Evaluating Tensor Product and Triangular Bézier Surfaces

Computer Science Department
University of Waterloo
Research Report CS-95-22

Jeromy Carrière
j2carrie@cgl.uwaterloo.ca

Abstract

Many papers describe techniques for evaluating spline curves and surfaces. While each paper provides some theoretical or empirical evidence with which to compare techniques, there exist few global comparisons [Peters94]. Also, papers describing particular algorithms often provide few details, making implementation of the technique presented difficult or impossible. This report attempts to illuminate the performance relationships between, and implementations of, various methods for rendering spline surfaces. Empirical results are given for bicubic tensor product Bézier surfaces, and for cubic and quartic triangular Bézier surfaces.

1 Introduction

Many techniques exist for evaluating spline surfaces, in particular, for evaluating m-simplex (eg. triangular) Bézier surfaces, tensor product Bézier patches and tensor product B-spline surfaces. Such techniques include de Casteljau evaluation [Farin93, Mann95], forward differencing [Foley90], adaptive forward differencing [Lien87] and SV-nested multiplication [Schumaker86], as well as several varieties of recursive subdivision [Foley90, Peters94]. Each technique has associated implementation peculiarities as well as theoretical and empirical performance behaviour.

Examples of global comparisons of evaluation techniques include [Peters94] and [Williams88]. Peters presents, via pseudocode, several algorithms for the evaluation of Bézier surfaces of arbitrary degree over simplices of arbitrary dimension. The algorithms are compared on several metrics, including approximate number of lines of code for implementation and theoretical time complexity. Some consideration is also given to storage complexity and stability. A performance comparison is presented for surfaces of degree 2 to degree 14.

Williams discusses only tensor product B-spline surfaces, evaluated so as to produce a wire frame representation. The asymptotic behaviours of evaluation techniques are compared via an operation-counting metric. Williams also considers how one may choose an appropriate evaluation technique based on the number of evaluations performed for each B-spline segment.

This report will present an overview of the implementation of several evaluation techniques for m-simplex and tensor product Bézier patches, using C++-like pseudocode. Emphases will be placed on empirical performance behaviour and the description of difficult implementation details. The approach taken will be oriented toward producing tessellated (or polygonalized) surface representations suitable for rendering via a polygon pipeline. Thus, methods for scanline rendering such as that described by Rockwood [Rockwood87] will not be considered. Surfaces of low degree (bicubic for tensor product patches and cubic and quartic for m-simplex patches) will be used for empirical testing as they are most commonly used in practice, although some discussion of asymptotic behaviour will be included. Also, consideration will be given to computation of normals, if they are not produced automatically by the evaluation technique, to allow the resulting surfaces to be shaded.

The remainder of the report will comprise, for tensor product and triangular surfaces, a description of the techniques implemented, a discussion of the drawbacks and limitations of particular techniques, a comparison of the performance of the techniques on particular surfaces and a brief discussion of the asymptotic behaviour of some of the methods.

2 Preliminaries

2.1 Pseudocode

The various techniques implemented will be described via detailed pseudocode, derived from C++. Along with simple point, vector and matrix classes, two further classes support the pseudocode:

TPBezier This class implements a representation of a tensor product Bézier surface. Several member functions provide the class interface, including:

```
void SetControlPoint( int i, int j, Point ptControl_point )
Point GetControlPoint( int i, int j )
GetControlPoints( Matrix mAx, Matrix mAy, Matrix mAz )
SetControlPoints( Matrix mAx, Matrix mAy, Matrix mAz )
int GetM()
int GetN()
```

MSimpBezier This class implements a representation of m-simplicial domain Bézier patches. Member functions include:

```
void SetControlPoint( int aI[], Point ptControl_point )
Point GetControlPoint( int aI[] )
int GetDegree()
int GetDimension()
```

Because the domain of the patch can theoretically be any dimension, control points must be referenced by arbitrary dimension multiindices. However, as this paper is concerned explicitly with surfaces, only triangular patches will be considered, and control points can be referenced via two (independent) integers.

Lastly, in the implementation of adaptive forward differencing, a template list class is used to record points along curves.

2.2 Tessellation

Some of the techniques described below (for tensor product surfaces: bilinear interpolation, modified de Casteljau evaluation; for m-dimension simplex surfaces: de Castejau evaluation, SV-nested multiplication) evaluate surfaces at individual domain points. As we desire a polygonal representation of the surface, we must tessellate the domain in order to apply these methods. Tessellation can easily be accomplished by sampling the surface at particular rates in the parametric directions and joining adjacent points to form triangles.

Dividing the domain into quadrilaterals is another option, assuming that the rendering engine can correctly render non-planar polygons.

3 Tensor Product Surfaces

A tensor product surface is given by

$$T(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} B_i^n(u) B_j^m(v),$$

where B_i^n is the ith Bernstein polynomial of degree n, $P_{i,j}$ is a control point in $\Re^3$, and (u,v) is a point in a rectilinear domain space.

3.1 Bilinear Interpolation

The straightforward approach to evaluation of a tensor product Bézier surface is de Casteljau evaluation in both parametric directions simultaneously, as shown in Figure 1 [Farin93]. This figure originally appeared in [Mann95].

3.1.1 Pseudocode

```
Point Compute( TPBezier& tpbezSurface, double dU, dV, Vector& vecdFdU, Vector& vecdFdV ) {

    int i, j, iR;
    int iDegree = tpbezSurface.GetDegree(); // assume m x m
```

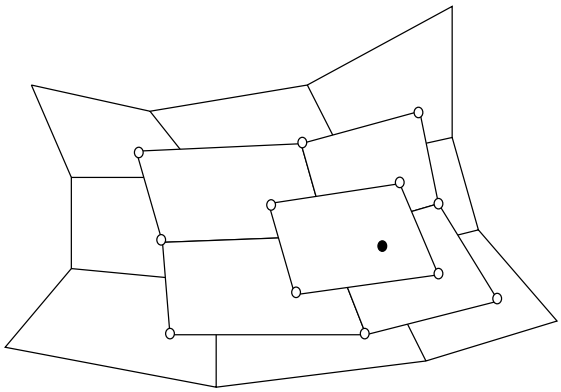

Figure 1: Bilinear Interpolation

```
  TPBezier* ptpbezTemp = new TPBezier( tpbezSurface );

  // for each degree down to 1 - bilinear patch
  for( iR = 1; iR < iDegree; iR++ ) {

    // for each point
    for( i=0; i <= iDegree-iR; i++ ) {
      for( j=0; j <= iDegree-iR; j++ ) {

        // perform interpolation
        ptpbezTemp->SetControlPoint( i, j,
                        (ptpbezTemp->GetControlPoint( i, j )*(1.0-dU)
                         + (ptpbezTemp->GetControlPoint( i+1, j )*dU))*(1.0-dV) +
                        (ptpbezTemp->GetControlPoint( i, j+1 )*(1.0-dU)
                         + (ptpbezTemp->GetControlPoint( i+1, j+1 )*dU))*dV );
      }
    }
}

// bilinear patch
ptLp1_0 = ptpbezTemp->GetControlPoint( 0, 0 )*(1-dU)
          + ptpbezTemp->GetControlPoint( 1, 0 )*dU;
ptLp1_1 = ptpbezTemp->GetControlPoint( 0, 1 )*(1-dU)
          + ptpbezTemp->GetControlPoint( 1, 1 )*dU;

ptL0_p2 = ptpbezTemp->GetControlPoint( 0, 0 )*(1-dV)
          + ptpbezTemp->GetControlPoint( 0, 1 )*dV;
ptL1_p2 = ptpbezTemp->GetControlPoint( 1, 0 )*(1-dV)
          + ptpbezTemp->GetControlPoint( 1, 1 )*dV;

// derivatives can just be read off
vecdFdU = (ptL1_p2 - ptL0_p2)*iDegree;
vecdFdV = (ptLp1_1 - ptLp1_0)*iDegree;

delete ptpbezTemp;

// compute the point on the surface
return( ptLp1_0*(1-dV) + ptLp1_1*dV );
```

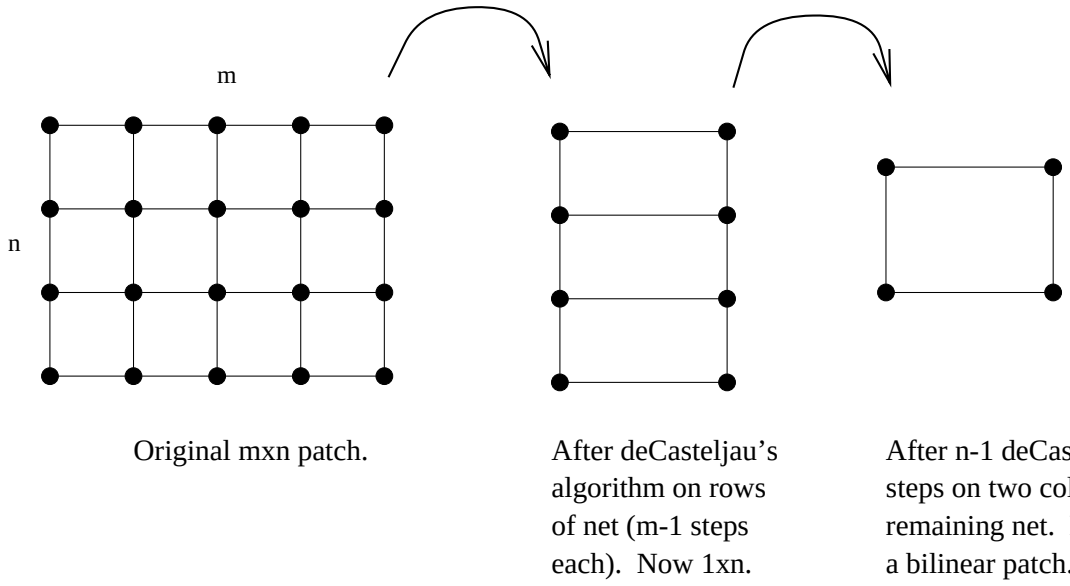

Figure 2: Modified deCasteljau evaluation.

```
}
```

We note that one improvement on this algorithm is precomputation of the control point weights. This requires the following changes to the code:

- Four new variables:
```
double uv = dU*dV;
double uV = dU*(1-dV);
double Uv = (1-dU)*dV;
double UV = (1-dU)*(1-dV);
```

- The interpolation line is replaced with
```
ptpbezNew->SetControlPoint( i, j,
                ptpbezTemp->GetControlPoint( i, j )*UV +
                ptpbezTemp->GetControlPoint( i+1, j )*uV +
                ptpbezTemp->GetControlPoint( i, j+1 )*Uv +
                ptpbezTemp->GetControlPoint( i+1, j+1 )*uv)
```

This reduces the number of vector multiplications in the inner loop from 6 to 4.

3.1.2 Limitations

The bilinear interpolation method, performance issues aside, has a limitation not addressed in the above pseudocode. If the patch is $m \times n$, rather than $m \times m$, the algorithm must perform repeated bilinear interpolation $min(m, n)$ times and then perform de Casteljau's algorithm $max(m, n) - min(m, n)$ times on the remaining curve to find the point on the surface.

3.2 Modified de Casteljau

Mann and DeRose [Mann95] presented a modified version of de Casteljau's algorithm that evaluates the surface successively in each parametric direction, each time stopping evaluation one step short of completion. See Figure 2 for a schematic diagram. The resulting bilinear patch is used to compute the derivatives and finally to compute a point on the surface.

