Appendix

A.1 System Architecture

Virtually all software developed for this dissertation was created within *Cedar*, an integrated programming environment (*IPE*) developed at the Xerox Palo Alto Research Center. Although originally developed as the native operating system for the Xerox Dorado computer, Cedar presently operates atop a Unix platform. The illustration below depicts the upper level layering of Cedar interfaces that support the three-dimensional geometric modeler used to implement the concepts described in this dissertation. Because Cedar is an IPE (by virtue of shared runtime memory), the following modules easily communicate with each other.

We believe that the integrated nature of Cedar facilitated the development of the large, complex, and reliable hierarchy depicted below. Other software organization of geometric modeling systems are described elsewhere (see, for example, chapter 1 of [Badler 1991] or the appendix of [Prusinkiewicz and Lindenmayer 1990]).

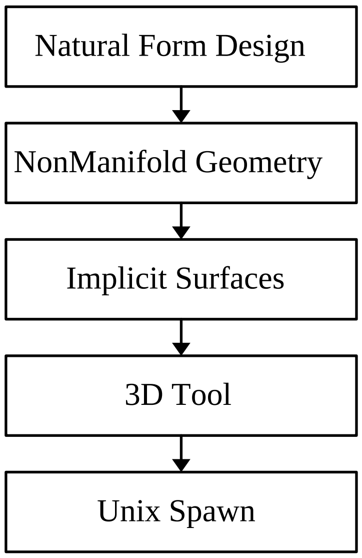

Figure A.1 Software Module Hierarchy

We briefly mention the method of rendering used for many of the shaded images in this dissertation. Cedar provides its own window system, so that access to the pixel raster must proceed through a Cedar interface. The images, however, were generated by RenderMan®, which runs as a Unix process. To allow RenderMan to display its results via Cedar, a C-language module is loaded with RenderMan.[1] A Cedar geometric display module, '3D tool,' invokes Renderman via a Unix spawning mechanism and transmits RIB (the RenderMan Interface Bytestream) data. Renderman responds by sending pixels to the C process, which transfers the pixels to a stream monitored by 3D Tool, which displays them on the Cedar raster.

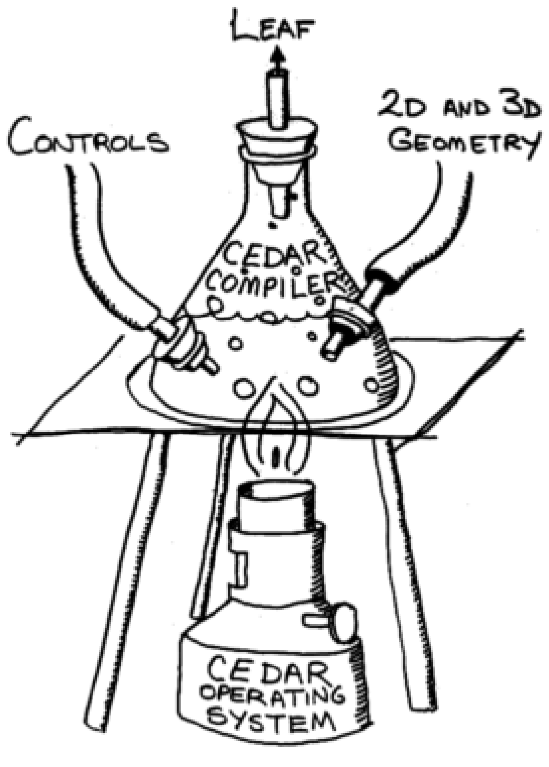

Figure A.2. An Alternate View of Software Module Hierarchy

by Andrew Glassner, 1986

A.2 An Implementation of the Combination Surface

Software developed for this dissertation was written in *Mesa*, an imperative programming language developed by Xerox Corp. It features shared memory and enforced type safety. We believe it is more readable than a corresponding version written in *C*.

To facilitate additional research, we present an implementation of the combination surface, given in pseudo-*Mesa*. The principal difficulty in converting the following to *C* is that *C* does not permit the nesting of procedures.

Library procedures, when called, are in bold-face. Variable types are italicized; language keywords (such as RETURN, IF, THEN, FOR, and WHILE) are printed in small-cap. Comments are italicized. We have not always distinguished record from pointer, but the difference should be clear from context. ***** indicates pointer, as in C. A real is a 32-bit floating point number; an int is a 32 bit integer. Records are defined by *Typename: TYPE ~ RECORD [recordbody]*; procedures are defined by *Procedurename (parameterlist) ~ {procedurebody}*. Assignment is denoted by *variable* ⇐ *expression*.

