Parametric Surface Representation for Visualization of Joint Biomechanics

An Undergraduate Thesis Prepared for

Kent A. Stevens

Professor of Computer and Information Science

University of Oregon

Prepared by

Eric Wills

Undergraduate

Computer and Information Science

University of Oregon

May 25, 2000

Table of Contents

Parametric Surface Representation for Visualization of Joint Biomechanics

Abstract

In vertebrates, articulation is achieved at joints where a pair of smooth surfaces are in gliding contact within a synovial capsule. The paired surfaces comprise a system whose geometry constrains delimit the joint's range of movements. The biomechanical design of a given joint can be explored by creating a 3D model of the component surfaces and visualizing their relative movements from differing perspectives. Greater understanding comes from systematically varying the geometry of the articular surfaces. To explore the space of geometrical designs, software for parameterically representing surfaces has been developed and integrated within a larger software framework for articulating skeletal systems.

Joint Biomechanics and Parametric Surfaces

The neck of the sauropod dinosaur *Diplodocus carnegii* contains 15 elongate cervical vertebrae. Articulation between each pair of vertebrae involves three synovial joints, a large central ball-and-socket (condyle/cotyle) plus two pair of zygapophyseal joints (two prezygapophyses at the anterior of one vertebra articulate with a pair of postzygapophyses at the posterior of the next vertebra. The pair of surfaces, or facets, within each joint are encapsulated within a synovial capsule, the surrounding ligaments of which ultimately limits the travel of one facet relative to the other. The shape of the joint surfaces defines the form of movement achieved within this envelope.

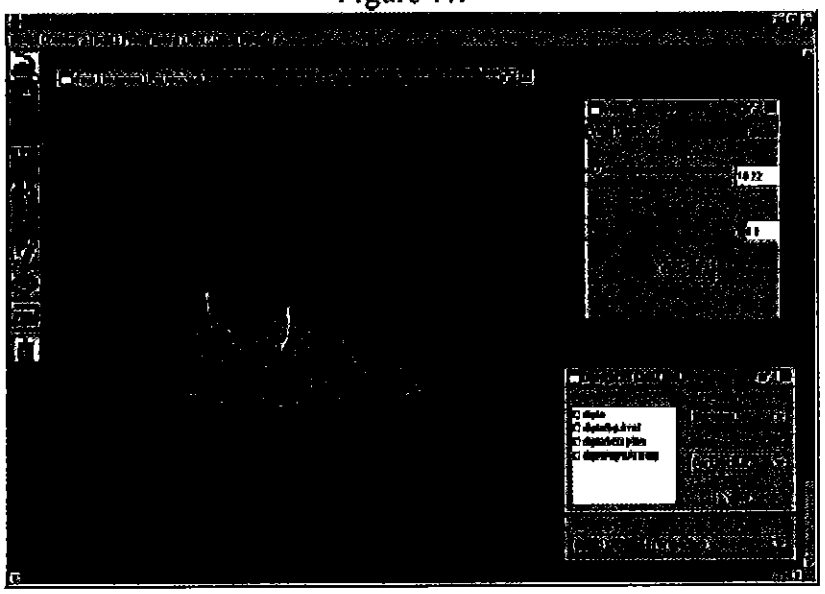
Joint biomechanics is partly a matter of geometry. Within the lubricating synovial capsule a pair of surfaces are in gliding and rotational contact. The functional significance of their geometry is not well understood. Clearly some joints have a deeply cupped, ball-and-socket (condyle/cotyle) arrangement. Others such as within the knee, are clearly designed for one surface to roll across the other (Stevens & Parrish, 1999).

A particularly interesting case is the system of three joints between successive vertebrae, for taken together there is an instantaneous axis of rotation that involves up to six surfaces (within three joints). Visualizing their movements is challenging, even when physically manipulating the neck bones of existing animals. The DinoMorph Project (Stevens, dinoMorph.html) is studying the skeletal structures of dinosaurs. Currently the emphasis is on the necks of the giant sauropods, the longest and most complex found in any vertebrate.

While manipulating a pair of cervical vertebrae gives some rough understanding of their functional morphology in the case of a giraffe or rhino, that is not an option for a fossilized sauropod dinosaur. The bones are far too brittle, massive, and distorted to permit direct manipulation. The alternative is to reconstruct a digital representation of their geometry and place these models into articulation. By parametrically defining 3D surfaces one can produce a close likeness to the shapes observed in fossil specimens. This is an economical, and often more practical means for digital modeling than direct 3D scanning, especially when the fossil material is distorted. With that in mind, Professor Kent Stevens, my advisor for this thesis, and I began development of a parametric surface editor.