3.2.1 Pseudocode

```
Point Compute( TPBezier& tpbezSurface, double dU, dV, Vector& vecdFdU, Vector& vecdFdV ) {

    int i, j, k;

    TPBezier* ptpbezTemp = new TPBezier( tpbezSurface );

    // control points of the bilinear patch
    Point ptLp1_0, ptLp1_1, ptL0_p2, ptL1_p2;

    // perform de Casteljau on rows of the control net
    for( i=0; i <= tpbezSurface.GetM(); i++ ) {
      for( k=1; k < tpbezSurface.GetN(); k++ ) {
        for( j=0; j <= tpbezSurface.GetN()-k; j++ ) {
          ptpbezTemp->SetControlPoint( i, j,
               ptpbezTemp->GetControlPoint( i, j )*(1-dV) +
               ptpbezTemp->GetControlPoint( i, j+1 )*dV );
        }
      }
    }

    // perform de Casteljau on the two columns of the
    // control net
    for( j=0; j <= 1; j++ ) {
      for( k=1; k < tpbezSurface.GetM(); k++ ) {
        for( i=0; i <= tpbezSurface.GetM()-k; i++ ) {
          ptpbezTemp->SetControlPoint( i, j,
               ptpbezTemp->GetControlPoint( i, j )*(1-dU) +
               ptpbezTemp->GetControlPoint( i+1, j )*dU );
        }
      }
    }

    // bilinear patch
    ptLp1_0 = ptpbezTemp->GetControlPoint( 0, 0 )*(1-dU)
      + ptpbezTemp->GetControlPoint( 1, 0 )*dU;
    ptLp1_1 = ptpbezTemp->GetControlPoint( 0, 1 )*(1-dU)
      + ptpbezTemp->GetControlPoint( 1, 1 )*dU;

    ptL0_p2 = ptpbezTemp->GetControlPoint( 0, 0 )*(1-dV)
      + ptpbezTemp->GetControlPoint( 0, 1 )*dV;
    ptL1_p2 = ptpbezTemp->GetControlPoint( 1, 0 )*(1-dV)
      + ptpbezTemp->GetControlPoint( 1, 1 )*dV;

    // derivatives can just be read off
    vecdFdU = (ptL1_p2 - ptL0_p2)*tpbezSurface.GetM();
    vecdFdV = (ptLp1_1 - ptLp1_0)*tpbezSurface.GetN();

    delete ptpbezTemp;

    // compute the point on the surface
    return( ptLp1_0*(1-dV) + ptLp1_1*dV );
}
```

3.3 Recursive Subdivision

If we represent a bicubic Bézier surface in matrix form, we have a matrix of control points:

$$P = \begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \end{bmatrix}$$

We can subdivide this surface into a "left" surface and a "right" surface (in one parametric direction, at the midpoint of the domain) by applying the following matrices, respectively:

$$L = \frac{1}{8} \begin{bmatrix} 8 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix}$$

$$R = \frac{1}{8} \begin{bmatrix} 1 & 3 & 3 & 1 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

So, LP is the left surface and RP is the right. We then subdivide in the other parametric direction (again at the midpoint) by applying L and R to $(LP)^T$ and $(RP)^T$. So, $L(LP)^T$ is the "left" half of the left surface, $R(LP)^T$ is the "right" half of the left surface, and so forth. In this way, we subdivide the 4×4 control net for a bicubic Bézier patch into four 4×4 control nets for 4 bicubic Bézier patches. Continuing in this manner, the control nets approach the surface in the limit.

We must now consider criteria for terminating the subdivision algorithm. The typical criterion is one of "flatness". When each of the points of the control net are within some fixed tolerance of planar, subdivision for the patch halts and the patch is drawn as a quadrilateral (with normals approximated using control points adjacent to the corners). Determining flatness of a bicubic patch is simply a matter of computing, for each control point, the distance from the point to the plane defined by three of the corner control points. If any of the points are further away than some fixed tolerance, the patch is not "flat" within that tolerance.

3.3.1 Pseudocode

```
ComputeSurface( TPBezier& tpbezSurface, double dTolerance ) {
    TPBezier tpbezLL( 3, 3 ); // four subpatches
    TPBezier tpbezRR( 3, 3 );
    TPBezier tpbezLR( 3, 3 );
    TPBezier tpbezRL( 3, 3 );

    if( Flat( tpbezSurface, dTolerance ) {
      tpbezSurface.DrawasQuadrilateral();
    } else {
      Subdivide( tpbezSurface, tpbezLL, tpbezLR,
                               tpbezRL, tpbezRR );
      ComputeSurface( tpbezLL, dTolerance );
      ComputeSurface( tpbezLR, dTolerance );
      ComputeSurface( tpbezRL, dTolerance );
      ComputeSurface( tpbezRR, dTolerance );
    }
}

Subdivide( TPBezier& tpbezSurface,
           TPBezier& tpbezLL, TPBezier& tpbezLR,
           TPBezier& tpbezRL, TPBezier& tpbezRR ) {

    // assume that mLeft and mRight are defined as above
```

```
        // matrices to store control point coordinates
        Matrix mAx( 4, 4 ), mAy( 4, 4 ), mAz( 4, 4 );
        Matrix mAx_l( 4, 4 ), mAy_l( 4, 4 ), mAz_l( 4, 4 );
        Matrix mAx_ll( 4, 4 ), mAy_ll( 4, 4 ), mAz_ll( 4, 4 );
        Matrix mAx_lr( 4, 4 ), mAy_lr( 4, 4 ), mAz_lr( 4, 4 );

        Matrix mAx_r( 4, 4 ), mAy_r( 4, 4 ), mAz_r( 4, 4 );
        Matrix mAx_rl( 4, 4 ), mAy_rl( 4, 4 ), mAz_rl( 4, 4 );
        Matrix mAx_rr( 4, 4 ), mAy_rr( 4, 4 ), mAz_rr( 4, 4 );

        GetControlPoints( mAx, mAy, mAz );

        // compute left and right surfaces, transposed in
        // preparation for subdivision in opposite direction
        mAx_l = (mLeft*mAx).Transpose();
        mAy_l = (mLeft*mAy).Transpose();
        mAz_l = (mLeft*mAz).Transpose();

        mAx_r = (mRight*mAx).Transpose();
        mAy_r = (mRight*mAy).Transpose();
        mAz_r = (mRight*mAz).Transpose();

        // other subdivision
        mAx_ll = mLeft*mAx_l;
        mAy_ll = mLeft*mAy_l;
        mAz_ll = mLeft*mAz_l;

        mAx_lr = mRight*mAx_l;
        mAy_lr = mRight*mAy_l;
        mAz_lr = mRight*mAz_l;

        mAx_rl = mLeft*mAx_r;
        mAy_rl = mLeft*mAy_r;
        mAz_rl = mLeft*mAz_r;

        mAx_rr = mRight*mAx_r;
        mAy_rr = mRight*mAy_r;
        mAz_rr = mRight*mAz_r;

        // construct the final surfaces
        tpbezLL.SetControlPoints( mAx_ll, mAy_ll, mAz_ll );
        tpbezLR.SetControlPoints( mAx_lr, mAy_lr, mAz_lr );
        tpbezRR.SetControlPoints( mAx_rr, mAy_rr, mAz_rr );
        tpbezRL.SetControlPoints( mAx_rl, mAy_rl, mAz_rl );
    }
```

The `DrawasQuadrilateral` function can approximate derivatives and thereby normals based on control points adjacent to the corner control points.

Also, it is worth noting that the matrix multiplications can be replaced by addition and multiplication operations on the elements of the matrices. This will provide a speed increase.

3.3.2 Limitations

This method has several limitations. First, the above implementation is specific to bicubic surfaces. However, this drawback is fairly easily overcome by more general subdivision code — using the intermediate points of de Casteljau

evaluation to form the subpatches (as for triangular patches in Section 4.2). Second, when subdivision schemes are used adaptively, as in this case, there is the opportunity for "cracks" to develop in the final surface. This is the result of adjacent subpatches being subdivided to different depths, destroying continuity along the joins [Foley90]. This can be overcome by subdividing to a fixed depth (a trivial modification to the above algorithm which allows removal of the flatness test) or by making the tolerance very small. Both of these options cause unnecessary subdivisions of some patches. More intelligent subdivision schemes connect adjoining patches, removing these undesirable effects. See [Barsky87].

3.4 Forward Differencing

The forward differencing approach presented in [Foley90] was examined, but testing did not clearly indicate that the algorithm was correct. Instead, the adaptive forward differencing algorithm from [Lien87] was modified to produce simple forward differences. This modified algorithm will be described here.

Lien et al. define a *parametric object* in a space of curves or surfaces S as a function $f : X \rightarrow S$, where X is the parameter space. If $X = \Re$, then $f = f(t)$ is a curve. If $X = \Re^2$, then $f = f(s,t)$ is a surface. They further define two operators for linear substitutions that transform f: the substitution $t/2$ is denoted by L, and the substitution $t+1$ by E.

To compute a curve using forward differencing, one begins with a curve, say C. The L transformation is then applied repeatedly to produce a curve of sufficiently fine parametrization: $D = L^n C$. Points on the curve can then be generated by computing ED, EED, ..., $E^{2^n}D$.

If the appropriate basis of representation is chosen for C then E and L become simple, easily computed transformations. Considering cubic curves, Lien et al. defines the *forward difference basis*:

$$\begin{aligned} B_3 &= \frac{1}{6}(t^3 - 3t^2 + 2t) \\ B_2 &= \frac{1}{2}(t^2 - t) \\ B_1 &= t \\ B_0 &= 1 \end{aligned}$$

Thus, the E matrix is:

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

and the L matrix is:

$$L = \begin{bmatrix} \frac{1}{8} & 0 & 0 & 0 \\ -\frac{1}{8} & \frac{1}{4} & 0 & 0 \\ \frac{1}{16} & -\frac{1}{8} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider a one-dimensional parametric cubic Bézier curve given by:

$$F(t) = aB_0^3 + bB_1^3 + cB_2^3 + dB_3^3$$

As a column vector, we have:

$$F = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

The forward difference representation is computed by multiplying by the matrix:

$$M_{B-FD} = \begin{bmatrix} -6 & 18 & -18 & 6 \\ 0 & 6 & -12 & 6 \\ -1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Now, let $F_{FD} = M_{B-FD}F$. The curve is then reparametrized: $F' = L^n F_{FD}$. Points on the curve are then given by the last entry in $F^i = E^i F'$, for $i = 0, \ldots, 2^n$. As an example, consider the curve:

$$F(t) = B_0^3 + 2B_1^3 + 3B_2^3 + 4B_3^3$$

then, for $n = 3$,

$$F = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, F_{FD} = \begin{bmatrix} 0 \\ 0 \\ 3 \\ 1 \end{bmatrix}, F' = \begin{bmatrix} 0 \\ 0 \\ \frac{3}{8} \\ 1 \end{bmatrix},$$

Now,

$$EF' = \begin{bmatrix} 0 \\ 0 \\ \frac{3}{8} \\ \frac{11}{8} \end{bmatrix}, EEF' = \begin{bmatrix} 0 \\ 0 \\ \frac{3}{8} \\ \frac{14}{8} \end{bmatrix}, E^3F' = \begin{bmatrix} 0 \\ 0 \\ \frac{3}{8} \\ \frac{17}{8} \end{bmatrix}, \ldots, E^8F' = \begin{bmatrix} 0 \\ 0 \\ \frac{3}{8} \\ 4 \end{bmatrix}$$

We can see that the last entries in these column vectors trace out the linear curve.

In order to extend this method to bicubic Bézier surfaces, one takes F (the 4×4 matrix of control points) and computes $F_{FD} = M_{B-FD}(M_{B-FD}F)^T$. F_{FD} is then reparametrized as $F' = L^n F_{FD}$. The last row of F_{FD} now describes a curve on the surface, that can be computed via the curve method above. To step to the next curve on the surface, E is applied to F'.

