In the 1960s, Steve Coons developed a family of parametric surfaces that we today call Coons patches (Coons 1967, Forrest 1972, Böhm et al. 1984), (Rogers and Adams 1990, pp. 422-425). They are defined by boundary curves around the perimeter of the patch and blending functions that are used to interpolate a surface between the boundary curves. Coons patches are very general; they are a superset of the more widely used tensor product surfaces.

The bilinear Coons patch is the class of Coons patches obtained when there are four boundary curves and the blending functions are linear. Suppose that we want to create a surface \( p(u, v) \) over the unit square that interpolates the four boundary curves \( p_{00}(u), p_{11}(u), p_{01}(v), \) and \( p_{10}(v) \), which correspond to the bottom, top, left, and right of the
VIII.2 Bilinear Coons Patch Image Warping

Figure 2. The sum of a surface lofted in the \(v\) direction between the bottom and top curves (a), plus a surface lofted in the \(u\) direction between the left and right curves (b), minus a bilinear surface through the four corner points (c), yields a bilinear Coons patch (d). This is a plot of \(x\) as a function of \((u, v)\); \(y\) and \(z\) are similar.

square, respectively (Figure 1). That is, we want \(p_{u0}(u) = p(u, 0), p_{u1}(u) = p(u, 1), p_{v0}(v) = p(0, v), \) and \(p_{v1}(v) = p(1, v)\) for all \(u\) and \(v\). For surface modeling applications, \(p\) will typically be a 3-vector: \(p(u, v) = (p_x(u, v), p_y(u, v), p_z(u, v))\), but the Coons patch can be generalized to arbitrary dimensions.

We can easily construct a surface that interpolates the bottom and top boundary curves by lofting: linear interpolation in \(v\) between corresponding points of the two curves yields the surface \((1 - v)p_{u0}(u) + vp_{u1}(u)\), shown in Figure 2(a). We can likewise create a surface that interpolates the left and right boundary curves: \((1 - u)p_{v0}(v) + up_{v1}(v)\), seen in Figure 2(b). Can these two be combined to yield a surface that interpolates all four curves? Coons observed that if we add these two surfaces, then subtract a bilinear surface through the four corner points, the resulting surface does just that! The bilinear surface, shown in Figure 2(c), is defined by \((1 - u)(1 - v)p_{00} + u(1 - v)p_{10} + (1 - u)vp_{01} + uvp_{11}\).

The only condition on the boundary curves to guarantee interpolation is that endpoints of the corner curves are coincident, that is, \(p_{00} = p_{u0}(0) = p_{v0}(0), p_{10} = p_{u0}(1) = p_{v0}(1), p_{01} = p_{u1}(0) = p_{v1}(0), \) and \(p_{11} = p_{u1}(1) = p_{v1}(1)\). Other than this restriction, the boundary curves can take any shape whatsoever.

If the endpoints of the corner curves are not coincident, then the surface will not interpolate them, in general, but a useful surface can still be constructed by taking the corner points to be the midpoints of the corresponding endpoints: \(p_{00} = (p_{u0}(0) + p_{v0}(0))/2, \) etc.

The formula for a bilinear Coons patch is thus:

\[
p(u, v) = (1 - v)p_{u0}(u) + vp_{u1}(u) \\
+ (1 - u)p_{v0}(v) + up_{v1}(v) \\
- (1 - u)(1 - v)p_{00} - u(1 - v)p_{10} - (1 - u)vp_{01} - uvp_{11} \tag{1}
\]
Bilinear Coons Image Warp

In the late 1970s, Lance Williams at the New York Institute of Technology (NYIT) Computer Graphics Lab employed the bilinear Coons patch for image warping. In image warping, we are given a source image and a 2D-to-2D mapping between the source image space and the destination image space, and we wish to compute the destination image. The most straightforward warping algorithm simply loops over each destination pixel, computes the corresponding source pixel, and copies it to the destination image. The mapping between source and destination spaces can be specified by either the forward mapping from source space to destination space or its inverse. The straightforward warping algorithm requires the inverse mapping.

The bilinear Coons patch, as defined above, is a mapping from 2D to an arbitrary number, \( n \), of dimensions. For image warping, we take the special case \( n = 2 \). Since the bilinear Coons patch defines a mapping that is very difficult to invert, in general, we will use it directly as the inverse mapping from destination space to source space. For that reason, we use coordinates \((x, y)\) for the source image and \((u, v)\) for the destination image.