In parallel to the development of the surface editor, Professor Stevens and I began making modifications to DinoMorph, a computational paleontology tool that allows scientists to view and manipulate dinosaur skeletons in 3D. We began making changes to the code that allowed the visualization of the vertebral joints. In addition, DinoMorph's own capabilities for picking and manipulating joints were improved. Figure 1.1 shows a *Diplodocus* skeleton visualized using DinoMorph.

Figure 1.1



With a surface editor to model the surfaces, and DinoMorph articulate the surfaces, research could then be done on the interactions between the joint surfaces. The next few sections will detail the development of the surface editor, and the use of DinoMorph to view and manipulate the vertebral joints.

Evolution of the Joint Surface Editor

In the process of modeling biological joint surfaces, the Joint Surface Editor's methods of defining a surface have changed several times. The definition of a surface boundary has evolved from a bitmap editor, to a Bezier curve boundary, and on to a Catmull-Rom spline boundary. The interpolation of the points across the surface has matured from piecewise linear interpolation, to Bezier curves, and on then to a combination of Bezier curves and Catmull-Rom splines. At each step in the development process, the tools for user interaction were enhanced, until an editor was created that

could quickly and efficiently model a joint surface. This tool became known as the Joint Surface Editor, or JSed.

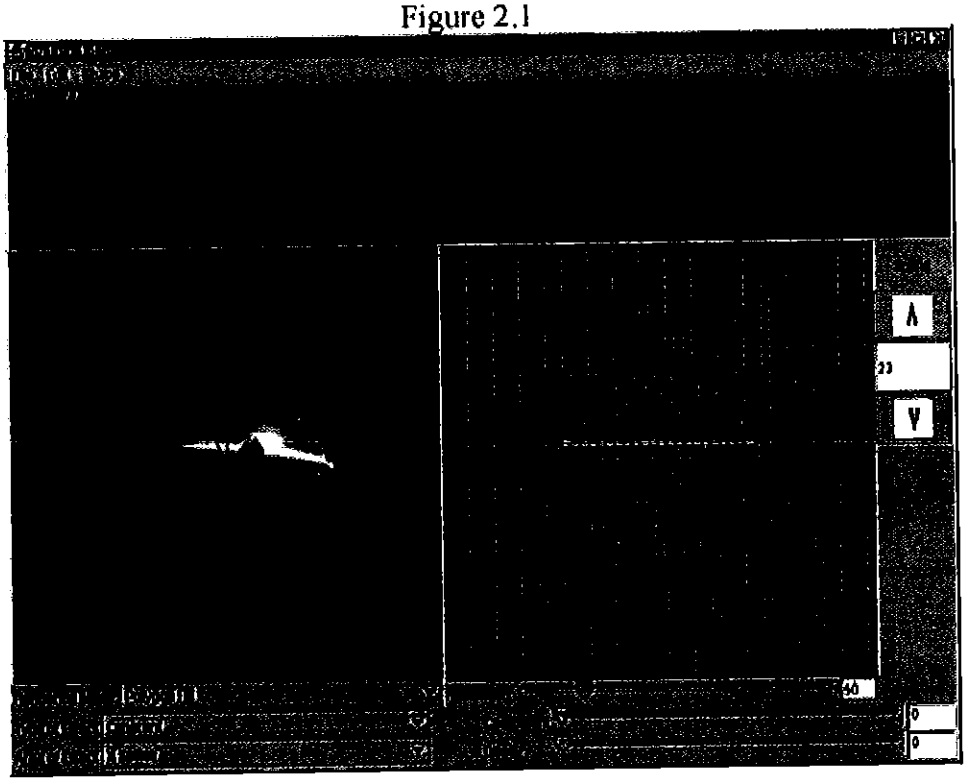
The development of JSed closely resembled a spiral evolution model. During each phase, Professor Stevens and I would define requirements, build a prototype, and then attempt to create joint surfaces using the prototype. The process of building surfaces using these various editors proved invaluable in determining the feature set for the next phase.

The evolution of JSed occurred in four major phases, each phase building upon the strengths and avoiding the weaknesses of the previous prototype. This process produced four distinct prototypes, which will be discussed in the following sections.

Piecewise Linear Interpolation with a Bitmap Editor

The first version of JSed allowed the user to define the surface boundary using a bitmap editor that represented the x-y plane. The points on the surface were defined using piecewise linear interpolation in both the x and y directions. The user was able to select any of the horizontal lines that made up the bitmap, and view that line as a spline in side view (the x-z plane). Using a spline editor written by fellow undergraduate John Bates, the user was able to change the z values of the currently selected spline. A 3D representation of the surface was also displayed in the window. The user could rotate and translate the 3D surface using the mouse. The bitmap editor and the spline editor were both written in Java using Swing. The 3D view was written in Java using Java3D, with the surface represented as a series of triangle strips. Java3D provides an object to hold and display these strips of triangles, called a TriangleStripArray. Figure 2.1 shows version 1 of JSed with the bitmap editor, the spline editor, and the 3D view.