3.4.1 Pseudocode

```
ComputeSurface( TPBezier& tpbezSurface,
                int iNum_steps_u,
                int iNum_steps_v ) {
    Matrix mAx( 4, 4 ), mAy( 4, 4 ), mAz( 4, 4 );
    Point aaptComputed_points[iNum_steps_u+1][iNum_steps_v+1];
    int i, j;

    // assume mL, mE, and mBeztoFD are defined as described above
    tpbezSurface.GetControlPoints( mAx, mAy, mAz );

    mAx = mBeztoFd*(mBeztoFd*mAx).Transpose();
    mAy = mBeztoFd*(mBeztoFd*mAy).Transpose();
    mAz = mBeztoFd*(mBeztoFd*mAz).Transpose();

    // assume number of steps is a power of 2
    for( i=0; i < (int)(log( (double)iNum_steps_p1 ) / log( 2.0 ));
         i++ ) {
      mAx = mL*mAx;
      mAy = mL*mAy;
      mAz = mL*mAz;
    }

    // for each curve
    for( i=0; i <= iNum_steps_u; i++ ) {

      // storage for the curve
      Matrix mCurve_x( 4, 1 );
      Matrix mCurve_y( 4, 1 );
      Matrix mCurve_z( 4, 1 );

      // copy the curve coefficients out of the matrix into
      // column vectors
      for( j=0; j < 4; j++ ) {
        mCurve_x.Set( j, 0, mAx.Get( 3, j ) );
```

```
            mCurve_y.Set( j, 0, mAy.Get( 3, j ) );
            mCurve_z.Set( j, 0, mAz.Get( 3, j ) );
          }

          // scale down the curve
          for( j=0; j < (int)(log( (double)iNum_steps_p2 ) / log( 2.0 ) );
               j++ ) {
            mCurve_x = mL*mCurve_x;
            mCurve_y = mL*mCurve_y;
            mCurve_z = mL*mCurve_z;
          }

          // for each step along the curve
          for( j=0; j <= iNum_steps_v; j++ ) {

            // point on the curve in the last entry
            aaptComputed_points[i][j] = Point( mCurve_x.Get( 3, 0 ),
                                               mCurve_y.Get( 3, 0 ),
                                               mCurve_z.Get( 3, 0 ) );

            // step forward
            mCurve_x = mE*mCurve_x;
            mCurve_y = mE*mCurve_y;
            mCurve_z = mE*mCurve_z;
          }

          mAx = mE*mAx;
          mAy = mE*mAy;
          mAz = mE*mAz;
        }
    }
```

To complete the algorithm, one need only generate polygons by attaching adjacent points from `aaptComputed_points`.

As for recursive subdivision, the matrix multiplications can be replaced by addition and multiplication operations on the elements of the matrices, resulting in a speed increase.

3.4.2 Limitations

There are a few limitations to this method. First, it is most convenient if the number of steps in both parametric directions is a power of two. Secondly, the development and algorithm presented above explicitly compute cubic curves and bicubic tensor product surfaces. In order to extend this algorithm to higher degree patches one must rederive the algorithm, which involves computing, for arbitrary degree, the E and L matrices (the cost of which is discussed in Section 3.7). Lastly, in order to compute normals for patches using this method, one must estimate derivatives using adjacent points, or evaluate the two surfaces of one lower degree (in one parametric direction) representing the partial derivatives.

3.5 Adaptive Forward Differencing

Adaptive forward differencing attempts to combine the strengths of forward differencing and recursive subdivision by adjusting the parametrization of the curve at each forward difference step so that the step moves a fixed "distance" along the curve. The algorithm presented here is an extension of the fixed forward differencing algorithm above, as described in [Lien87].

Where the above algorithm repeatedly applies the L transformation before beginning to take forward steps, the adaptive algorithm attempts to take a forward step, examines the distance between the current point and the new point on the curve, and adjusts the parametrization if the next point is too near or too distant. The process is then repeated.

In order to determine when to adjust the parametrization of the curve, the distance between the current point and the next point on the curve (from EF) is compared against some threshold. If the distance is greater than the threshold, the L matrix is applied to F. If the distance is less than half of the threshold, L^{-1} is applied [Lien87]. This process continues until the distance to the next point falls within the desired range. The forward step is then taken and the process is repeated. However, one must be careful to avoid infinite loops when the distance to the next point cannot fall between half of the threshold and the threshold, no matter what the parametrization.

It is possible to perform adaptive forward differencing for surfaces by computing test curves in the parameter orthogonal (say s) to the one in which curves are being computed (say t). To compute the step size, δs, between one curve $f(s_i, t)$ and the next curve $f(s_i + \delta s, t)$, the minimum step size used by the test curves at $s = s_i$ is used. The implementation below does not include this functionality; instead, it takes fixed forward difference steps between successive curves.

3.5.1 Pseudocode

```
ComputeSurface( TPBezier& tpbezSurface,
                int iNum_steps_u,
                double dTolerance ) {
    Matrix mAx( 4, 4 ), mAy( 4, 4 ), mAz( 4, 4 );

    // need a dynamic structure to record points along
    // individual curves, since there'll be a variable
    // number of them
    List<Point> alptComputed_points[iNum_steps_u+1];
    int i, j;

    // assume mL, mLinv, mE, and mBeztoFD are defined as
    // described above
    tpbezSurface.GetControlPoints( mAx, mAy, mAz );

    mAx = mBeztoFd*(mBeztoFd*mAx).Transpose();
    mAy = mBeztoFd*(mBeztoFd*mAy).Transpose();
    mAz = mBeztoFd*(mBeztoFd*mAz).Transpose();

    // assume number of steps is a power of 2
    for( i=0; i < (int)(log( (double)iNum_steps_p1 ) / log( 2.0 ));
         i++ ) {
      mAx = mL*mAx;
      mAy = mL*mAy;
      mAz = mL*mAz;
    }

    // for each curve (fixed step size)
    for( i=0; i <= iNum_steps_u; i++ ) {

      // storage for the curve
      Matrix mCurve_x( 4, 1 );
      Matrix mCurve_y( 4, 1 );
      Matrix mCurve_z( 4, 1 );

      // copy the curve coefficients out of the matrix into
      // column vectors
      for( j=0; j < 4; j++ ) {
        mCurve_x.Set( j, 0, mAx.Get( 3, j ) );
        mCurve_y.Set( j, 0, mAy.Get( 3, j ) );
        mCurve_z.Set( j, 0, mAz.Get( 3, j ) );
      }
```

```
// record the first point, it's on the curve
alptComputed_points[i].attachr( Point( mCurve_x.Get( 3, 0 ),
                                mCurve_y.Get( 3, 0 ),
                                mCurve_z.Get( 3, 0 ) ) );

// how many steps left?
int iNum_steps = 1;

for( ; iNum_steps > 0; ) {

// count ups and downs
int iNum_ups;
int iNum_downs;

// need to record the old curve
Matrix mCurve_old_x = mCurve_x;
Matrix mCurve_old_y = mCurve_y;
Matrix mCurve_old_z = mCurve_z;

// adjust up or down as necessary
iNum_ups = iNum_downs = 0;

// loop until we find a good step size
for( ;; ) {

  // record the curve before the forward step
  Matrix mCurve_x_prev = mCurve_x;
  Matrix mCurve_y_prev = mCurve_y;
  Matrix mCurve_z_prev = mCurve_z;

  // attempt to take a forward step
  mCurve_x = mE*mCurve_x;
  mCurve_y = mE*mCurve_y;
  mCurve_z = mE*mCurve_z;

  // get the previous and next points on the curve
  Point pOld( mCurve_old_x.Get( 3, 0 ),
              mCurve_old_y.Get( 3, 0 ),
              mCurve_old_z.Get( 3, 0 ) );

  Point pNew( mCurve_x.Get( 3, 0 ),
              mCurve_y.Get( 3, 0 ),
              mCurve_z.Get( 3, 0 ) );

  // compute the distance between points and compare
  // to the tolerance

  // only step up if no downstep has been taken, and
  // vice versa
  // also can't step up unless the number of steps
  // remaining is even
  if( ((pOld-pNew).Magnitude() > dTolerance)
      && (iNum_ups == 0) ) {

    // step is too big, adjust down
    // double the number of steps left
    iNum_steps *= 2;
```

```
            mCurve_x = mL*mCurve_x_prev;
            mCurve_y = mL*mCurve_y_prev;
            mCurve_z = mL*mCurve_z_prev;
            iNum_downs++;
          } else if( ((pOld-pNew).Magnitude() < dTolerance/2.0)
                  && (iNum_downs == 0)
                  && ((iNum_steps % 2) == 0) ) {

            // step is too small, adjust up
            iNum_steps /= 2;

            mCurve_x = mLinv*mCurve_x_prev;
            mCurve_y = mLinv*mCurve_y_prev;
            mCurve_z = mLinv*mCurve_z_prev;
            iNum_ups++;
          } else {
            // good

            iNum_steps--;

            alptComputed_points[i].attachr( pNew );

            // done this step
            break;
          }
        }
      }

      // next curve

      mAx = mE*mAx;
      mAy = mE*mAy;
      mAz = mE*mAz;
    }
```

Because there are potentially a different number of points on adjacent curves, it is important to consider how these points can be joined to form polygons to represent the surface. The approach used here is simple, but produces fairly good polygonalizations in most cases.

Assume we have two adjacent curves, C_1 with r samples, and C_2 with s samples. Further assume, without loss of generality, that $r > s$. Let $k = \frac{s}{r}$. Then, for $i = 0, \ldots, r$, polygons can be constructed by joining points $\{C_1[i], C_1[i+1], C_2[\lfloor (i+1)k \rfloor], C_2[\lfloor ik \rfloor]\}$ if $\lfloor (i+1)k \rfloor > \lfloor ik \rfloor$ and $\lfloor (i+1)k \rfloor < s$, and points $\{C_1[i], C_1[i+1], C_2[\lfloor ik \rfloor]\}$ otherwise.

Figures 3 (a) and (b) show examples of this approach. From Figure 3 (b) we can see that there will be occasions where non-planar quadrilaterals and long, thin triangles will be produced. These cases will occur when two adjacent curves have very different curvatures, which indicates that the curve-to-curve spacing is too great. Also, it is important to note that one must be careful to maintain consistent polygon orientations when using this scheme.

3.5.2 Limitations

As mentioned above, performing adaptive differencing across the surface is difficult from an implementation perspective. Also, as for forward differencing, normals must be computed by estimating derivatives using adjacent points on the surface or using the partial derivative surfaces.

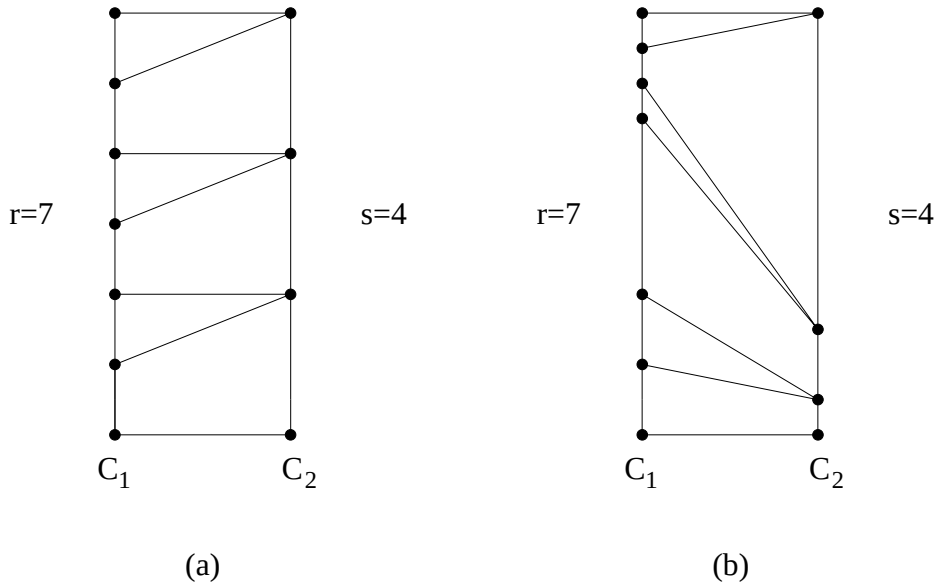

Figure 3: An approach to joining adjacent curves.