Looping is specified either by:

```
FOR variable IN [begin..end)
    loops variable through the values begin, begin+1, begin+2, ... end–1
FOR variable ⇐ initial, variable ⇐ next WHILE condition
    sets variable to initial value first time through, subsequently sets
    it to the expression next, continuing while condition is true
```

Types and Constants

The skeleton for the ‘starfish’ model (figures 5.71 through 5.73) consists of a parent limb and four child limbs. The endpoints of each limb are referred to as $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. The radii associated with the endpoints are r_1 and r_2. When the limb is in isolation, these points will be on the convolution surface. As noted in section 5.6.11, the union surface extends beyond these points. In order that the free ends of the union and convolution surfaces agree, we redefine the segment endpoints as

eUnion$_1$ and ***eUnion***$_2$, moved towards the segment interior by the amounts r_1 and r_2, respectively. The plane normals $\boldsymbol{n}_1$ and $\boldsymbol{n}_2$ represent the normals to the planes that approximate the joints at the limb endpoints.

The skeleton could be represented as a simple list of limbs, rather than a tree structure; indeed, the only use in this implementation of the *parent* and *children* fields is during initialization of each limb. The tree structure is the logical relative to a biological skeleton, however, and lends itself well to interactive design; thus, we retain it as a representation for the skeleton. This necessitates the use of a *callback procedure* when processing the entire skeleton. Callbacks are passed to **ApplyToTree**, a library procedure that applies the callback to the root of the skeleton and, recursively, to its children. Each node has a single parent, with the exception of the single root, which has no parent. Roots of a botanical tree should be represented as child limbs of the logical root.

```
Limb: TYPE ~ RECORD [
                 Point e1, e2,                -- segment endpoints
                 Point eUnion1, eUnion2,      -- union segment endpoints
                 Point n1, n2,                -- plane normals at endpoints
                 Real r1, r2,                 -- radii at endpoints
                 Limb parent, *children];     -- parent limb, list of child limbs
```

A repeated computation during the evaluation of the implicit surface function is the calculation of distance from $\boldsymbol{p}$ to the nearest point on a limb; this is stored in the ***Near*** record as ***point***. In addition, we store the ***projection*** of $\boldsymbol{p}$ onto the line of the limb segment and we store *alpha* such that ***point*** = $\boldsymbol{e}_1+alpha(\boldsymbol{e}_2-\boldsymbol{e}_1)$. Subsequently, *alpha* may be used to compute the radius and plane normal at ***point***.

```
Near: TYPE ~ RECORD [                         -- info about a point's proximity to a limb
                 Point point,                 -- point on limb nearest to query point
                 Point projection,            -- nearest point on line (not segment) containing limb
                 Real alpha,                  -- nearest point is alpha between limb endpoints
                 Real radius];                -- limb radius at nearest point
```

The following constants and variables are global to the implementation.

```
Real pi = 3.1415926535;
Int integralTableSize ⇐ 10000;          -- number of entries in Gauss integration table
Real *integralTable;                    -- Gauss integration table
Real size, delta;                       -- polygonization step size, gradient step size
Limb *joint;                            -- the skeleton
Shape shape;                            -- resulting surface
```