Figure 2.1



The primary flaw with this prototype was in adjusting the amplitude of the surface. The x and y boundaries were easily created using the bitmap, but adjusting the z value of any one point on the surface would create a spike. Figure 2.1 illustrates this effect. Painstaking effort was required to perfect even a small portion of the segment. As a result, Professor Stevens and I decided to add some area effect tools. Figure 2.1 shows the GUI controls for some of these tools at the bottom of the screen.

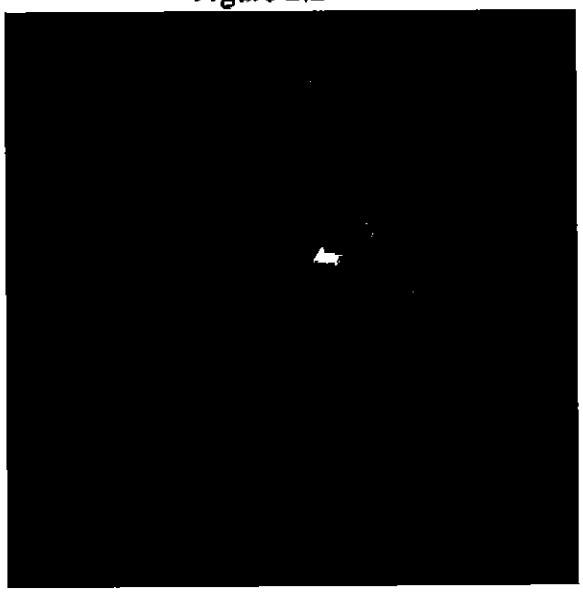
The first addition was a tool that applied a segment of a sine curve to the amplitude of the shape. It was thought, at the time, that the user could apply a sine curve, and then apply area effect tools to specific regions of the surface to model the desired shape. These area effect tools would raise and lower regions of the surface in either a constant, linear, or an exponential fashion. The user would be able to control the area of effect, or the rate of decay of the effect.

The first and only area effect tool created had a constant effect on the currently selected horizontal spline. When a point on the positive side of the x axis was raised or lowered, all points to the right of that point (points with greater x values) would be raised and lowered with the selected point. In other words, the difference between the selected point's original z value and its new z value would be added to the z values of all points

with a greater x value than the currently selected point. Likewise, when a point on the negative side of the x axis was raised or lowered, the points to the left of that selected point would be raised and lowered with the point.

The constant effect tool seemed to work at first, but we soon realized that it was easy to create creases in the shape, and difficult to smooth out these creases. Again, it took painstaking effort to adjust all of the individual splines to create a smooth surface, let alone the desired surface. Figure 2.2 shows a surface created by applying a sine curve, and then using the constant effect tool to raise a region of the surface. It is apparent that smoothing out this surface would require a great deal of work.

Figure 2.2



The difficulties in creating surfaces using piecewise linear interpolation of splines led us to research other forms of interpolation. We researched several curve types, but concluded that Bezier curves might give the user the added control necessary to accurately model a joint surface.

A Single Bezier Surface Patch

Around the same time that we had decided to try using Bezier curves, I was thinking of possible final project topics for a Computer Graphics course I was taking. The course had covered Bezier curves and surfaces earlier in the term, so Professor Gary Meyer agreed to let me implement a Bezier surface editor for my final project.

Since the interface for the previous editor seemed sound, I began writing a new editor with roughly the same look and feel as the old editor. This new editor, however, had completely different methods for boundary definition and surface interpolation. Like the version 1 of JSed, this new editor had three main windows: a window looking down on the surface (the x-y view), a window looking at the side view of the currently selected curve (the x-z view), and a window containing the 3D view of the shape. Again, the user was able to rotate and translate the surface using the mouse. The x-y window and x-z window were both written in Java using Swing and the 3D window was written in Java using Java3D. Again, a TriangleStripArray was used to display the surface. Figure 2.3 shows version 2 of JSed, with its three main windows and its simplified interface.

