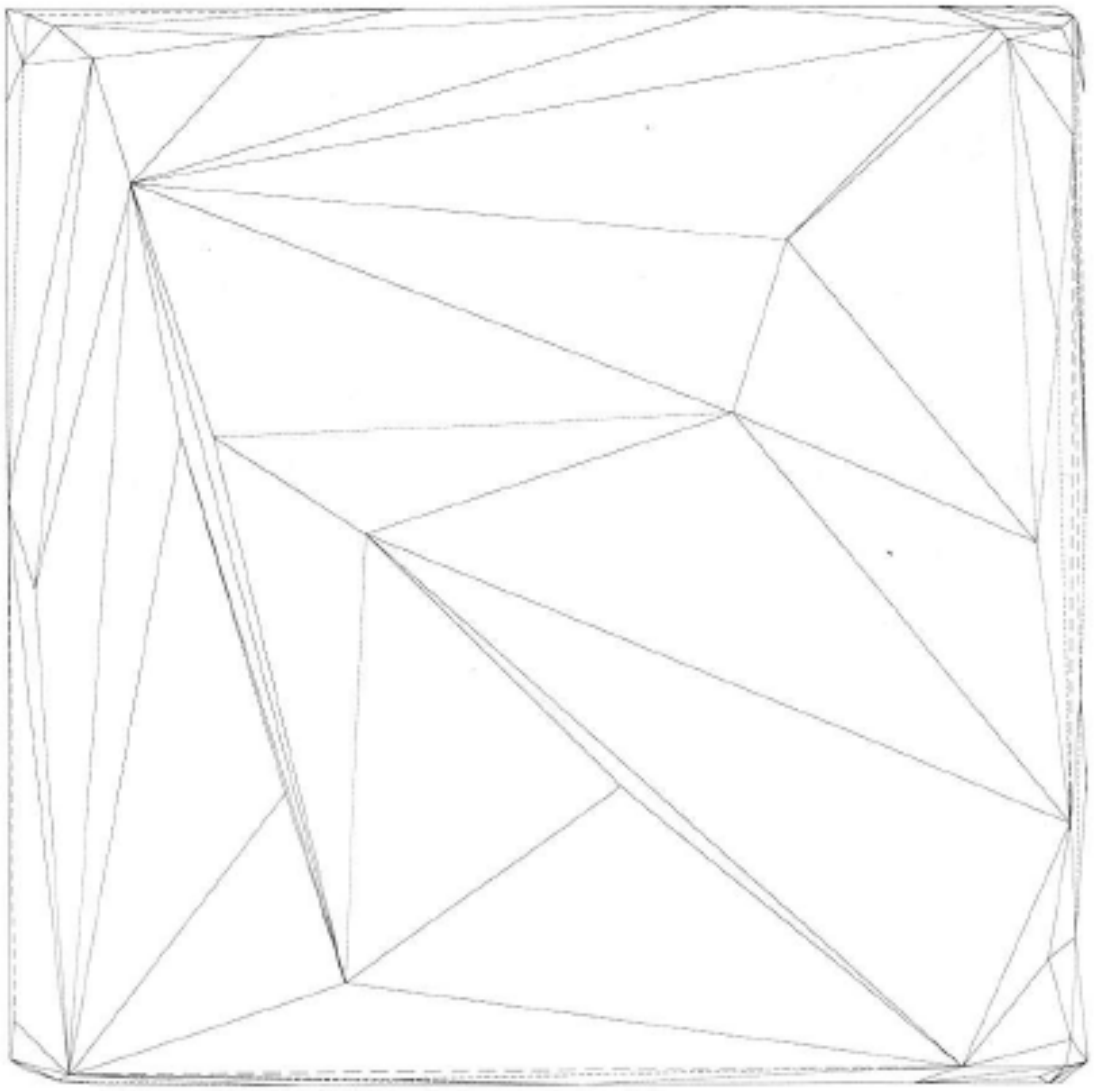


FIGURE 4.14 Hull of 10,000 points in a cube.