Williams's idea was to allow a user to draw four boundary curves on the source image with a tablet, mouse, or other input device and use the bilinear Coons patch to warp the region bounded by those curves into a rectangle. If the rectangle has corners \((u_0, v_0)\), width \( nu + 1 \), and height \( nv + 1 \), then the warp is performed by scanning the destination image like so:

\[
\text{for } iv \leftarrow 0 \text{ to } nv \\
\quad \text{for } iu \leftarrow 0 \text{ to } nu \\
\quad \quad (x, y) \leftarrow p(iu/nu, iv/nv) \\
\quad \quad \text{dest}[uO + iu, vO + iv] \leftarrow \text{source}[x, y]
\]

If higher quality results are desired, filtering can be used instead of simply copying the nearest pixel.

The bilinear Coons patch image warp, or bilinear Coons warp for short, is a very flexible transformation. The boundary curves are arbitrary; they can be linear or higher degree polynomials; they can be defined by one function or piecewise; they can cross themselves; they can be roughly arc-length parameterized (constant speed) or not; and they can be defined by an array of points. Most transformations that are used for image warping, including piecewise affine, bilinear, biquadratic, or bicubic mapping (Wolberg 1990), are defined by a much smaller number of control points, giving the user less control over the shape of the boundary. On the other hand, the bilinear Coons patch...
VIII.2 Bilinear Coons Patch Image Warping

Figure 3. Bilinear Coons patches in source space with \( u \) and \( v \) isoparametric curves: (a) a bilinear patch, (b) a quadrilateral with non–arc-length-parameterized boundary curves, (c) a circle, (d) a concave region, (e) a wiggly region exhibiting fold-over. The four boundary curves are shown in bold, and corners are shown as white dots.

A warp does not allow the user direct control over the interior of the patch, as does a bicubic tensor product patch, for instance.

Figure 3 shows examples of the deformations that are possible with the bilinear Coons warp.

\[\text{\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{fig3.png}
\end{center}
\caption{Bilinear Coons patches in source space with \( u \) and \( v \) isoparametric curves: (a) a bilinear patch, (b) a quadrilateral with non–arc-length-parameterized boundary curves, (c) a circle, (d) a concave region, (e) a wiggly region exhibiting fold-over. The four boundary curves are shown in bold, and corners are shown as white dots.}
\end{figure}\]

\]

\[\text{\textbf{Implementation}}\]

In an implementation, the boundary curves are best stored as arrays; this allows arbitrary curves to be accommodated quickly and easily. The bottom and top curves \( \mathbf{p}_u^0 \) and \( \mathbf{p}_u^1 \) should be resampled to create arrays whose length is the width of the destination image, \( nu+1 \), and the left and right curves \( \mathbf{p}_v^0 \) and \( \mathbf{p}_v^1 \) should be resampled to create arrays whose length is the height, \( nv+1 \). The following code implements arc-length-parameterized curve resampling, that is, successive points of the output curve are (approximately) equispaced. Arc-length-parameterized boundary curves yield smoother warps.

```c
typedef struct { /* 2D POINT OR VECTOR */
    float x, y;
} Point2f;

typedef struct { /* A CURVE DEFINED BY A SEQUENCE OF POINTS */
    int npt; /* number of points */
    Point2f *pt; /* array of npt points */
} Curve;

#include <assert.h>
#define ALLOC(ptr, type, n) assert(ptr = (type *)malloc((n)*sizeof(type)))

static double len(double x, double y) {return sqrt(x*x+y*y);}

/* resample: resample curve a to create curve b, with n points
 * allocates b->pt to have length n */
```
static void resample(Curve *a, Curve *b, int n) {
    int i;
    double step, l, d;
    Point2f *ap, *bp;