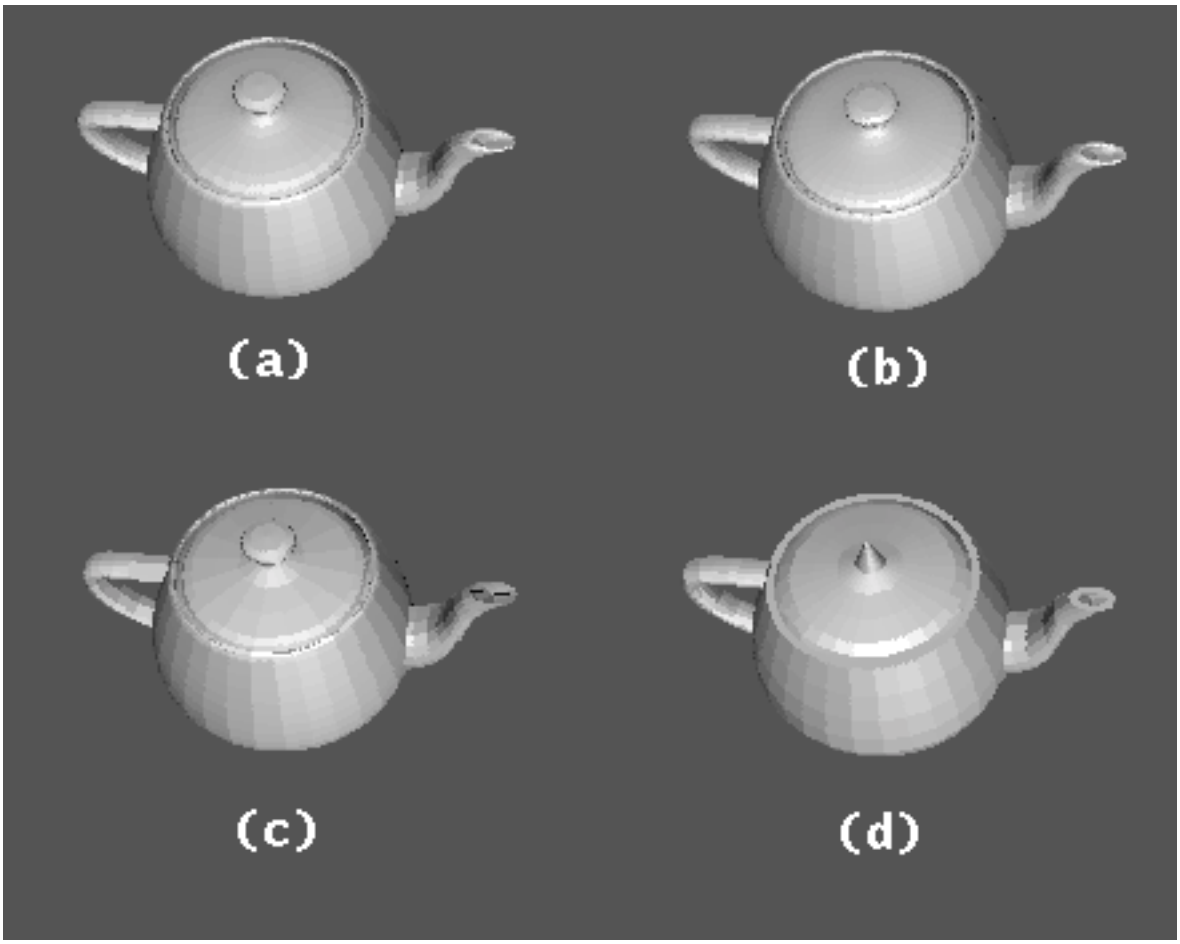

Figure 4: Various evaluations of the Utah teapot.

Evaluation Technique	Parameters	Tessellation Required?	Image	Time (s)
modified de Casteljau	iNum_steps_u = 10 iNum_steps_v = 10	yes	(a)	2.1
forward differencing†	iNum_steps_u = 8 iNum_steps_v = 8	no	(b)	1.4
forward differencing‡	iNum_steps_u = 8 iNum_steps_v = 8	no	-	4.0
recursive subdivision	dTolerance = .001	no	(c)	2.8
adaptive forward differencing††	iNum_steps_u = 8 dTolerance = .02	no	(d)	1.5
adaptive forward differencing‡	iNum_steps_u = 8 dTolerance = .02	no	-	3.4
bilinear interpolation	iNum_steps_u = 10 iNum_steps_v = 10	yes	(a)	2.3
bilinear interpolation II*	iNum_steps_u = 10 iNum_steps_v = 10	yes	(a)	2.0

Table 1: Comparison of evaluation techniques for tensor product surfaces. (†estimated normals, ††no normals, ‡exact normals, *precomputed weights)

3.6 Comparison

In order to compare the techniques described above, each was used to evaluate the Utah teapot, which is made up of bicubic patches. Figure 4 shows the flat shaded results and Table 1 describes the parameters used for evaluation along with the time (to the nearest tenth of a second) taken by each technique.

The times shown in the above table include only tessellation (if required) and evaluation. They do not include time for computing normals from derivative vectors. The recursive subdivision time measures only the cumulative time of the subdivision operations themselves. The modified de Casteljau algorithm and the two versions of bilinear interpolation compute the same values and construct identical tessellations.

Comparing times for the various evaluation techniques is difficult, as an accurate comparison would require determining a metric by which the images produced could be compared.

The teapot evaluated using adaptive forward differencing exhibits a few flaws, such as a misshapen lid. Flaws such as this occur due to the use of Euclidean distance between successive curve points as the metric for determining whether the curve should be reparametrized. The original adaptive forward difference algorithm, presented by Lien et al., used pixel distance as the metric, with a threshold of approximately one pixel. In this case, the flaws would not appear.

Examining the relative evaluation times, we see that forward differencing with estimated normals is the fastest. However, if we compute exact normals with this method, using partial derivative "surfaces", we find that the time increases by a factor of approximately 2.85 to 4.0 seconds. Similarly, adaptive forward differencing with no normals is second fastest at 1.5 seconds, but computing normals increases the time by a factor of approximately 2.3 to 3.4 seconds. We see different time factors for computing exact normals with these two methods since the partial derivative surfaces can be computed at lower cost than the actual surface with adaptive forward differencing. This is the case as the forward steps attempted with intermediate parametrizations need not be applied to the derivative surfaces. With normal forward differencing, we see an approximate tripling in execution time, as all computation performed for the actual surface must be performed for both derivative surfaces.

Therefore, forward differencing is the fastest evaluation technique if estimated derivatives are satisfactory and the surface being evaluated is a bicubic patch. In order to evaluate surfaces of arbitrary degree, time for computation of the transformation matrices must be included. As discussed in the next section, if exact normals are required and fixed sampling is acceptable, de Casteljau's algorithm is probably best for arbitrary degree patches. Bilinear interpolation with precomputed weights is slightly faster for bicubic patches. If one desires an adaptive algorithm, recursive subdivision is the best choice if normals are required (assuming cracks are either acceptable or repaired) and adaptive forward differencing is best otherwise.

3.7 Higher Degree

Even though low degree patches are most commonly used in practice, it is interesting to consider the asymptotic behaviour of the algorithms presented above. The number of multiplications required by an algorithm will be used as a metric for comparison. While additions, function call overhead and loop control may contribute significantly to the actual time required by an algorithm, the number of multiplications will provide a rough estimate of performance, as it has the largest contribution to execution time. It is important to note that if the analysis below shows two algorithms performing similarly, it may not be clear which is in fact the more efficient. Also, while the analysis correctly predicts the ordering of the empirical results in Table 1 from most efficient to least efficient for bicubics, it does not correctly predict the relative times. This is likely due to overhead that plays a continually smaller role as degree increases.

Bilinear interpolation without precomputed weights requires $n(n+1)(2n+1)$ multiplications for each evaluation of an $n \times n$ tensor product patch; precomputation of weights reduces the number of multiplications to $\frac{2}{3}n(n+1)(2n+1)$. The modified de Casteljau technique requires $(n+3)((n+1)n-2)+6$ multiplications. While each of this algorithms is $O(n^3)$, modified de Casteljau evaluation is asymptotically twice as fast as bilinear interpolation and 1.5 times faster than bilinear interpolation with precomputed weights [Mann95]. In order to compare these techniques to the forward differencing and recursive subdivision algorithms, the costs must be scaled by the product of the number of tessellation steps in the u and v parameter directions.

Recursive subdivision of an $n \times n$ tensor product patch requires 6 matrix multiplications per subdivision operation, each of which requires $(n+1)^3$ multiplications. Subdividing a surface to a fixed depth d thus uses:

$$4^{d-1}6(n+1)^3$$

multiplications. Evaluating recursive subdivision with an adaptive termination condition (ie. flatness) is dependent on the surface being considered.

The non-adaptive forward differencing algorithm presented above requires

$$11n^3 + 18log_2(S_u)n^3 + S_u\left[3n^3 + log_2(S_v)n^2 + S_v n^2\right]$$

multiplications to evaluate a complete $n \times n$ patch where S_u and S_v are the number of steps performed in the u and v parameter directions, respectively. This cost includes the expense of:

1. computing a forward difference basis matrix of the appropriate degree;
2. computing the Bézier to forward difference conversion matrix (M_{B-FD});
3. computing the step down matrix (L);
4. converting the surface to the forward difference basis;
5. reparametrizing the surface so that S_u curve to curve steps may be taken;
6. reparametrizing each individual curve so that S_v curve steps may be taken;
7. taking S_v forward steps along each of S_u curves; and
8. taking S_u curve to curve forward steps.

Again, consideration of the adaptive version of forward differencing is difficult, as it depends on the particular patch being evaluated. A presentation of the computation of the various matrices for arbitrary degree as used in this technique (and its adaptive counterpart) can be found in [Shantz88].

3.7.1 de Casteljau Methods *vs* Recursive Subdivision

To compare de Casteljau methods (bilinear interpolation and modified de Casteljau) with recursive subdivision, we choose a particular sampling rate, say $S_u \times S_v$, for the former. Then we have:

$$T_{bilinear} = S_u S_v n(n+1)(2n+1)$$

and

$$T_{subdivision} = 4^{d-1}6(n+1)^3$$

Equating these two quantities, we can solve for d, determining the subdivision depth at which the bilinear interpolation and subdivision techniques perform similarly. We have:

$$\begin{aligned} 4^{d-1}6(n+1)^3 &= S_u S_v n(n+1)(2n+1) \\ 4^{d-1} &= S_u S_v \frac{n(n+1)(2n+1)}{6(n+1)^3} \\ 4^{d-1} &= S_u S_v \frac{n(2n+1)}{6(n+1)^2} \end{aligned}$$

so

$$d = \lfloor \log_4 \left(S_u S_v \frac{n(2n+1)}{6(n+1)^2} \right) + 1 \rfloor$$

For $S_u = S_v = 10$ and $n = 3$, $d = \lfloor 3.22 \rfloor = 3$. Now, to consider asymptotic behaviour, we find:

$$\lim_{n \to \infty} S_u S_v \frac{n(2n+1)}{6(n+1)^2} = \frac{1}{3} S_u S_v$$

Therefore, bilinear interpolation at a sampling rate of $S_u \times S_v$ is of similar cost to recursive subdivision with $\frac{1}{3} S_u S_v$ subdivisions, or to a fixed depth of $\lfloor log_4(\frac{1}{3} S_u S_v) + 1 \rfloor$. We find the corresponding value to be $\frac{2}{9} S_u S_v$ when compared to bilinear interpolation with precomputed weights and $\frac{1}{6} S_u S_v$ when compared to modified de Casteljau evaluation.

Alternately, we can compute the number of polygons produced by recursive subdivision for the same time cost as another method. The number of polygons produced by recursive subdivision is $3n + 1$, where n is the total number of subdivisions as computed above. Therefore, for the same time cost as bilinear interpolation at a sampling rate of $S_u \times S_v$, recursive subdivision can produce $3 \cdot \frac{1}{3} S_u S_v + 1 = S_u S_v + 1$ polygons. In the time that bilinear interpolation with precomputed weights takes for a sampling rate of $S_u \times S_v$, recursive subdivision can construct $\frac{2}{3} S_u S_v + 1$ polygons. And in the time required for modified de Casteljau evaluation to compute a sampling of $S_u \times S_v$, recursive subdivision only constructs $\frac{1}{2} S_u S_v + 1$ polygons.

Note that while recursive subdivision computes fewer polygons than the de Castlejau methods for a fixed amount of time, the quality of the recursive subdivision approximation may be better since it chooses its polygons more wisely.