Figure 2.3

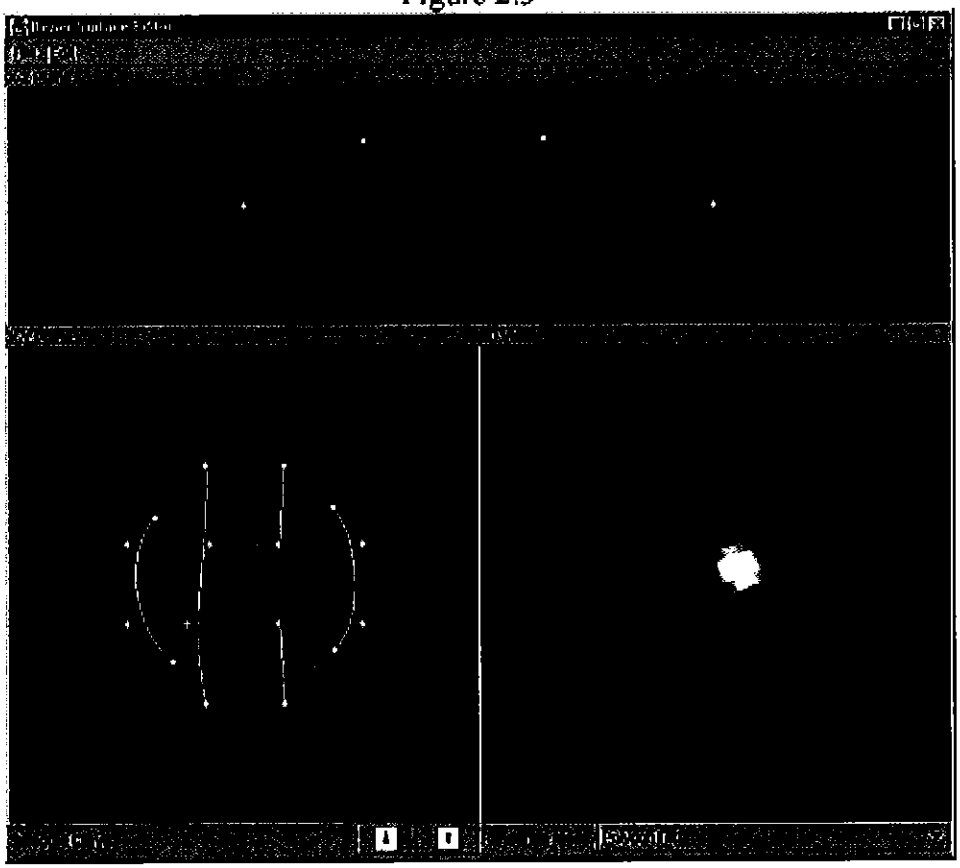
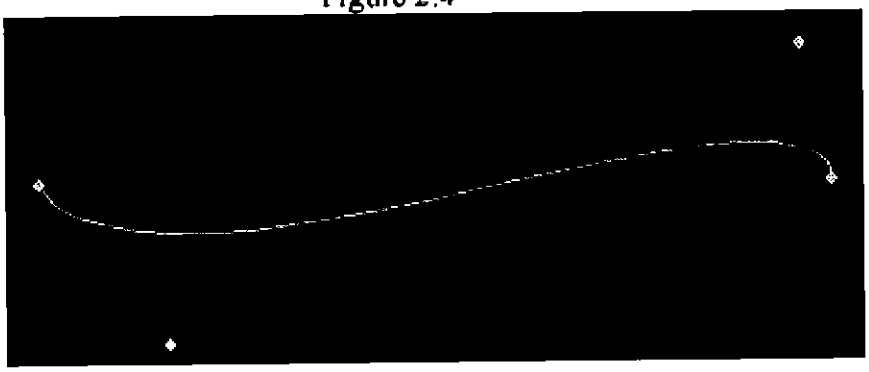


Figure 2.3 shows the sixteen control points defining the surface in the x-y view. In the x-y view, the user was able to manipulate the boundary of the surface by clicking and dragging one of the control points on the boundary. By clicking the up and down arrows at the bottom of the screen, the user was able to select which curve would be visible in the x-z view. Clicking and dragging any of the points in the x-z view would

change the surface amplitude. A Bezier surface is continuous with only sixteen control points, so it was easy to define a surface and impossible to crease the surface.

Of the sixteen control points, each set of four control points in the horizontal and vertical direction made up one Bezier curve, for a total of eight Bezier curves. A Bezier curve is a cubic polynomial curve segment made up of four points, P1, P2, P3, and P4. The starting and ending tangent vectors of the curve are defined by the vectors <P1 P2> and <P3 P4>. This means that P1 and P4 are the starting and ending points of the curve, and are therefore interpolated by the curve. The points P2 and P3 are not interpolated by the curve, as they define the vectors tangent to the curve at P1 and P4. The four control points and their resulting Bezier curve can be clearly seen in Figure 2.4.