    assert(a->npt>=2);
    for (step=0., ap=a->pt, i=a->npt-1; i>0; i--, ap++)
        step += len(ap[1].x-ap[0].x, ap[1].y-ap[0].y);
    step /= n-1;  /* length of each output segment (ideally) */
    ALLOC(b->pt, Point2f, n);
    b->npt = n;
    d = .0001; /* = 0 + tolerance for round-off error */
    for (ap=a->pt, bp=b->pt, i=a->npt-1; i>0; i--, ap++) {
        l = len(ap[1].x-ap[0].x, ap[1].y-ap[0].y);
        d += l;
        /* d is the remaining length of the line segment from ap[0] to ap[1]
         * that needs to be subdivided into segments of length step */
        while (d>O.) {
            bp->x = ap[1].x - d/l*(ap[1].x-ap[0].x);
            bp->y = ap[1].y - d/l*(ap[1].y-ap[0].y);
            bp++;
            d -= step;
        }
    }
    assert(bp-b->pt == n); /* check that we made requested no. of pts. */
}

With the boundary curves stored in arrays, the Coons warp, if implemented directly
from Equation (1), requires 24 multiplications and 8 table lookups per destination pixel
(remember that p is a 2-vector). These counts do not include the cost of copying and filtering pixels.

These formulas can be optimized substantially, however. We can use forward differencing to incrementalize the terms that are linear in u, pulling six of the eight corner references out of the inner loop into the variable q:

for iu ← 0 to nu
    v ← iu/nu
    q ← p_{0v}[iu] - (1-v)p_{00} - vp_{01}
    dq ← (p_{1v}[iu] - p_{0v}[iu] - (1-v)(p_{10} - p_{00}) - v(p_{11} - p_{01}))/nu
for iu ← 0 to nu
    (x, y) ← p ← p_{u0}[iu] + v(p_{u1}[iu] - p_{u0}[iu]) + q
    dest[x0 + iu, y0 + iv] ← source[x, y]
    q ← q + dq

This leaves 2 multiplications and 4 array references per destination pixel.

Eliminating the last two multiplications (without precomputing everything) seems impossible until we observe that the sum of the first two terms of p, which we denote...
pu[iu] = (1−v)pu[0][iu]+vpu[1][iu], changes by a constant increment within any column of the destination image. If we precompute the array Δpu[iu] = (pu[1][iu]−pu[0][iu])/nv then pu can be updated from scan line to scan line by incrementing it by Δpu. In addition to these optimizations, we can switch from floating-point to integer arithmetic. The resulting inner loop computes the warp with just six adds, two shifts, and six indirect memory references.

An implementation in C is listed below. This code uses the previous formulas with one change: the comments refer to pu[0] as “top” and pu[1] as “bottom,” since most frame buffers have their y-coordinate pointing down. For maximum portability, the code calls generic routines pixel_read and pixel_write.

```c
#define SHIFT 20 /* number of fractional bits in fixed point coords */
#define SCALE (1<<SHIFT)

typedef struct {
  /* INTEGER POINT AND VECTOR */
  int px, py; /* position */
  int dx, dy; /* incremental displacement */
} ipoint;

/*
  * coons_warp: warps the picture in source image into a rectangular
  * destination image according to four boundary curves, using a
  * bilinear Coons patch.
  * bound[0] through bound[3] are the top, right, bottom, and left
  * boundary curves, respectively, clockwise from upper left.
  * (These comments are written assuming that y points down on your
  * frame buffer. Otherwise, bound should proceed CCW from lower left.)
  * The lengths of bound[0] and bound[2] are assumed to be the width of
  * the destination rectangle, and the lengths of bound[1] and bound[3]
  * are assumed to be the height of the rectangle.
  * The upper left corner of the destination rectangle is (u0,v0).
  *
  * Paul Heckbert 25 Feb 82, 15 Oct 93
  */

void coons_warp(Pic *source, Pic *dest, Curve *bound, int u0, int v0) {
  register ipoint *pu;
  register int u, x, y, qx, qy, dqx, dqy;
  int nu, nv, n, count;
  float du, dv, fv;
  Point2f p00, p01, p10, p11, *pu0, *pul, *pou, *plv;
  ipoint *pu;

  nu = bound[0].npt-1; /* nu = dest_width-1 */
  nv = bound[1].npt-1; /* nv = dest_height-1 */
  assert(bound[2].npt==nu+1);
  assert(bound[3].npt==nv+1);

  pu0 = &bound[0].pt[0]; /* top boundary curve */
  plv = &bound[1].pt[0]; /* right */
```