3.7.2 de Casteljau Methods *vs* Forward Differencing

If we choose a particular sampling rate amenable to the forward differencing algorithm, say $S_u = S_v = 8$, we find its cost of evaluation becomes $90n^3 + 88n^2$. At the same sampling rate, bilinear interpolation costs $128n^3 + 192n^2 + 64n$ and modified de Casteljau evaluation $64n^3 + 256n^2 + 64n$. Cost for these techniques for $n = 1, \ldots, 9$ is shown in Table 3.7.2. We see that for patches of bi-degree higher than 1, modified de Casteljau is cheaper than bilinear interpolation, and for bi-degree greater than 6, it is cheaper than forward differencing. However, bilinear interpolation with precomputed weights is cheaper than modified de Casteljau for bi-degree up to 6.

3.8 Other Techniques

Integer adaptive forward differencing [Chang89] is an integer implementation of the adaptive forward differencing algorithm. While it promises speed improvements over the floating point algorithm, the implementation is more complicated, and error analysis is difficult. Therefore, this technique was not implemented.

4 M-dimensional Simplex Surfaces

A surface over an m-dimensional simplex is given by:

$$B(u) = \sum_{\vec{i}, |\vec{i}| = n} P_{\vec{i}} B^n_{\vec{i}}(u)$$

where

$$\vec{i} = (i_0, \ldots, i_d), \quad B^n_{\vec{i}}(u) = \left(\begin{array}{c} n \\ \vec{i} \end{array} \right) u_0^{i_0} \cdot \ldots \cdot u_d^{i_d}, \quad |\vec{i}| = \sum_{j=0}^{d} i_j, \quad \left(\begin{array}{c} n \\ \vec{i} \end{array} \right) = \frac{n!}{(i_0! \cdots i_d!)},$$

Bi-Degree	Forward differencing	modified de Casteljau	Bilinear interpolation	Bilin. interp. w/precomputed weights
1	178	384	384	256
2	1072	1664	1920	1280
3	3222	4224	5376	3584
4	7168	8448	11520	7680
5	13450	14720	21120	14080
6	22608	23424	34944	23296
7	35182	34944	53760	35840
8	51712	49644	78336	52224
9	72738	67968	109440	72960

Table 2: Cost (in number of multiplications) of techniques for $n = 1, \ldots, 9$.

and where $u = (u_0, u_1, \ldots, u_d)$ are the barycentric coordinates of a point in our domain with respect to a particular simplex. This generalization of the Bernstein polynomials to a d dimensional space gives us a mapping from a d simplex onto a piece of a d dimensional manifold.

Here, we will concentrate on triangular patches ($d = 2$), where the domain is a triangle, and the manifold is a three dimensional surface patch. The algorithms presented here assume a canonical domain triangle: $\Delta(0,0)(0,1)(1,0)$. Farin's book provides a more detailed introduction to triangular Bézier patches [Farin93].

4.1 de Casteljau Evaluation

As with tensor product surfaces, the straightforward approach to evaluation of a triangular patch of this type is via the de Casteljau algorithm. The algorithm for higher dimensional functions is a direct generalization of the algorithm for curves, based on repeated linear interpolation. Beginning with a triangular control net $P_{\vec{i}}^0$, $|\vec{i}| = n$ for a degree n patch, we evaluate the surface at a point $u = (u_0, u_1, u_2)$ in the domain by computing:

$$P_{\vec{i}}^r(u) = u_0 P_{\vec{i}+(1,0,0)}^{r-1}(u) + u_1 P_{\vec{i}+(0,1,0)}^{r-1}(u) + u_2 P_{\vec{i}+(0,0,1)}^{r-1}(u)$$

for $r = 1, \ldots, n$ and $|\vec{i}| = n - r$. [Farin93]

A schematic of the evaluation for a quadratic patch is shown in Figure 5. Note that the triangle formed in the last step of the algorithm spans the tangent plane at the evaluation point on the surface.

4.1.1 Pseudocode

```
Point Compute( MSimpBezier& msbezSurface,
               double dP1, double dP2,
               Vector vecV1, vecV2 ) {

    // pointers for intermediate surfaces
    MSimpBezier* pmsbSurface = &msbezSurface;
    MSimpBezier* pmsbOld;
    MSimpBezier* pmsbPoint1;
    MSimpBezier* pmsbPoint2;
    MSimpBezier* pmsbOrig;

    int i;

    // stop one from the end
    for( i=1; i < msbezSurface.GetDegree(); i++ ) {
      pmsbOld = pmsbSurface;
      pmsbSurface = deCasteljau( pmsbOld, dP1, dP2 );
```

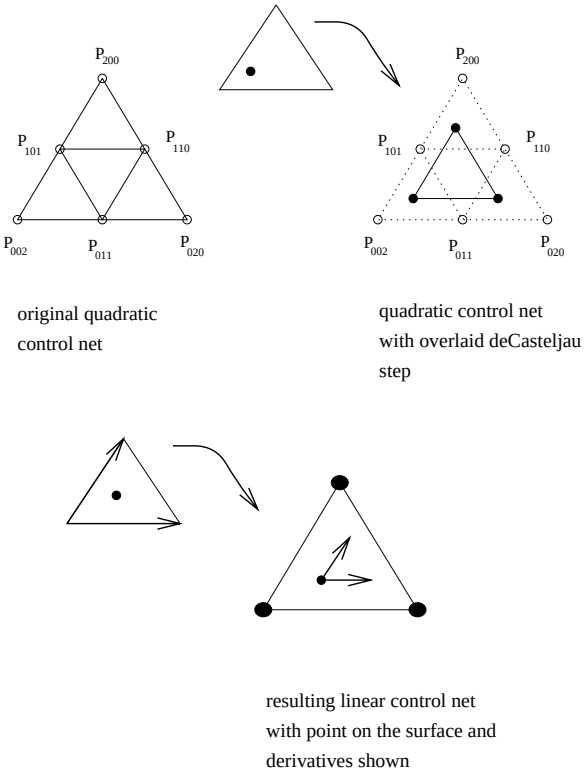

Figure 5: Schematic of a control net for a quadratic triangular patch.

```
  if( &msbezSurface != pmsbOld ) {
    delete pmsbOld;
  }
}

// we now have a linear patch - evaluate it at the corners
// of the domain and subtract the resulting surface points
// to get directional derivatives suitable for normal
// computation
pmsbPoint1 = deCasteljau( pmsbSurface, 0.0, 1.0 );
pmsbPoint2 = deCasteljau( pmsbSurface, 0.0, 0.0 );
pmsbOrig = deCasteljau( pmsbSurface, 1.0, 0.0 );

vecV1 = pmsbPoint1->GetControlPoint( 0, 0 ) -
  pmsbOrig->GetControlPoint( 0, 0 );
vecV2 = pmsbPoint2->GetControlPoint( 0, 0 ) -
  pmsbOrig->GetControlPoint( 0, 0 );

delete pmsbPoint1;
delete pmsbPoint2;
delete pmsbOrig;

// complete the evaluation
pmsbOld = pmsbSurface;
pmsbSurface = deCasteljau( pmsbOld, dP1, dP2 );
delete pmsbOld;
```

```
    return( pmsbSurface->GetControlPoint( 0, 0 ) );
}

MSimpBezier* deCasteljau( MSimpBezier* pmsbezSurface,
                          double dP1, double dP2 ) {
    int i, j, k;
    int iDeg = pmsbezSurface->GetDegree();
    MSimpBezier* pmsbReturn = new MSimpBezier( pmsbezSurface->GetDimension(),
                                               iDeg-1 );

    // for each multiindex I=(i, j, k) with |I| = iDeg
    // thus, can only really evaluate 2-simplices
    for( i=0; i <= iDeg-1; i++ ) {
      for( j=0; j <= iDeg-1; j++ ) {
        for( k=0; k <= iDeg-1; k++ ) {
          if( i+j+k != iDeg-1 ) {
            continue;
          }

          Point ptPoint( 0.0, 0.0, 0.0 );

          ptPoint = pmsbezSurface->GetControlPoint( i+1, j )*dP1
                    + pmsbezSurface->GetControlPoint( i, j+1 )*dP2
                    + pmsbezSurface->GetControlPoint( i, j )*(1.0-dP1-dP2);

          pmsbReturn->SetControlPoint( i, j, ptPoint );
        }
      }
    }

    return( pmsbReturn );
}
```

Here, the `MSimpBezier` class hides the details of the technique used for storage of the control net for triangular patches. Naïvely, one could store the control points in a three-dimensional array, indexed by $\vec{i}$. However, we know that i_0, i_1 and i_2 are not independent, so we can store the net in a two-dimensional array, indexed by i_0 and i_1 (for example). This approach is fastest, but is not most space-efficient. The most space-efficient technique is to store the control net in a linear array of size:

$$Dim(n, d) = Dim(n-1, d) + Dim(n, d-1)$$

where $Dim(n, 0) = Dim(0, d) = 1$ and $Dim(n, d) = 0$ for n or d negative. This approach derives from the observation that one can represent a degree n, dimension d control net as two subnets, one for a degree $n-1$, dimension d surface, and one for a degree n, dimension $d-1$ surface. Using this approach, one must convert the multiindex $\vec{i}$ into a offset into the "simplicial" array:

```
// accept multiindex of arbitrary size

int ConvertMI( int ai[], int iDimension, int iDegree ) {
  int iSum = iDegree;
  int iOffset = 0;
  int j;

  for( j=0; j < iDimension; j++ ) {
    iSum -= aiI[j];
    iOffset += Dim( iSum-1, iDimension-j );
  }
```

```
  return( iOffset );
}
```

Using this storage technique, de Casteljau evaluation can actually be performed without explicitly converting each multiindex into an offset, effectively performing the conversions incrementally as the evaluation progresses.

```
MSimpBezier* deCasteljau( MSimpBezier* pmsbezSurface,
                          double dP1, double dP2 ) {
    int i, j, k;
    int iDeg = pmsbezSurface->GetDegree();
    MSimpBezier* pmsbReturn = new MSimpBezier( pmsbezSurface->GetDimension(),
                                               iDeg-1 );
    i = 0;
    j = 1;
    int iRow, iCol;

    for( iRow = 0; iRow <= iDeg-1; iRow++ ) {
      for( iCol = 0; iCol <= iRow; iCol++ ) {
        pmsbReturn->SetControlPoint( i,
                                     pmsbezSurface->GetControlPoint( i )*dP1
                                     + pmsbezSurface->GetControlPoint( j )*dP2
                                     + pmsbezSurface->GetControlPoint( j+1 )*(1.0-dP1-dP2 ) );
        i++;
        j++;
      }
      j++;
    }

    return( pmsbReturn );
}
```

Here, the `GetControlPoint` and `SetControlPoint` calls reference linearly into the simplicial array.

4.2 Recursive Subdivision

The motivation for recursively subdividing triangular Bézier patches is identical to that for tensor product surfaces: we wish to subdivide the control net into some number of subnets, each of which is a better approximation to a piece of the surface. Ideally, one would perform this process recursively and halt when each subnet is sufficiently flat. This would produce an approximation to the complete surface, to within a specified tolerance. Problems concerning using flatness as a termination condition are discussed below.

When subdividing tensor product surfaces, above, we subdivided in each parametric direction at the midpoint of the domain ($u = \frac{1}{2}$, $v = \frac{1}{2}$). When subdividing triangular patches, we must choose a point in the domain which will define the subpatches. The obvious choice is the centroid of the domain triangle. Figure 6 shows the subdivision of a quadratic patch at at arbitrary point in the domain. Figure 6 (a) shows the de Casteljau evaluation of a point on the surface. Figure 6 (b) shows the intermediate de Casteljau evaluation points and Figure 6 (c) shows how the intermediate points form the control nets of the three subpatches. In this way, a single subdivision operation is in some sense equivalent to a de Casteljau evaluation of a point on the surface.

In the algorithm below, the three subpatches are constructed from the intermediate de Casteljau points (computed using the de Casteljau algorithm above) by manipulation of the multiindices. Each subpatch is constructed row by row, one row for each de Casteljau step. Construction starts at the bottom row of the new patch (eg. for a quadratic, the control points with multiindices (0,0,2), (0,1,1) and (0,2,0)) which is constructed from the outermost control points of the original patch, and finishes with the single point comprising the degree 0 patch. The algorithm assumes that the control nets are stored in the simplicial array form discussed above.