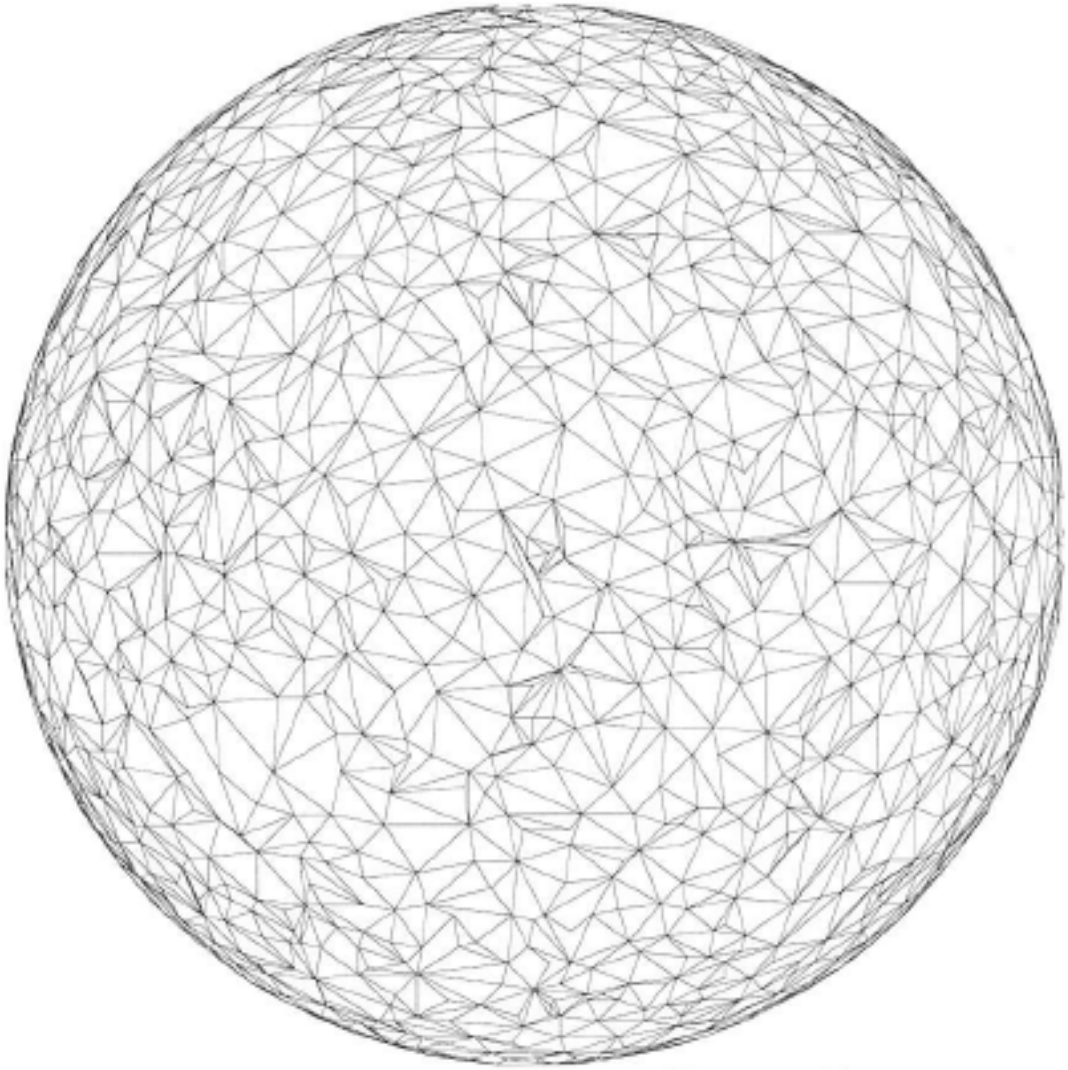


FIGURE 4.15 Hull of 10,000 points near the surface of a sphere.