Figure 2.4



Several techniques for drawing Bezier curves and surfaces exist. The technique discussed in my Computer Graphics class was recursive subdivision, one of the easiest methods to implement and most efficient to run. The technique involves breaking up a curve into two segments, and determining the control points of those two curves. We then subdivide each of those curves into two smaller curves. After just a few subdivisions, interpolating all of the new control points will approximate the Bezier curve specified by the original four control points. I wrote the following procedures to return the points on a curve. The method getCurrentPoints is called with the number of subdivisions, which in turn calls the method subdivideCurve to subdivide the curve. The method subdivideCurve calls itself recursively until the curve reaches its desired order. The points on the curve are then returned.

```
private Point3d[] controlPoints =  new Point3d[4];
private Point3d[] curvePoints;
private int       index;
```

```
// gets the points on the curve based on a number of subdivisions
public Point3d[] getCurvePoints(int numDivisions) {

   curvePoints = new Point3d[(int)(Math.pow(2, numDivisions+2))];
   index = 0;
   subdivideCurve(controlPoints, numDivisions, 0);

   return curvePoints;

} // method getCurvePoints

// subdivides a curve from four points to eight points
private void subdivideCurve(Point3d[] points, int numDivisions, int order) {

   // checks to see if the order is still less than the desired number of
   // subdivions
   if (order < numDivisions) {

      // arrays to hold the subdivided pieces
      Point3d[] a = new Point3d[4];
      Point3d[] b = new Point3d[4];
      for (int i = 0; i < 4; i++) {
         a[i] = new Point3d();
         b[i] = new Point3d();
      } // for

         // subdivides the curve
         // calculates a[0] as points[0]
         a[0].x = points[0].x;
         a[0].y = points[0].y;
         a[0].z = points[0].z;

         // calculates a[1] as (points[0] + points[1])/2
         a[1].x = (points[0].x + points[1].x)/2;
         a[1].y = (points[0].y + points[1].y)/2;
         a[1].z = (points[0].z + points[1].z)/2;

         // calculates a[2] as a[1]/2 + (points[1] + points[2])/4
         a[2].x = a[1].x/2 + (points[1].x + points[2].x)/4;
         a[2].y = a[1].y/2 + (points[1].y + points[2].y)/4;
         a[2].z = a[1].z/2 + (points[1].z + points[2].z)/4;

         // calculates b[2] as (points[2] + points[3])/2
         b[2].x = (points[2].x + points[3].x)/2;
```

```
            b[2].y = (points[2].y + points[3].y)/2;
            b[2].z = (points[2].z + points[3].z)/2;

            // calculates b[3] as v[3]
            b[3].x = points[3].x;
            b[3].y = points[3].y;
            b[3].z = points[3].z;

            // caulculates b[1] as (points[1] + points[2])/4 + b[2]/2
            b[1].x = (points[1].x + points[2].x)/4 + b[2].x/2;
            b[1].y = (points[1].y + points[2].y)/4 + b[2].y/2;
            b[1].z = (points[1].z + points[2].z)/4 + b[2].z/2;

            // calculates a[3] as (a[2] + b[1])/2
            a[3].x = (a[2].x + b[1].x)/2;
            a[3].y = (a[2].y + b[1].y)/2;
            a[3].z = (a[2].z + b[1].z)/2;

            // calculate b[0] as a[3]
            b[0].x = a[3].x;
            b[0].y = a[3].y;
            b[0].z = a[3].z;

          // recurse on each piece
          subdivideCurve(a, numDivisions, order+1);
          subdivideCurve(b, numDivisions, order+1);

        } // if

        // else store the piece in the point array
        else {

          // stores the points
          for (int i = 0; i < 4; i++) {
            curvePoints[index++] = points[i];
          } // for

        } // else

      } // method subdivideCurve
```

After writing procedures to get the points on a Bezier curve, the next step was to write procedures to get the points on the Bezier surface defined by the sixteen control

points. Since recursive subdivision was used to interpolate the curves, it made sense to use the same technique to interpolate the surface. I used the recursive subdivision algorithm that was presented by Professor Meyer in my Computer Graphics class. The algorithm used the procedures above to split each of the four horizontal curves into two separate curves, creating a total of eight points along each horizontal curve. In effect, this subdivision created eight vertical curves. For each of the eight vertical curves, the above procedures were again used to divide each curve into two curve segments. The result was a new patch containing four smaller patches. Each of these patches was then subdivided, and so on until the desired surface order was reached. After just of a few subdivisions, the surface would contain enough points to appear smooth. The 3D figure in Figure 2.3 is the product of three subdivisions. The surface has a total of 256 points.

Professor Stevens and I then began testing the new editor. We were delighted to see that it had many of the desired surface interpolation properties. Most joint surfaces are fairly simple, and the editor allowed us to quickly create a simply surface. We found, however, that creating the desired surface boundary shapes was not as easy.

Joint surfaces usually have complex boundaries. Using a single Bezier curve for each of the four sides of the surface allowed for just four inflection points, namely at the four corners of the shape. Without inflection points along the sides of the surface, it was difficult to create a concave shape, and impossible to create an accurate boundary for certain joint shapes. Using this editor, we were able to model condyles and cotyles, but not zygapophyses. To solve the boundary problem, we decided to add a bitmap editor to the current editor with the hope that it would allow us to fine-tune the boundary of the surface.