4.2.1 Pseudocode

```
ComputeSurface( MSimpBezier& msbezSurface,
```

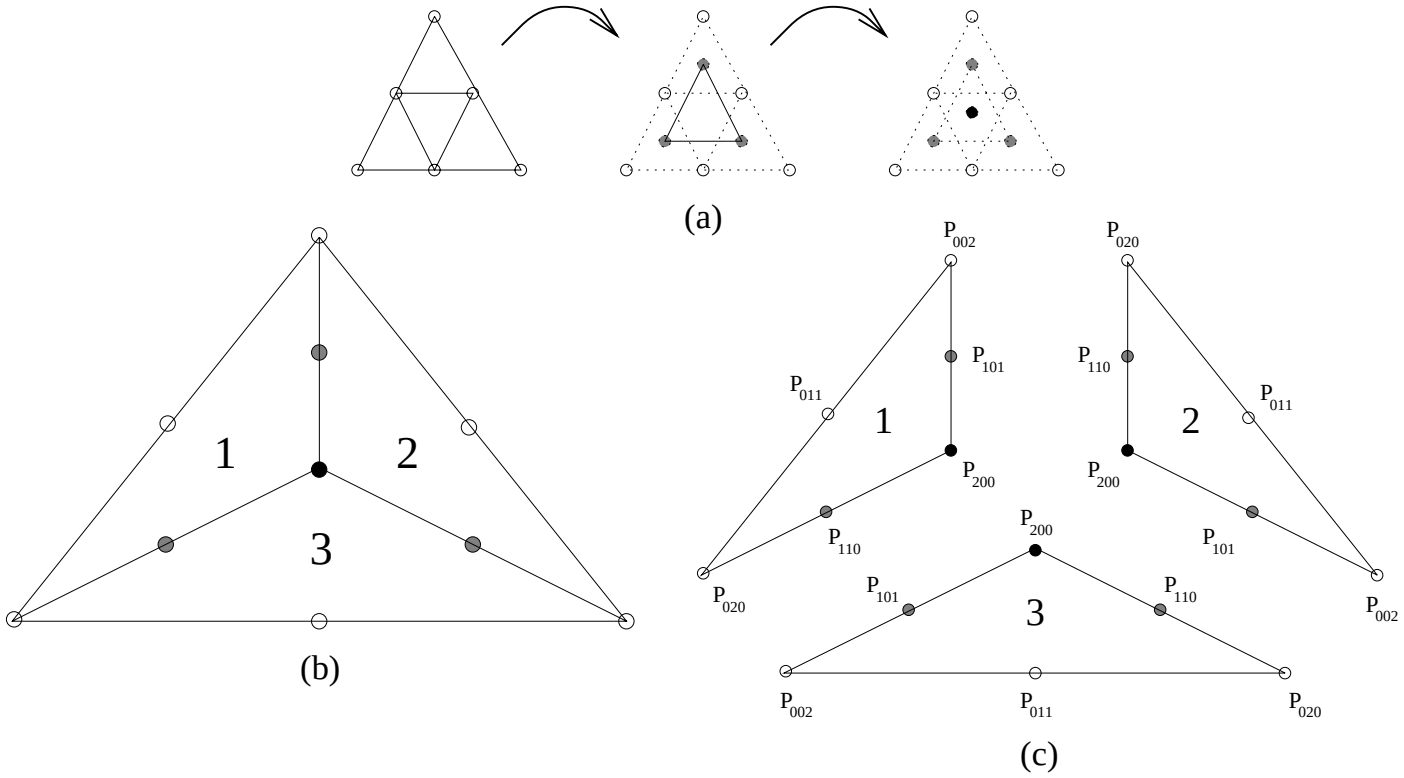

Figure 6: Subdivision of a quadratic triangular Bézier patch.

```
                double dTolerance ) {

    // barycentric coordinates of domain centroid
    double dP1 = 1.0/3.0;
    double dP2 = 1.0/3.0;

    // three subpatches of same dimension and degree
    MSimpBezier msbP( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbQ( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbR( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );

    if( Flat( msbezSurface, dTolerance ) ) ) {
      msbezSurface.DrawasTriangle();
    } else {
      Subdivide( msbezSurface, dP1, dP2, msbP, msbQ, msbR );

      ComputeSurface( msbP, dTolerance );
      ComputeSurface( msbQ, dTolerance );
      ComputeSurface( msbR, dTolerance );
    }
}

Subdivide( MSimpBezier& msbezSurface,
           double dP1, double dP2,
```

```
        MSimpBezier& msbP,
        MSimpBezier& msbQ,
        MSimpBezier& msbR ) {

int i, iRight, iLeft, j;

// pointer to the current surface
MSimpBezier* pmsbSurface = this;
MSimpBezier* pmsbOld;

// each time through the loop, points from the current
// surface (initially the original surface) are placed
// into appropriate places in the sub-control nets, then
// a single de Casteljau step is performed on the surface
// to reduce it's degree
for( i=iDegree; i >= 0; i-- )  {
  int iMI[3]={0,0,0}; // multiindex

  iRight = pmsbSurface->GetDimension();
  iLeft = iRight-1;
  iMI[iLeft] = pmsbSurface->GetDegree();

  // first, place i points into the control net
  // of the first subsurface
  // eg. for a quadratic patch, first pass:
  // i=2, iRight=2, iLeft=1, so MI=(0, 2, 0),
  // (0, 1, 1), (0, 0, 2) and points are set
  // in the simplicial array at offsets 3, 4, 5
  // second pass:
  // i=1, iRight=2, iLeft=1, so MI=(0, 1, 0),
  // (0, 0, 1)  and points are set in the
  // simplicial array at offsets 1, 2
  // last pass: MI=(0, 0, 0), offset 0 is set

  for( j=0; j < i+1; j++ ) {
    // place points directly into simplicial array
    msbP.SetControlPoint( i*(i-1)/2+i+j,
                          pmsbSurface->GetControlPoint( iMI[0], iMI[1] ) );

    if( iMI[iLeft] == 0 ) {
      iLeft = (iLeft + 1) % 3;
      iRight = (iRight + 1) % 3;
      break;
    }

    iMI[iLeft]--;
    iMI[iRight]++;
  }

  // now, place i points into the control net
  // of the second subsurface, as above
  for( j=0; j < i+1; j++ ) {
    msbQ.SetControlPoint( i*(i-1)/2+i+j,
                          pmsbSurface->GetControlPoint( iMI[0], iMI[1] ) );
    if( iMI[iLeft] == 0 ) {
      iLeft = (iLeft + 1) % 3;
      iRight = (iRight + 1) % 3;
```

```
            break;
          }

          iMI[iLeft]--;
          iMI[iRight]++;
        }

        // now, place i points into the control net
        // of the third subsurface, as above
        for( j=0; j < i+1; j++ ) {
          msbR.SetControlPoint( i*(i-1)/2+i+j,
                                pmsbSurface->GetControlPoint( iMI[0], iMI[1] ) );
          if( iMI[iLeft] == 0 ) {
            iLeft = (iLeft + 1) % 3;
            iRight = (iRight + 1) % 3;
            break;
          }

          iMI[iLeft]--;
          iMI[iRight]++;
        }

        // reduce degree and continue
        pmsbOld = pmsbSurface;
        pmsbSurface = deCasteljau( pmsbSurface, dP1, dP2 );

        if( pmsbOld != this ) {
          delete pmsbOld;
        }
      }
  }
```

The flatness test for triangular Bézier patches is analogous to that for tensor product patches. The `DrawasTriangle` function can approximate derivatives and thereby normals based on control points adjacent to the corner control points.

4.2.2 Limitations

Examining Figure 7, which shows a subdivision to depth 3 of a domain triangle, we see that the resulting triangles have poor aspect ratios. This problem will transfer to the range, and results in many long, thin triangles. This problem appears because the algorithm never divides a domain triangle anywhere but in the interior — an edge is never divided. Also, for the same reason, the planarity condition for termination of the subdivision can be problematic: there is no guarantee that the subpatches will ever become planar to within the given tolerance. The problem is amplified if one or more boundary curves of the original patch are themselves non-planar. However, one benefit in never subdividing an edge is cracks will not develop in the final surface.

4.3 Binary Subdivision

Peters presents an algorithm for binary subdivision of triangular Bézier patches, using a technique based exclusively on curve subdivision. The algorithm discussed here performs the same subdivisions, but uses the general patch subdivision function from above. The algorithm simply recursively subdivides the patch at the midpoint of one of the domain edges.

4.3.1 Pseudocode

```
ComputeSurface( MSimpBezier& msbezSurface,
```

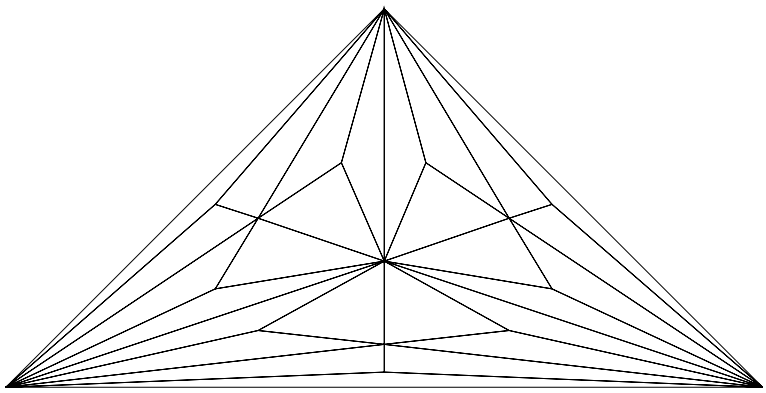

Figure 7: Centroid subdivision of a domain triangle.

```
                double dTolerance ) {

        // storage for three new patches of same dimension
        // and degree
        MSimpBezier msbP( msbezSurface.GetDimension(),
                          msbezSurface.GetDegree() );
        MSimpBezier msbQ( msbezSurface.GetDimension(),
                          msbezSurface.GetDegree() );
        MSimpBezier msbR( msbezSurface.GetDimension(),
                          msbezSurface.GetDegree() );

        // domain coordinates for subdivision
        double adP[]={0.0, 0.5, 0.5};

        if( Flat( msbezSurface, dTolerance ) ) {
          msbezSurface.DrawasTriangle();
        } else {
          Subdivide( msbezSurface, adP, msbP, msbQ, msbR );

          // ignore one of three (P) - it's degenerate
          ComputeSurface( msbQ, dTolerance );
          ComputeSurface( msbR, dTolerance );
        }
}
```

4.3.2 Limitations

Binary subdivision, as Figure 8 shows, can improve dramatically the poor aspect ratio problem of centroid subdivision. Further improvements to the algorithm, such as choosing the edge to subdivide based on criteria such as longest surface edge or greatest curvature are also possible.

Because, unlike the centroid subdivision algorithm, this algorithm does divide domain edges, the "crack" problem appears. Altering the algorithm to perform subdivision of edges globally would remove this problem, but a global solution is difficult to implement.

4.4 4:1 Subdivision

4:1 subdivision is an alternate subdivision algorithm for triangular patches that divides the surface, at each step, into four subpatches rather three. The approach presented here requires five subdivision operations (de Casteljau evaluations). A similar approach is discussed by Böhm [Böhm83], which requires only four subdivision operations.

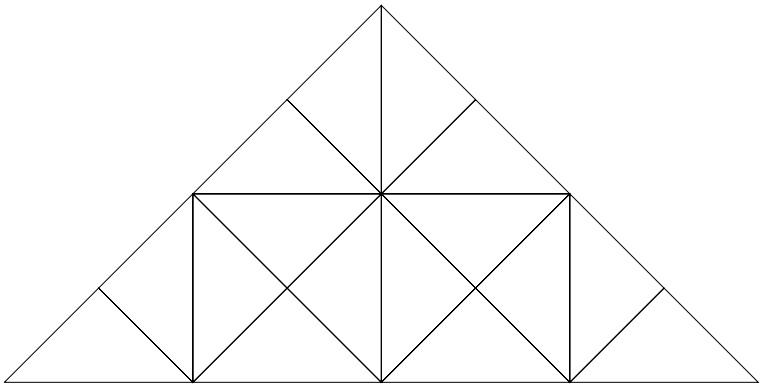

Figure 8: Binary subdivision of a domain triangle.