Skeleton Creation for ‘Starfish’ Joint

The following procedure creates the skeleton used in section 5.6.14.5.

```
Limb *MakeJoint ~ {
    SetLimb (Limb *limb) ~ {
        move union ‘endpoints’ towards the center of the segment:
        limb.eUnion1 ⇐ IF limb.parent=NIL
            THEN Add(limb.e1, SetLength(Sub(limb.e2, limb.e1), limb.r1))
            ELSE limb.e1;
        limb.eUnion2 ⇐ IF limb.children=NIL
            THEN Add(limb.e2, SetLength(Sub(limb.e1, limb.e2), limb.r2))
            ELSE limb.e2;
        compute the joint plane normals:
            IF limb.parent = NIL AND limb.children = NIL THEN RETURN;
            IF limb.children = NIL
                THEN limb.n2 ⇐ limb.parent.n2 -- e2 free endpoint, so plane normal is that of e1
                ELSE {                     -- not free, so plane normal depends on e1 and child e2:
                    Point *pts ⇐ AddPointToList(NIL, limb.e1);
                    ignore limb.e2 because it is internal to (not on the perimeter of) the polygon
                    FOR Limb *list ⇐ limb.children, list ⇐ list.next WHILE list # NIL
                        pts ⇐ AddPointToList(pts, list.data.e2);
                    limb.n2 ⇐ NormalFromPolygon(pts);
                    }
            limb.n1 ⇐ IF limb.parent = NIL THEN limb.n2 ELSE limb.parent.n2;
        }
    create a skeleton consisting of a single joint: one parent and four children, each with radius .1:
        Limb *joint ⇐ AllocateLimb((−.5, −.21, 0), (0, 0, 0), .1, .1, NIL);
        joint.children ⇐ ListLimb(
            AllocateLimb(joint.e2, (.06, .6, 0), .1, .1, joint)
            AllocateLimb(joint.e2, (.6,  .4, 0), .1, .1, joint)
            AllocateLimb(joint.e2, (.7, -.2, 0), .1, .1, joint),
            AllocateLimb(joint.e2, (.1, -.4, 0), .1, .1, joint));
    set various fields of each limb:
        ApplyToTree(joint, SetLimb);
    RETURN(joint);
    }
```

Implicit Surface Definition

The following procedures implicitly define the combination surface.

```
Real Subtend (Near near, Point e1, e2, Real radius) ~ {
    return area of filter subtended by segment e1e2, normalized by radius:
    integrating filter is centered at near.projection
        Real e1distance ⇐ −Distance(near.projection, e1);
        Real e2distance ⇐  Distance(near.projection, e2);
    if projection external to segment, reverse corresponding distance:
        IF near.alpha < 0 THEN e1distance ⇐ −e1distance;
        IF near.alpha > 1 THEN e2distance ⇐ −e2distance;
    return approximation to definite integral:
        RETURN(GaussIntegral(e1distance/radius, e2distance/radius));
    }

Real Convexity (Point p, Limb limb) ~ {
    return nonnegative number representing convexity: 0 is concave, 1 is convex
    compute plane normal as interpolation of endpoint normals:
        Real alpha ⇐ NearestToLimb(limb.e1, limb.e2, p).alpha;
        Point planeNormal ⇐ Unit(Interp(alpha, limb.n1, limb.n2));
    compute gradient at point p:
        Point gradient ⇐ Gradient(p, ConvolveValue, delta);
    compute angle between gradient and plane normal as measure of convexity:
        Real angle ⇐ ArcCos(Abs(Dot(planeNormal, gradient)));
        RETURN(1−Ease(angle/(pi/2));
    }

Real ConvolveValue (Real point) ~ {  -- this is f(p) for the convolution surface
    ConvolveLimb (Limb limb) ~ {
        compute convolution as product of 1D integration and 2D filter:
            Near near ⇐ NearestToLimb(limb.e1, limb.e2, point);
            value ⇐ value+
                Subtend(near, limb.e1, limb.e2, near.radius)* -- integration filter
                Gauss(Distance(point, near.projection)/near.radius); -- distance filter
        }
    compute convolution value as sum of convolution of each limb in skeleton:
        Real value ⇐ 0;
        ApplyToTree(joint, ConvolveLimb);
        RETURN(value);
    }

Real JointValue (Point p) ~ {  -- this is f(p) for the combination surface
    compute implicit surface function value as a combination of union and convolution surfaces:
    LimbValues (Limb limb) ~ {
        compute nearest for limb segment and for union segment (might be the same):
            Near near ⇐ NearestToLimb(limb.e1, limb.e2, p);
            Point nearUnionPoint ⇐
                NearestToLimb(limb.eUnion1, limb.eUnion2, point).point;
        compute value for union surface:
            Real temp ⇐ Gauss(Distance(p, nearUnionPoint)/near.radius);
            IF temp > unionValue THEN {unionValue ⇐ temp; nearest ⇐ limb}
        compute value for convolution surface:
            convolutionValue ⇐ convolutionValue+
                Subtend(near, limb.e1, limb.e2, near.radius)*
                Gauss(Distance(p, near.projection)/near.radius);
        }
    initialize variables and compute values for each limb:
        Limb nearest;
        Real convolutionValue ⇐ 0, unionValue ⇐ 0;
        ApplyToTree(joint, LimbValues);
    convexity controls combination of values:
        RETURN(convolutionValue+Convexity(p, nearest)*(unionValue−convolutionValue));
    }
```