pul = &bound[2].pt[nu];       /* bottom */
pOv = &bound[3].pt[nv];       /* left */
/* arrays pul and pOv are in the reverse of the desired order, 
running from right to left and bottom to top, resp., so we 
index them with negative subscripts from their ends (yeeeha!) */

p00.x = (pOv[0].x + pu0[0].x)/2.;  /* upper left patch corner */
p00.y = (pOv[0].y + pu0[0].y)/2.;
p10.x = (pu0[nu].x + plv[0].x)/2.;  /* upper right */
p10.y = (pu0[nu].y + plv[0].y)/2.;
p11.x = (plv[nv].x + pUll-nul.x)/2.; /* lower right */
p11.y = (plv[nv].y + pUll-nul.y)/2.;
p01.x = (pul[0].x + pOv[-nu].x)/2.; /* lower left */
p01.y = (pul[0].y + pOv[-nu].y)/2.;

du = 1./nu;
dv = 1./nv;

ALLOC(pua, Ipoint, nu+1);
for (pu=pua, u=0; u<=nu; u++, pu++) {
    pu->dx = (pu[-u].x - pu0[u].x)*dvSCALE + .5;
    pu->dy = (pu[-u].y - pu0[u].y)*dvSCALE + .5;
    pu->px = pu0[u].x*SCALE + .5;
    pu->py = pu0[u].y*SCALE + .5;
}

count = 0;
for (fv=0., v=0.; v<=nv; v++, fv+=dv) {
    qx = (p0v[-v].x - (1.-fv)*p00.x - fv*p01.x + .5)*SCALE + .5;
    qy = (p0v[-v].y - (1.-fv)*p00.y - fv*p01.y + .5)*SCALE + .5;
    dqx = (plv[v].x - p0v[-v].x - (1.-fv)*(p10.x-p00.x) - fv*(p11.x-p01.x))
         *duSCALE + .5;
    dqy = (plv[v].y - p0v[-v].y - (1.-fv)*(p10.y-p00.y) - fv*(p11.y-p01.y))
         *duSCALE + .5;
    for (pu=pua, u=0; u<=nu; u++, pu++) {
        x = pu->px+qx >> SHIFT;
        y = pu->py+qy >> SHIFT;
        pixel_write(dest, uO+u, vO+v, pixel_read(source, x, y));
        qx += dqx;
        qy += dqy;
        pu->px += pu->dx;
        pu->py += pu->dy;
    }
}
free(pua);

On a Silicon Graphics workstation with a MIPS R4000 processor, the code above 
warps a 512 x 486 image in .25 seconds. The speedup relative to the unoptimized 
algorithm on this machine is a factor of 4; on other machines with relatively slower 
multipliers or slower floating point, the speedup would be greater.
This code can be modified to do filtering. As it is written, blockiness (rastering) results where the warp locally scales up the image; this could easily be improved using a bilinear reconstruction filter (Hill 1994). Eliminating aliasing where the warp locally scales down is more difficult.

Color plates VIII.2.1–4 show several examples of images created using the bilinear Coons warp. The boundary curves that generated them are shown in figure 4.

Acknowledgments and History

This work could not have been done without Lance Williams, who was the first to apply the Coons patch to image warping. Also instrumental in this work was another colleague at the NYIT Computer Graphics Lab, Tom Brigham, who in 1982 bet me a cheeseburger that I couldn’t double the speed of our Coons warp program. Before these optimizations, our program took two minutes to warp a 486-line image on a PDP 11/34. Afterward, it took 20 seconds. Settlement was made at the Landmark Diner.

Tom Brigham went on to use the program described here as a building block in his pioneering “morphing” work around 1983. In his approach, two images are decomposed into a mosaic of potentially overlapping regions bounded by closed curves, and a pair-wise correspondence is established from regions of one image to regions of the other. The regions and correspondence are specified interactively by the user/animator. To metamorphose a fraction $\alpha$ from one image to the other, Brigham warped the shapes and cross-dissolved the colors of each of the regions by this fraction $\alpha$, drawing them one by one into a frame buffer. Individual regions were warped by interpolating between Coons patches. The most difficult step in this process is inversion of a Coons warp; Brigham accomplished this with a large two-dimensional lookup table.

In more recent work, Pete Litwinowicz at Apple has employed meshes of Coons patches to animate textured regions as part of a keyframe animation system (Litwinowicz 1991).