A Bezier Surface Patch with a Bitmap Editor

The next incarnation of JSed involved taking the implementation for the previous phase, augmenting it, and adding a bitmap editor. The first major task was to rewrite the surface interpolator so that it could produce a surface of n x n points for any integer n within practical limits. With an arbitrary surface resolution, we could then use a bitmap editor to "cut" a shape out of the patch. Using recursive subdivision, the number of points was bound by the number of subdivisions. The number of points on the surface

was equal to 4^n where n was the number of subdivisions. After reading up on efficient interpolation techniques, I decided to employ a technique knows as forward differences. To understand how the forward differences technique works, we must first look at how a cubic curve is calculated by brute force. A point on a Bezier curve is defined parametrically as follows:

The Bezier geometry matrix is
Gb = [P1x, P1y, P1z, 1
P2x, P2y, P2z, 1
P3x, P3y, P3z, 1
P4x, P4y, P4z, 1]

The Bezier basis matrix is
Mb = [-1, 3, -3, 1
3, -6, 3, 0
-3, 3, 0, 0
1, 0, 0, 0]

A point Q is defined by
Q(t) = [t^3, t^2, t, 1] * Mb * Gb

The forward differences technique is an improvement on the brute force method. Where the brute force method requires 10 additions and 9 multiplies to derive each 3D point, forward differences requires only 9 additions per 3D point (Foley et al. 1990). The elimination of the multiplies greatly improves the efficiency of interpolating a curve.

Forward differences uses a new matrix called a difference matrix, created with a specified curve resolution. Again, we multiply the geometry and basis matrices to get the coefficient matrix, but now we multiply the coefficient and difference to get a set of deltas. The deltas can then be used to successively derive each point on the curve. The difference matrix is computed as follows:

S = 1/curve resolution

The difference matrix is
E = [0, 0, 0, 1
S^3, S^2, S, 0
6S^3, 2S^2, 0, 0
6S^3, 0, 0, 0]

```
D = E * Mb * Gb

The first column of D is Dx = x, dx, d2x, and d3x
The second column of D is Dy = y, dy, d2y, and d3y
The third column of D is Dz = z, dz, d2z, and d3z
```

I used the following code to return the points on a curve using forward differences:

```
// an array of 3d points to hold the individual points on the curve
Point3d[] curvePoints = new Point3d[res+1];

// stores the first point on the curve
curvePoints[0] = new Point3d(x, y, z);

// loops through the rest of the points on the curve
for (int i = 1; i <= res; i++ ) {

   //move to next points through deltas
   x += dx;  dx += d2x;  d2x += d3x;
   y += dy;  dy += d2y;  d2y += d3y;
   z += dz;  dz += d2z;  d2z += d3z;

   // stores the current point
curvePoints[i] = new Point3d(x, y, z);

} // for

   // returns the points on the curve
return curvePoints;
```

Using Bezier curves with n points on each curve, I was then able to create surfaces with n x n points, for any integer n. To create the surfaces, I used a technique similar to the recursive subdivision technique used in version 2. The new technique involved calculating the points on each of the four curves in the horizontal direction. These points were then stored in a point array of size 4 x n. The points on each of the n vertical curves were then calculated, and stored in an array of size n x n. With the surface containing n x n points, an n-1 x n-1 bitmap was able to control the starting and ending points of each triangle strip on the 3D. By controlling the starting and ending points of