Surface Normals

The following procedure computes the surface normal at a given point ***p***. The normal is ∇f, where ∇ is approximated by differencing; delta is the step size.

```
Point Normal (Point p, Real delta) ~ {
    Point UnionNormal (Point p) ~ {
        to avoid discontinuity in the union surface normal, compute union surface normal
        as a weighted sum of normals, one from each limb:
        Point LimbNormal (Limb limb) ~ {
            compute unweighted union surface normal:
            RETURN(Unit(Sub(p, NearestToLimb(limb.eUnion1, limb.eUnion2, p).point)));
            }
        FindNearestLimb (Limb limb) ~ {
            if p on or virtually on limb, set found true
            Real weight ⇐ WeightOfLimb(limb);
            IF weight > .9999 THEN found ⇐ TRUE;
            IF weight > maxWeight THEN {maxWeight ⇐ weight; nearestLimb ⇐ limb}
            }
        Real WeightOfLimb (Limb limb) ~ {
            return weight for this limb; 1 if on union surface, decreases away from surface:
                Near near ⇐ NearestToLimb(limb.eUnion1, limb.eUnion2, p);
                RETURN(2*Gauss(Length(Sub(p, near.point))/near.radius));
            }
        WeightNormal (Limb limb) ~ {
            weight limb’s contribution to normal by product of 1 minus other limb’s weight:
            Real weight ⇐ 1.0;
            determine weight:
                Scale (Limb ll) ~ {
                    IF limb # ll THEN weight ⇐ weight*Ease(1−WeightOfLimb(ll));
                    }
                ApplyToTree(joint, Scale);
            accumulate:
                normal ⇐ Add(normal, Mul(LimbNormal(limb), weight);
            }
        SumClose (Limb limb) ~ {
            ignore limb if p not on surface; otherwise, accumulate weighted limb normal:
                Real weight ⇐ WeightOfLimb(limb);
                IF weight > .9999
                    THEN normal ⇐ Add(normal, Mul(LimbNormal(limb), weight));
            }
        Bool found ⇐ FALSE;
        Real maxWeight ⇐ 0;
        Point normal ⇐ (0, 0, 0);
        find nearest limb:
            ApplyToTree(joint, FindNearestLimb);
        compute normal:
            IF found
                THEN ApplyToTree(joint, SumClose)
                    p on surface, ignore limbs not containing p
                ELSE ApplyToTree(joint, WeightNormal);
                    p off surface, sum weighted normal from each limb
        RETURN(Unit(normal));
        }
    compute normal for union surface and set nearestLimb:
        Limb nearestLimb;
        Point unionNormal ⇐ UnionNormal(p);
    compute normal for convolution surface:
        Point convolutionNormal ⇐ Gradient(p, ConvolveValue, delta);
    combine convolution normal and union normal according to convexity:
        Real convexity ⇐ Convexity(p, nearestLimb);
        RETURN(Unit(Interp(convolutionNormal, unionNormal, convexity)));
    }
```

Execution

The conversion of the implicit definition of the combination surface to polygons is accomplished by the following procedure, in which the error function is approximated and stored in a table, the skeleton is created, the implicit surface is polygonized, and vertex normals are computed.

```
Main ~ {
    initialize integration table, set polygonization and gradient step sizes
        MakeGaussTable();
        delta ⇐ 0.01*(size ⇐ 0.025);
    create the skeleton:
        joint ⇐ MakeJoint();
    polygonize the surface:
        shape ⇐ Polygonize(JointValue, size);
    compute the surface normals:
        FOR i IN [0..vertices.length)
            shape.vertices[i].normal ⇐ Normal(shape.vertices[i].point, delta);
    }
```