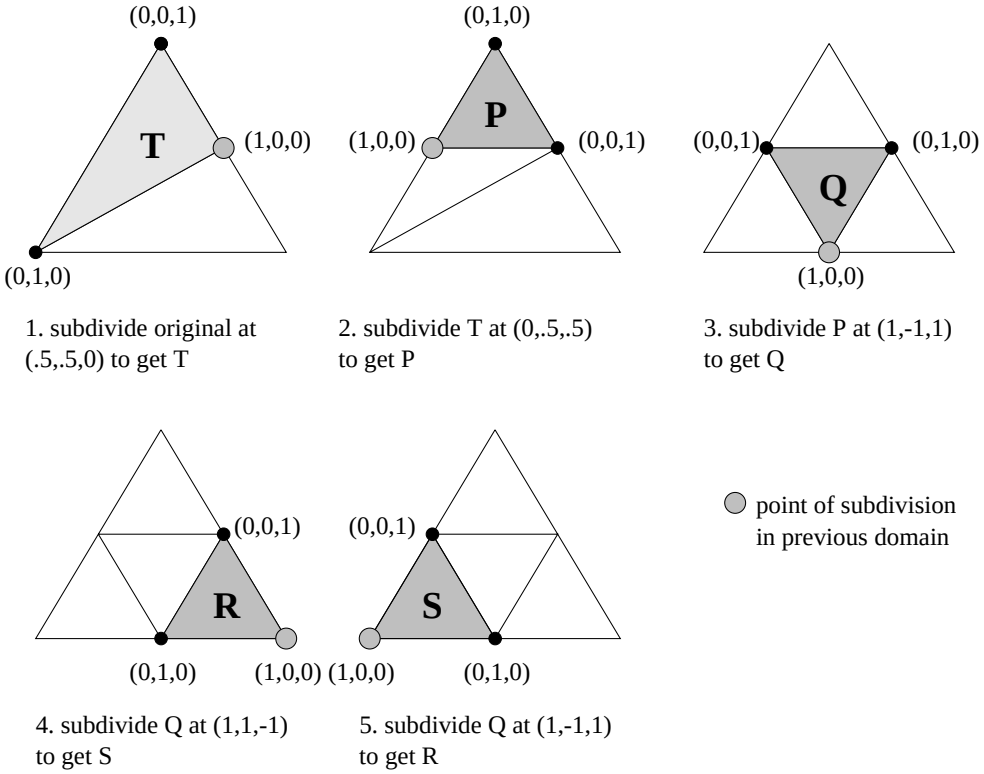

Figure 9: 4:1 subdivision of a domain triangle.

Figure 9 shows a schematic diagram of a 4:1 subdivision of a domain triangle. As before, this process is performed recursively until the patches flat to within a specified tolerance. The pseudocode below uses the `Subdivide` function from above.

4.4.1 Pseudocode

```
ComputeSurface( MSimpBezier& msbezSurface,
                double dTolerance ) {

    // domain coordinates for subdivision points,
    // one for each step
    double adP1[] = {0.5, 0.5, 0.0},
           adP2[] = {0.0, 0.5, 0.5},
           adP3[] = {1.0, -1.0, 1.0},
           adP4[] = {1.0, 1.0, -1.0},
           adP5[] = {1.0, -1.0, 1.0};

    // storage for four new patches of same dimension
    // and degree
    MSimpBezier msbP( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbQ( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbR( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbS( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );

    // storage for four temporaries
    MSimpBezier msbT1( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbT2( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );
    MSimpBezier msbT3( msbezSurface.GetDimension(),
                      msbezSurface.GetDegree() );

    if( Flat( msbezSurface, dTolerance ) ) ) ) {
      msbezSurface.DrawasTriangle();
    } else {

      // step 1, subdivide the original patch at (0.5, 0.5, 0.0)
      Subdivide( msbezSurface, adP1, msbT1, msbT2, msbT3 );

      // step 2, divide Q (T2) again to get one of the new
      // patches (P)
      Subdivide( msbT2, adP2, msbT1, msbP, msbT3 );

      // step 3, divide (flip) P to get one another of the
      // new patches (Q)
      Subdivide( msbP, adP3, msbT1, msbQ, msbT3 );

      // remember: the new patch Q is oriented incorrectly
      // but: the rest of the patches are flips of Q, so they
      // end up oriented correctly

      // step 4, divide (flip) Q to get another (R)
      Subdivide( msbQ, adP4, msbT1, msbT2, msbR );
```

```
        // step 5, again (S)
        Subdivide( msbQ, adP5, msbT1, msbS, msbT2 );

        ComputeSurface( msbR, dTolerance );
        ComputeSurface( msbS, dTolerance );
        ComputeSurface( msbP, dTolerance );

        // now, mark Q as reversed and subdivide it
        msbQ = Reverse( msbQ );
        ComputeSurface( msbQ, dTolerance );
    }
}
```

The function `Reverse` is used to fix the orientation of the new patch Q. This function simply reverses each row of the control net by copying the control points directly into a new simplicial array.

4.4.2 Limitations

While 4:1 subdivision overcomes the poor aspect ratio problem of subdivision at the domain centroid, it still suffers from the "crack" problem of 2:1 subdivision.

4.5 SV-Nested Multiplication

SV-nested multiplication [Schumaker86] is an algorithm based on the efficient evaluation of the *Modified Bernstein-Bézier* (MBB) form. For $d = 2$, we have:

$$B(u) = \sum_{\vec{i},\,|\vec{i}|=n} \frac{n!}{(n-i_0)!(n-i_1)!i_1!} P_{\vec{i}} B^n_{\vec{i}}(u)$$

The conversion from standard Bernstein-Bézier form to MBB form is thus very simple. Schumaker then observes that evaluation can be performed efficiently by writing B in nested form. For $n = 2$, we may write:

$$B(u) = u_0^2 \left[\frac{u_1}{u_0} \left(\frac{u_1}{u_0} P_{(0,2,0)} + \frac{u_2}{u_0} P_{(0,1,1)} + P_{(1,1,0)} \right) + \frac{u_2}{u_0} \left(\frac{u_2}{u_0} P_{(0,0,2)} + P_{(1,0,1)} \right) + P_{(2,0,0)} \right] \tag{1}$$

Because this formula has a singularity when $u_0 = 0$, it is used only when $u_0 > u_1$ and $u_0 > u_2$. One of the two following formulae is used when u_1 or u_2 is largest.

$$B(u) = u_1^2 \left[\frac{u_0}{u_1} \left(\frac{u_0}{u_1} P_{(2,0,0)} + \frac{u_2}{u_1} P_{(1,0,1)} + P_{(1,1,0)} \right) + \frac{u_2}{u_1} \left(\frac{u_2}{u_1} P_{(0,0,2)} + P_{(0,1,1)} \right) + P_{(0,2,0)} \right] \tag{2}$$

$$B(u) = u_2^2 \left[\frac{u_0}{u_2} \left(\frac{u_0}{u_2} P_{(2,0,0)} + \frac{u_1}{u_2} P_{(1,1,0)} + P_{(1,0,1)} \right) + \frac{u_1}{u_2} \left(\frac{u_1}{u_2} P_{(0,2,0)} + P_{(0,1,1)} \right) + P_{(0,0,2)} \right] \tag{3}$$

The pseudocode below simply generalizes the above formula to arbitrary degree.

4.5.1 Pseudocode

```
Point Compute( MSimpBezier& msbezSurface,
               double dP1, double dP2 ) {
    int i, j;
    int iDegree = msbezSurface.GetDegree();
    int iDegree_fact = Factorial( iDegree );
    double dP3 = 1.0 - dP1 - dP2; // barycentric coordinates

    MSimpBezier msbNew( msbezSurface.GetDimension(),
                        msbezSurface.GetDegree() );
```

```
// convert to MBB form
for( i=0; i <= iDegree; i++ ) {
  for( j=0; j <= i; j++ ) {
    double dCoeff = (double)iDegree_fact/((double)Factorial( iDegree-i ) *
                                          (double)Factorial( i-j ) *
                                          (double)Factorial( j ) );

    msbNew.SetControlPoint( iDegree-i, i-j,
                            msbezSurface.GetControlPoint( iDegree-i,
                                                          i-j )*dCoeff );
  }
}

if( (dP1 >= dP2 )
    && (dP1 >= dP3 ) ) {
  // region 1

  double dP2P1 = dP2/dP1, dP3P1 = dP3/dP1;
  Point ptA = msaNew.GetPoint( 0, iDegree );

  for( i=1; i <= iDegree; i++ ) {
    Point ptB = msaNew.GetPoint( 0, iDegree-i );

    for( j=1; j <= i; j++ ) {
      ptB = ptB * dP3P1 + msaNew.GetPoint( j, iDegree-i );
    }
    ptA = ptA * dP2P1 + ptB;
  }

  return( ptA*pow( dP1, (double)iDegree ) );
} else if( (dP2 > dP1)
           && (dP2 >= dP3 ) ) {
  // region 2

  double dP1P2 = dP1/dP2, dP3P2 = dP3/dP2;
  Point ptA = msaNew.GetPoint( iDegree, 0 );

  for( i=1; i <= iDegree; i++ ) {
    Point ptB = msaNew.GetPoint( iDegree-i, 0 );

    for( j=1; j <= i; j++ ) {
      ptB = ptB * dP3P2 + msaNew.GetPoint( iDegree-i, j );
    }
    ptA = ptA * dP1P2 + ptB;
  }

  return( ptA*pow( dP2, (double)iDegree ) );
} else if( (dP3 > dP1)
           && (dP3 > dP2) ) {
  // region 3

  double dP1P3 = dP1/dP3, dP2P3 = dP2/dP3;
  Point ptA = msaNew.GetPoint( iDegree, 0 );

  for( i=1; i <= iDegree; i++ ) {
    Point ptB = msaNew.GetPoint( iDegree-i, i );
```

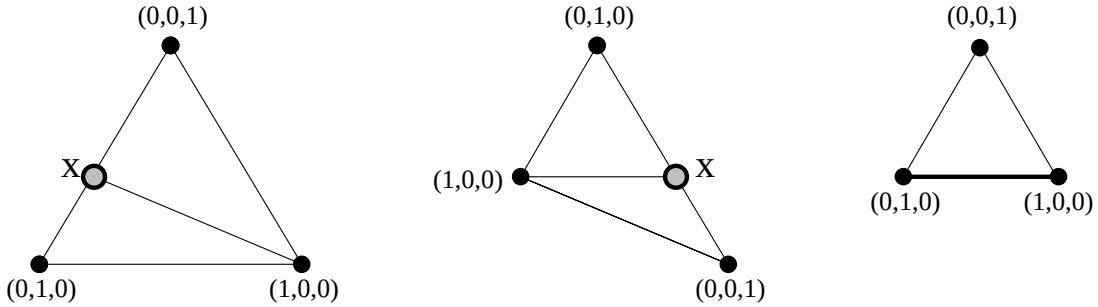

Figure 10: Domain subdivision for isoparametric evaluation.

```
        for( j=1; j <= i; j++ ) {
          ptB = ptB * dP2P3 + msaNew.GetPoint( iDegree-i, i-j );
        }
        ptA = ptA * dP1P3 + ptB;
      }

      return( ptA*pow( dP3, (double)iDegree ) );
    }
}
```

4.5.2 Limitations

The factorials that must be computed by this method may appear to be a problem as a 32 bit integer can only represent up to 13!. However, the greatest factorial that is computed is $n!$, where n is the degree of the polynomial, and one rarely uses patches of such high degree. Also, in order to compute normals with this method, they must be estimated using adjacent points, or computed using the two surfaces of one lower degree representing the partial derivatives.

4.6 Isoparametric Evaluation

Isoparametric evaluation is one of two algorithms presented by Peters [Peters94], both based on curve evaluation. While the version discussed here only considers the two-dimensional simplex ($d = 2$) case, the generalized algorithm is applicable to "surfaces" over simplices of arbitrary dimension. Also, Peters observes that, when evaluating on the edge of a domain simplex, one can subdivide a polynomial over an m-simplex using only the univariate de Casteljau algorithm. For the purposes of this paper, the subdivision algorithm from above will be used instead. This change will not make a significant difference when evaluating surfaces of low degree.