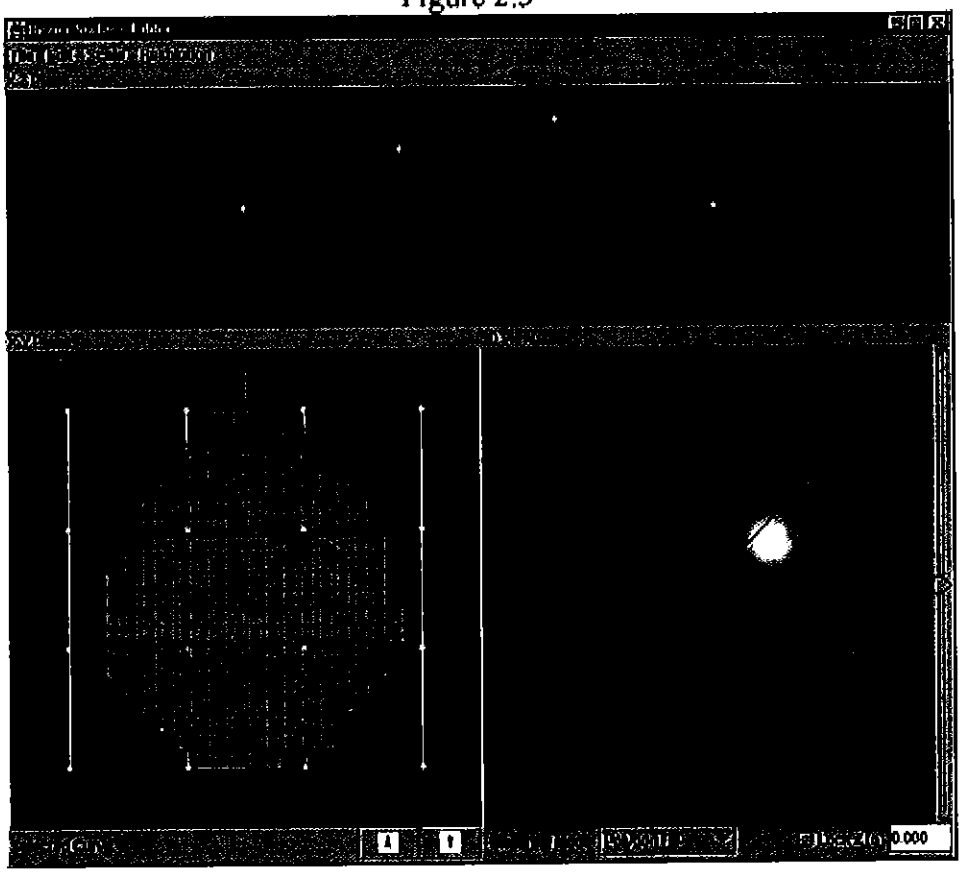
each triangle strip, a shape could be "cut" out of the Bezier patch. I used the following code to generate the points on the surface:

```
// gets the points on the surface based on a surface resolution
public Point3d[][] getSurfacePoints(int res) {

  Point3d[][] s             = new Point3d[4][res+1];
  Point3d[][] surfacePoints  = new Point3d[res+1][res+1];

  // calculates the curves in the s direction
  for (int i = 0; i < 4; i++) {

    // gets the curve and stores it in s
    s[i] = new BezierCurve(controlPoints[i]).getCurvePoints(res);

  } // for

  // calculates curves in the t direction
  for (int i = 0; i <= res; i++) {

    // gets the points in the t direction
    Point3d[] tControlPoints = {s[0][i], s[1][i], s[2][i], s[3][i]};

    // gets the curve and stores it in surfacePoints
    Point3d[] t = new BezierCurve(tControlPoints).getCurvePoints(res);

    for (int j = 0; j <= res; j++) {

      // stores t[j] in surfacePoints[j][i]
      surfacePoints[j][i] = t[j];

    } // for

  } // for

  return surfacePoints;

} // method getSurfacePoints
```

The interface for this version of the editor resembled a combination of the interface from versions 1 and 2. The editor had the same look and feel as version 2 of the editor, but contained a bitmap editor in the x-y view like the first version. The user could

select one of the four horizontal Bezier curves to view and manipulate in the x-z view. As with version 1 and 2, a three-dimension window allowed the user to rotate and translate the surface. A new feature was added that would allow the user the to lock the center of the surface to any z value. This feature allowed the user to create a surface with any amplitude and ensure that the center of the surface would remain at the origin of the surface's local space. The three main windows, along with the interface controls of version 3 are shown in Figure 2.5.

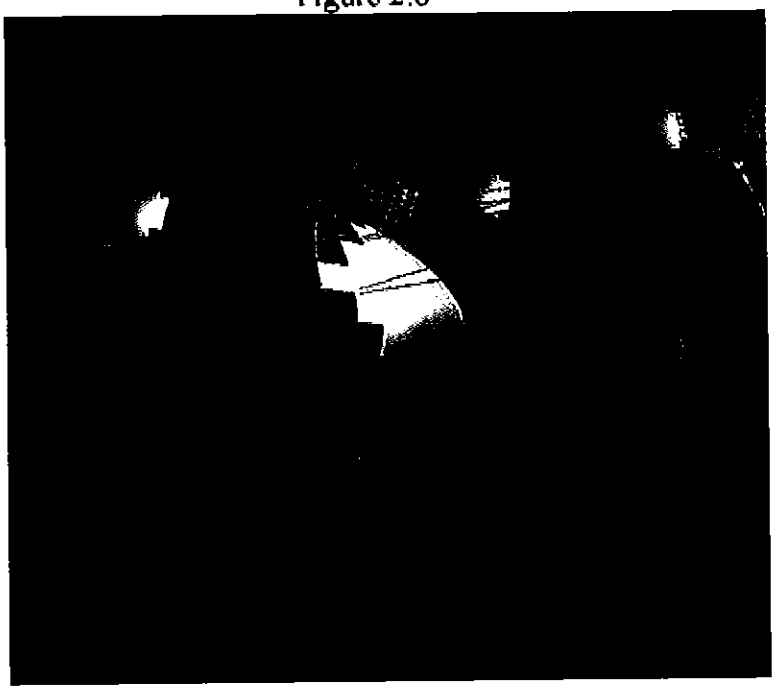
Figure 2.5



Upon testing this editor, Professor Stevens and I thought we had a winner. We began modeling zygapophyses, and were pleased with the resulting shapes. We didn't run into any trouble until we created a condyle for cervical vertebra 15 in the cervical vertebrae of a *Diplodocus*. Upon seeing the results in DinoMorph, it became appearant that the method for defining the surface boundary needed to be refined. The condyle had a relatively steep amplitude. The steep amplitude caused the surface, which was cut out by the bitmap editor, to have inconsistent z values around its boundaries. This inconsistency had not been a problem when modeling zygapophyses, because the