Low-Level Functions

***Real* Abs** (*Real* r)
Return the absolute value of r

***Real* Min** (*Real* r1, r2)
Return the minimum of r1, r2

***Real* Max** (*Real* r1, r2)
Return the maximum of r1, r2

***Int* Floor** (*Real* x)
Return the greatest integer less than x

Real SqRt *(Real x)*
Return the square root of x

Real Exp *(Real x)*
Return e to the power x

***Real* ArcCos** (*Real* cosine)
Return the angle (in radians) of the given cosine

***Point* Sub** (*Point* v1, v2)
Return the 3D vector subtraction: v1-v2

***Point* Add** (*Point* v1, v2)
Return the 3D vector addition: v1+v2

***Point* Mul** (*Point* v, *Real* s)
Return the 3D vector v scaled by s

***Point* Unit** (*Point* v)
Return the 3D vector v normalized to unit length

***Real* Length** (*Point* v)
Return the length of the vector

***Point* SetLength** (*Point* v, *Real* length)
Scale and return the 3D vector v so that it has the specified length

***Real* Distance** (*Point* p1, p2)
Return the distance between the two points

***Real* Dot** (*Point* v1, v2)
Return the dot product of the two vectors

***Point* Interp** (*Real t*, Point v1, v2)
Return the interpolated 3D vector = v1+t(v2-v1)

***Point* NormalFromPolygon** (*Point* *polygon)
Return the 3D vector normal to the (approximately flat) polygon

***Near* NearestToLimb** (*Point* e1, e2, p)

```
    Determine the point on limb (e1, e2) closest to the point p

NearAcc NearnessAccelerator (Point e1, e2)
    Compute terms to accelerate future NearestToLimb evaluations

Point *AllocateReals (Int n)
    Return a pointer to n reals

Point *AddPointToList (Point *pts, Point p)
    Add p to a sequence of points
```

Gaussian Evaluation

```
Real standardDeviation ⇐ 0.69314715;
    width coefficient (to Cedar precision) such that Gauss(1) = ½
    actual value between .6931470 and .6931473)

Real Gauss (Real x) ~ {RETURN(Exp(−x*x*standardDeviation))}
    Returns 1 for x <= 0 and 1/2 at x = 1

Real GaussIntegral (Real x1, x2) ~ {
    return area under curve of Gaussian, scaled to have integral 1
    Real Area (Real x) ~ { -- area from -∞ to x
        Real AreaForPosX (Real x) ~ { -- area from -∞ to x, x > 0
            Real scaledX ⇐ x*integralTableScale; -- fit x to table domain
            Int tableIndex ⇐ Min(integralTableSize−1, Floor(scaledX));
            Real area ⇐ integralTable[tableIndex];
            IF index < integralTableSize−1 THEN -- linear interpolation:
                area ⇐ area+(scaledX−tableIndex)*(integralTable[tableIndex+1]−area);
            RETURN(area);
            };
        RETURN(IF x < 0 THEN 1−AreaForPosX(−x) ELSE AreaForPosX(x));
        };
    RETURN(Area(Max(x1, x2))−Area(Min(x1, x2)));
    }

MakeGaussTable ~ {
    integralTable ← AllocateReals(integralTableSize);
    Real integralTableXRange ⇐ 10; -- Gauss(6) = 0.000000000 (Cedar accuracy)
    Real integralTableScale ⇐ integralTableSize/integralTableXRange;
    Real scale ⇐ .5/(SqRt(pi/standardDeviation)*integralTableScale);
    Real value1 ⇐ Gauss(0);
    integralTable[0] ⇐ .5; -- area from x = -inf to x = 0
    FOR i IN [1..integralTableSize) { -- create table from x = 0 to tableXRange
        Real value2 ⇐ Gauss(i/integralTableScale);
        integralTable[i] ⇐ integralTable[i−1]+scale*(value1+value2);
        value1 ⇐ value2;
        }
    }
```

Miscellany

```
Limb *AllocateLimb (Point e1, e2, Real r1, r2, Limb *parent)
    Return a pointer to a limb, and assign its endpoints, radii, and parent

Limb *ListLimb (Limb *l1, *l2, *l3 . . .)
    Return a pointer to a list of limbs

Shape Polygonize (Real function (Point p), Real size)
    Return a shape given an implicit surface function and a polygonization step size

Point Gradient (Point p, Real function (Point p), Real delta) ~ {
    Return the gradient of the function at the point p, using delta as the approximating step
    Real value ← function(p);
    Real x ← value−function((p.x+delta, p.y, p.z));
    Real y ← value−function((p.x, p.y+delta, p.z));
    Real z ← value−function((p.x, p.y, p.z+delta));
    RETURN(Unit((x, y, z)));
    }

ApplyToTree (Limb *root, Proc action (Limb *l)) ~ {
    Apply the procedure ‘action’ to root and its descendents
    Inner: (Limb *l) ~ {
        action(l);
        FOR List of *Limb list ⇐ l.children, list.next WHILE list # NIL {
            IF list.data # NIL THEN Inner(list.data);
            }
        }
    IF root # NIL THEN Inner(root);
    }

Real Ease (Real x) ~ {
    Ease in to and out of a particular interpolation.
    Ease(0) = 0, Ease(1) = 1, and Ease’(0) = Ease’(1) = 0
    A cubic, sigmoidal function based on the Wyvill kernel, in section 9.5.10
    IF x < 0 RETURN(0);
    IF x > 1 RETURN(1);
    RETURN(($4x^6−17x^4+22x^2$)/9);
    }
```