In the two-simplex case, isoparametric evaluation evaluates a triangular Bézier surface by computing curves that lay on the surface and then evaluating the curves. Two subdivision operations are performed, followed by a restriction of the domain to a single edge — thus producing a curve on the surface. See Figure 10. The curve is then evaluated using the univariate de Casteljau algorithm.

4.6.1 Pseudocode

```
ComputeSurface( MSimpBezier& msbezSurface,
                int iNum_steps_1, int iNum_steps_2 ) {
    int i, j;
    MSimpBezier msbezCurve( 1, msbezSurface.GetDegree() );
    MSimpBezier msbezScratch( 1, msbezSurface.GetDegree() );
```

```
    Point aaptComputed_points[iNum_steps_1+1][iNum_steps_2+1];

    // vary x to choose different curves on surface
    for( i=0; i <= iNum_steps; i++ ) {
      Slice( msbezSurface, (double)i/(double)iNum_steps,
             msbezCurve );

      // sample the curve
      for( j=0; j <= iNum_steps; j++ ) {

        aaptComputed_points[i][j] = CurvedeCasteljau( msbezCurve,
                                                      (double)j/(double)iNum_steps,
                                                      msbezScratch, msbezScratch );
      }
    }
}

void Slice( MSimpBezier& msbezSurface,
            double dX,
            MSimpBezier& msbezOut ) {

    int iDegree = msbezSurface.GetDegree();

    // two subsurfaces, d=2
    MSimpBezier msbezSurface_1( 2, iDegree );
    MSimpBezier msbezSurface_2( 2, iDegree );

    MSimpBezier msbezScratch( 2, iDegree );

    // barycentric coordinates of subdivision point
    double adP[3] = {0.0, 0.0, 0.0};

    // first subdivision, take ``P''
    adP[0] = dX; adP[1] = 0.0; adP[2] = 1.0-dX;
    Subdivide( msbezSurface, adP, msbezSurface_1, msbezScratch, msbezScratch );

    // second subdivision, take ``R''
    adP[0] = 0.0; adP[1] = dX; adP[2] = 1.0-dX;
    Subdivide( msbezSurface_1, adP, msbezScratch, msbezScratch, msbezSurface_2 );

    // now, extract the curve by taking the correct points
    // from the control net
    int i=0, j=1, k;

    for( k=0; k < iDegree+1; k++ ) {
      msbezOut.SetControlPoint( k, msbezSurface_2.GetControlPoint( i ) );
      i += j;
      j++;
    }
}

Point CurvedeCasteljau( MSimpBezier& msbezCurve,
                        double dX ) {

    // assume msbezCurve is _really_ a curve, ie. m=1
    int i, j;
```

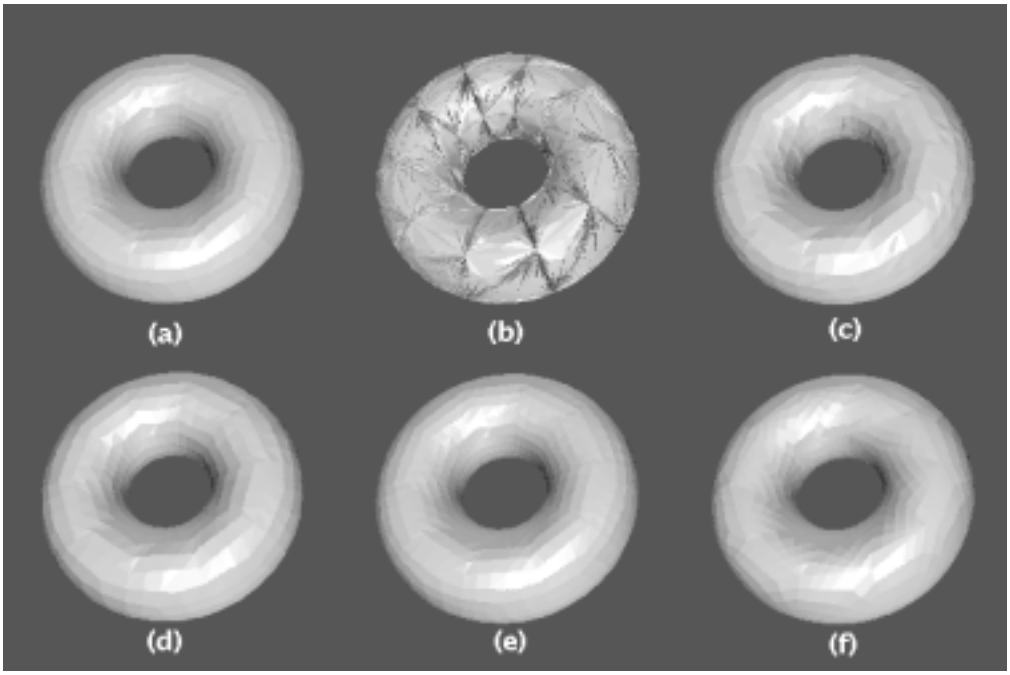

Figure 11: Various evaluations of a torus.

```
    int iDeg = msbezCurve.GetDegree();
    MSimpBezier msbezTemp = msbezCurve; // for intermediate curves

    for( i=iDeg; i >= 0; i-- ) {
      for( j=0; j < i; j++ ) {
        msbezTemp.SetControlPoint( j,
                                   msbezTemp.GetControlPoint( j )*dX
                                   + msbezTemp.GetControlPoint( j+1 )*( 1 - dX ) );
      }
    }

    return( msbezTemp.GetControlPoint( 0 ) );
}
```

To complete the algorithm, one need only generate polygons by attaching adjacent points from `aaptComputed_points`.

4.6.2 Limitations

The primary limitation with the isoparametric evaluation algorithm is the difficulty involved in computing normals — the algorithm does not readily yield derivative information. Peters suggests two possible fixes: "evaluation with different choices of fixed parameters or storing and connecting the surface points to analyze the divided differences of the piecewise linear approximant."

4.7 Comparison

In order to compare the techniques described above, each was used to evaluate two data sets: a torus composed of cubic patches and the same torus with the cubic patches degree raised to quartic. Figure 11 shows the flat shaded results and Table 3 describes the parameters used for evaluation along with the time (to the nearest tenth of a second) taken by each technique.

Scheme				Cubic		Quartic	
Evaluation Technique	Parameters	Tessellation Required?	Image	Time (s)	Avg Polygons Per Patch	Time (s)	Avg Polygons Per Patch
de Casteljau	iNum_steps_u = 5 iNum_steps_v = 5	yes	(a)	1.2	25	1.8	25
centroid subdivision	dTolerance = .02	no	(b)	2.5	68.2	3.9	63
2:1 subdivision	dTolerance = .02	no	(c)	1.4	19.3	2.2	18.8
4:1 subdivision	dTolerance = .02	no	(d)	1.9	16	3.2	16
SV-nested multiplication†	iNum_steps_u = 5 iNum_steps_v = 5	yes	(e)	1.1	25	2.2	25
SV-nested multiplication‡	iNum_steps_u = 5 iNum_steps_v = 5	yes	-	2.4	25	4.7	25
isoparametric evaluation†	iNum_steps_u = 5 iNum_steps_v = 5	no (?)	(f)	1.7	25	2.4	25

Table 3: Comparison of evaluation techniques for cubic and quartic triangular Bézier patches. (†no normals, ‡exact normals)

The times shown in the these tables include only tessellation (if required) and evaluation. They do not include time for computing normals from derivative vectors. The times for recursive subdivision techniques measure only the cumulative time of the subdivision operations themselves.

As before, comparing times between evaluation techniques is difficult since we have no metric for comparing the resulting images. We can ignore the subdivision at the domain centroid algorithm as it produces a very poor final result. SV-nested multiplication and de Casteljau evaluation appear to be comparable techniques in terms of speed, with de Casteljau performing better for quartics and SV-nested multiplication performing better for cubics. However, SV-nested multiplication does not by default compute normal information. Computing normals with SV-nested multiplication by calculating two partial derivative surfaces increases its time by a factor of two (approximately).

Therefore, de Casteljau evaluation is the fastest technique. If an adaptive technique is desired, 2:1 subdivision is the best choice, assuming cracks are acceptable.

The table also shows the average number of polygons produced per patch. These values are useful in comparing the final tessellations produced by the adaptive versus the non-adaptive schemes; using this data one could adjust the tolerances for the adaptive schemes so that they produce approximately the same number of polygons in the final tessellation as the non-adaptive schemes (although the adaptive schemes will potentially need fewer polygons than the non-adaptive schemes to produce the same quality surface). Also, we notice that the average number of polygons produced per patch is slightly lower when using the degree raised data. This should be expected as the quartic data set approximates the surface better than the cubic data set. As a final note, the large number of polygons required by centroid subdivision was due to the difficulties in flatness criteria discussed earlier.

4.8 Higher Degree

Peters gives a thorough treatment of the asymptotic behaviour of the algorithms discussed here (among others) in [Peters94]. As mentioned previously, Peters' implementation of the binary subdivision and isoparametric evaluation techniques performs exclusively curve subdivisions, rather than the generalized subdivisions used in the implementations presented here. This implementation difference will play a role as degree is increased.

In summary, Peters shows binary subdivision subdivision to be the fastest technique by a large margin as degree increases. Equilateral (4:1) subdivision is next fastest, followed by isoparametric evaluation.

4.9 Other Techniques

Techniques not discussed here include the Difference Interpolation Method (D.I.M.), presented in the univariate form by Volk [Volk88] and a forward differencing method discussed by de Boor. [de Boor78]. According to Peters, the

forward differencing method becomes unstable at degree 7, and the D.I.M. method experiences an overflow condition at degree 9.

References

[Barsky87] B. Barsky, T. DeRose, M. Dippé, "An Adaptive Subdivision Method with Crack Prevention for Rendering Beta-spline Objects", Report UCB/CSD 87/348, Department of Computer Science, University of California, Berkeley, CA, 1987

[Böhm83] W. Böhm, "Subdividing multivariate splines", *Computed Aided Design* **15**, Number 6, November 1983, pp. 345-352.

[de Boor78] C. de Boor, "A Practical Guide to Splines", *Applied Mathematical Sciences*, Volume 27, Springer-Verlag, New York.

[Chang89] S. Chang, M. Shantz, R. Rocchetti, "Rendering cubic curves and surfaces with Integer Adaptive Forward Differencing", *Computer Graphics*, **23**, Number 3, November 1989, pp. 157-166.

[Farin93] G. Farin, *Curves and Surfaces for CAGD*, Third Edition, Academic Press, Inc., San Diego, 1993.

[Foley90] J. D. Foley et al., *Computer Graphics Principles and Practice*, Addison-Wesley Publishing Company, Reading Massachusetts, 1990.

[Lien87] S. L. Lien, M. Shantz, V. Pratt, "Adaptive forward differencing for rendering curves and surfaces", *Computer Graphics* **21**, Number 4, July 1987, pp. 111-118.

[Mann95] S. Mann, T. DeRose, "Computing values and derivatives of Bézier and B-spline tensor products", *Computer Aided Geometric Design* **12**, Number 1, February 1995, pp. 107-110.

[Peters94] J. Peters, "Evaluation and approximate evaluation of the multivariate Bernstein-Bézier form on a regularly partitioned simplex", *ACM Transactions on Mathematical Software*, Volume 20, Number 4, December 1994, pp. 460-480.

[Rockwood87] A. Rockwood, "A Generalized Scanning Technique for Display of Parametrically Defined Surfaces", IEEE Computer Graphics and Applications, Volume 7, Number 8, August 1986, pp. 15-26.

[Shantz88] M. Shantz, S. Chang, "Rendering Trimmed NURBS with Adaptive Forward Differencing", *Computer Graphics*, **22**, Number 4, August 1988, pp. 189-198.

[Schumaker86] L. L. Schumaker, W. Volk, "Efficient evaluation of multivariate polynomials", *Computed Aided Geometric Design* **3**, pp. 149-154.

[Volk88] "An efficient raster evaluation method for univariate polynomials", *Computing* **40**, pp. 163-173.

[Williams88] A. W. Williams, "A Comparison of Tensor Product B-Spline Surface Evaluation Methods", Master's Thesis, University of Waterloo, 1988.