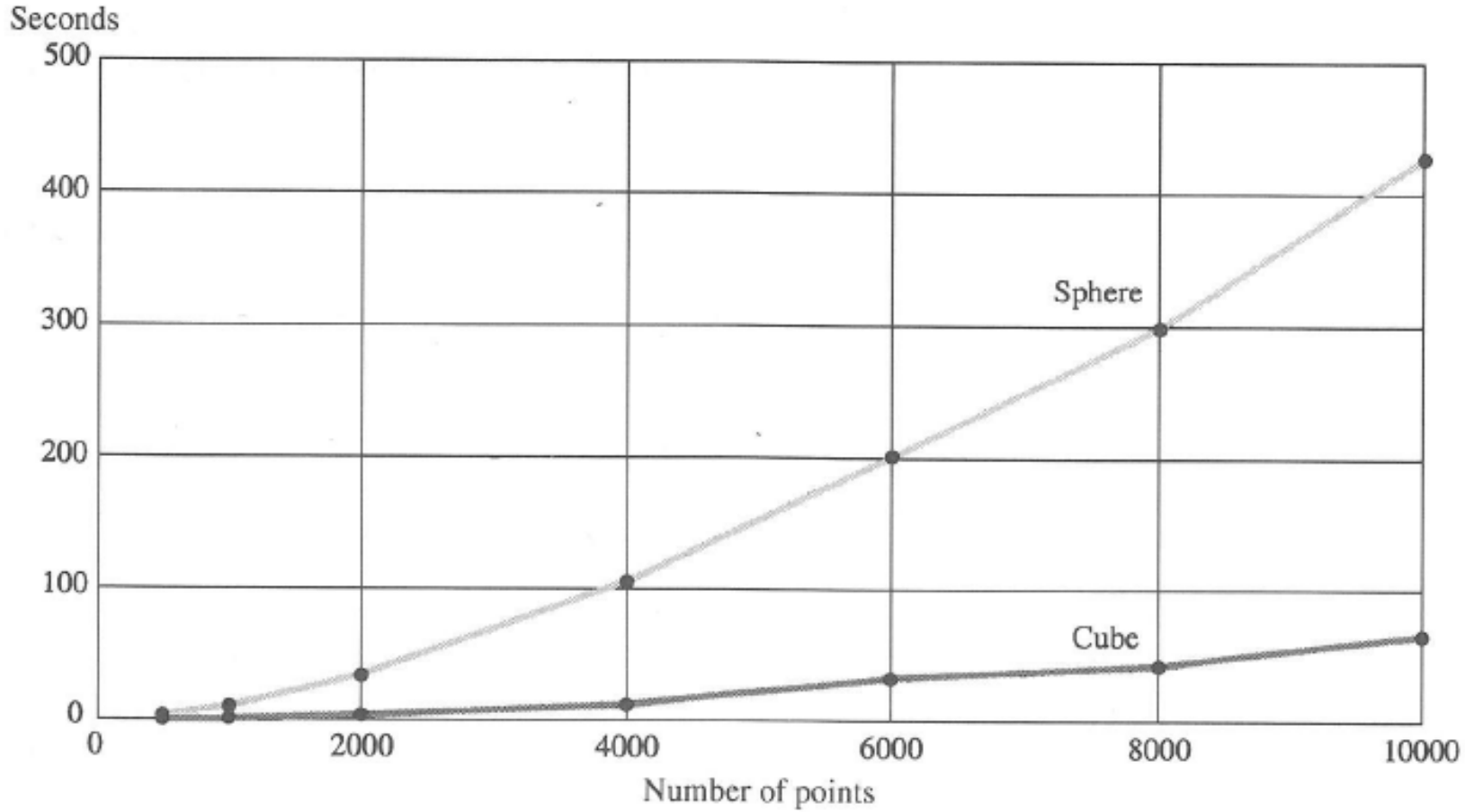


FIGURE 4.16 Runtimes for random points in a cube and near a sphere surface.