A.3 An Implementation of Polygonal Convolution

```
SetGeometry: PROC [Primitive p] ~ {
    PairsFromProjection: PROC RETURNS [PairList list] ~ {
        FOR n: INT IN [0..p.nVertices) DO
            q: Triple ← GetPoint[p, n];
            t: Triple ← ProjectPointToPlane[q, pln];
            pp: Pair ← [Dot[t, xAxis], Dot[t, yAxis]];
            list ← CONS[[Dot[t, xAxis], Dot[t, yAxis]], list];
            ENDLOOP;
    };
normal: Triple ← p.normal ← PolygonNormal[p.points, p.indices, TRUE];
pln: Plane ← FromPointAndNormal[GetPoint[p, 0], normal];
plane: Quad ← p.plane ← [pln.x, pln.y, pln.z, pln.w];
xAxis: Triple ← p.xAxis ← IF p.twist.tw0 # 0 OR p.twist.tw1 # 0    -- x axis in the plane
    THEN Unit[Sub[p.twist.p1, p.twist.p0]]
    ELSE Ortho[normal];
yAxis: Triple ← p.yAxis ← Cross[xAxis, normal];
origin: Triple ← Mul[normal, -p.plane.w];            -- origin of plane
mm: Box ← MinMaxOfPairs[PairsFromProjection[]];       -- 2d image bounds
min: Pair ← Sub[mm.min, [2*p.extent, 2*p.extent]];     -- extent margin
p.size ← Add[[4*p.extent, 4*p.extent], G2dVector.Sub[mm.max, mm.min]];
p.worldToImageScale ← MIN[REAL[p.res.x]/p.size.x, REAL[p.res.y]/p.size.y];
p.origin ← Add[origin, Combine[xAxis, min.x, yAxis, min.y]];
```

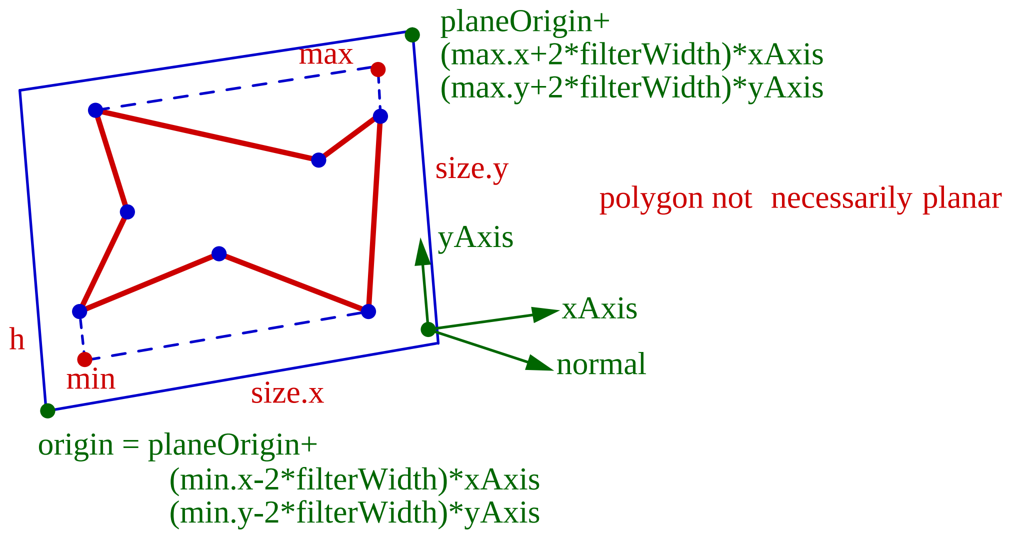
planeOrigin+
(max.x+2*filterWidth)*xAxis
(max.y+2*filterWidth)*yAxis
max
size.y
polygon not necessarily planar
yAxis
xAxis
normal
h
min
size.x
origin = planeOrigin+
(min.x-2*filterWidth)*xAxis
(min.y-2*filterWidth)*yAxis

Figure A.3. A Polygonal Skeletal Element

```
};
p.twoOverExtent ← 2.0/p.extent;
p.filter.support ← p.worldToImageScale*p.extent;
p.bounds ← GetBounds[p];
};
```

A.4 Notes

1. This module was implemented by Andrew Glassner.