Volume Overflow

1. Computing the volume with integer arithmetic is not guaranteed to give the correct result, due to the possibility of overflow
2. When a computation exceeds the possible range, the C program proceeds without a complaint
3. (unlike division by zero, integer overflow is not detected and reported back to the C program)
4. This does not affect many programs, because the numbers used never become very large

Volume Overflow

1. But, our critical volume computation multiplies three coordinates together.
2. See fully expanded determinant in Equation 4.6:

$$
\begin{aligned}
&-b_x c_y d_z + a_x c_y d_z + b_y c_x d_z - a_y c_x d_z - a_x b_y d_z \\
&+a_y b_x d_z + b_x c_z d_y - a_x c_z d_y - b_z c_x d_y + a_z c_x d_y \\
&+a_x b_z d_y - a_z b_x d_y - b_y c_z d_x + a_y c_z d_x + b_z c_y d_x \\
&-a_z c_y d_x - a_y b_z d_x + a_z b_y d_x + a_x b_y c_z - a_y b_x c_z \\
&-a_x b_z c_y + a_z b_x c_y + a_y b_z c_x - a_z b_y c_x .
\end{aligned}
\tag{4.8}
$$

Volume Overflow

1. Let us explore the “safe range” of this computation
2. This is not an easy question to anwser
3. The smallest example on which I could make the computation err uses coordinates of only ± 512
4. The idea behind this example is that a regular tetrahedron maximizes its volume among all tetrahedra with fixed maximum edge length
5. So start with the regular tetrahedron T defined by (1,1,1),(1,-1,-1),(-1,1,-1), and (-1,-1,1), which is formed by four vertices of a cube centered on the origin.

Volume Overflow

1. Scaled by a constant c, the volume of this tetrahedron is $16c^3$. with $c = 2^9 = 512$, the volume is $2^{3(9)+4} = 2\ 31$ thus

$$\begin{vmatrix} 512 & 512 & 512 & 1 \\ 512 & -512 & -512 & 1 \\ -512 & 512 & -512 & 1 \\ -512 & -512 & 512 & 1 \end{vmatrix} = 2^{31} = 2147483648. \qquad (4.9)$$

2. However, evaluating Equation (4.9) results in the value $2147483648 = -2^{31}$
3. Fortunately there is a way to extend the safe range of the computation on contemporary machines without much additional effort
4. Most machines allocate doubles 64bits, over 50 of which are used for the mantissa.
5. So, integer calculations can be performed more accurately with floating-point numbers.

Volume Overflow

1. Several coping strategies:
    1. Report arithmetic overflow.
        1. This does not extend the range of the code, but at least the user will know when it fails
    2. Use higher precision arithmetic
        1. Machines are now offering 64-bit integer computations
    3. Use bignums
        1. LISP and Mathematica use arbitrary precision arithmetic, often called “bignums”
    4. Incorporate special determinant code

Volume Overflow

1. The VolumeSign code needs to be made: It returns only the sign of the volume, not the volume itself for the visibility tests
2. It make sense to use an AreaSign function paralleling the VolumeSign function
3. Code 4.23 is used throughout the code distributed with this book wherever only the sign of the area is needed
4. Note how integers are forced to doubles so that the multiplication has more bits available to it

Volume Overflow

```
int     AreaSign( tPointi a, tPointi b, tPointi c )
{
    double area2;

    area2=( b[0] - a[0] ) * (double)( c[1] - a[1] ) -
          ( c[0] - a[0] ) * (double)( b[1] - a[1] );

   /* The area should be an integer. */
    if        ( area2 > 0.5 )     return 1;
    else if   ( area2 < -0.5 )    return -1;
    else                          return 0;
}
```

Code 4.23 AreaSign.