zygapophyses have small amplitudes. Figure 2.6 illustrates the problem with inconsistent z values, and how using a bitmap editor to define a boundary causes the surface to have rough, blocky edges.

Figure 2.6



We decided that we needed a smooth boundary for the surfaces. The smooth boundary of the second prototype was pleasing with its four Bezier curves, but they didn't give the user enough control over the boundary. After some discussion, a consensus was reached: using two Bezier curves for each side of the boundary would give the user an extra inflection point and maintain a smooth boundary.

Catmull-Rom Spline and Bezier Curve Surface Interpolation with a Catmull-Rom Spline Boundary

For the next prototype of JSed, the bitmap editor was removed and curves were again used describe the boundary of the surface. This time, however, we used more control points to describe the surface. In order to accommodate two Bezier curves per boundary side, seven control points were needed along each side. With seven control points along each boundary side, the surface was defined by a total of 49 control points. Each row and column of seven points described a curve defined by two Bezier curve segments, for a total of twenty-eight Bezier curve segments. The advantages and

disadvantages of using two Bezier curve segments to describe a curve were almost immediately obvious.

On the positive side, the endpoint of the first Bezier curve segment describing the curve was the same as the starting point of the second curve segment. The points being equal ensured that the curve was C0 continuous, meaning that there were no breaks in the curve. The point where the endpoint of the first curve segment and the starting point of the second curve segment met was the fourth of the seven control points, and therefore the midpoint of the curve. Since Bezier curves interpolate their endpoints, the midpoint of the curve was interpolated. With the midpoint of each curve interpolated, the user had even more precise control over each curve, and control over key points such as the point at the center of the surface.

On the negative side, the third, fourth, and fifth point on each curve had to remain collinear. If these three points were not collinear, meaning that they did not lie on the same line, the curve would lose its C1 continuity. C1 continuity indicates that a curve preserves slope through all of its control points. If a curve was not C1 continuous, a crease would appear in the curve. With a crease in any one curve, the surface would also crease.

Enforcing C1 continuity was easy, but it was difficult to control the surface with continuity enforced. Controlling each curve remained intuitive, but manipulating the z value of a point on a horizontal curve could have drastic effects on the z value of a point on another horizontal curve. This effect was caused because C1 continuity needed to be preserved not only in the horizontal direction, but also in the vertical direction. The third, forth, and fifth point on each vertical curve also needed to remain collinear. The result was that the user would raise one region of the surface, only to inadvertently lower another region. This lack of control over the surface made it almost impossible to accurately model a shape. Figure 2.7 shows a curve composed of two Bezier curve segments, with C1 continuity enforced.

Figure 2.7

It was evident that curves composed of two Bezier curve segments could not be used to interpolate the entire surface. This conclusion was unfortunate because the curves composed of two Bezier curve segments provided the user with a quick and efficient way of approximating the cross section of a shape. It seemed that we could use another type of curve to describe the boundary and vertical curves, while still using curves comprised of two Bezier curve segments for the horizontal curves.

I found a type of spline described in Foley, van Dam, Feiner, and Hughes (1990) known as the Catmull-Rom spline. The Catmull-Rom spline had two important benefits that made it attractive: the spline interpolates all but the first and last of its control points, and it maintains C1 continuity. Continuity is preserved because the curve passes through each control point in a direction parallel to the line connecting the points on either side of the control point. For a Catmull-Rom spline described by m points, the curve is composed of m-3 curve segments, each defined by a separate geometry matrix. The Catmull-Rom geometry and basis matrix are as follows:

The Catmull-Rom geometry matrix is

$$G_{cs} = \begin{bmatrix} P_{i-3x}, & P_{i-3y}, & P_{i-3z}, & 1 \\ P_{i-2x}, & P_{i-2y}, & P_{i-2z}, & 1 \\ P_{i-1x}, & P_{i-1y}, & P_{i-1z}, & 1 \\ P_{ix}, & P_{iy}, & P_{iz}, & 1 \end{bmatrix}$$

The Bezier basis matrix is

$$M_{cs} = \begin{bmatrix} -1, & 3, & -3, & 1 \\ 2, & -5, & 4, & -1 \\ -1, & 0, & 1, & 0 \\ 0, & 2, & 0, & 0 \end{bmatrix}$$

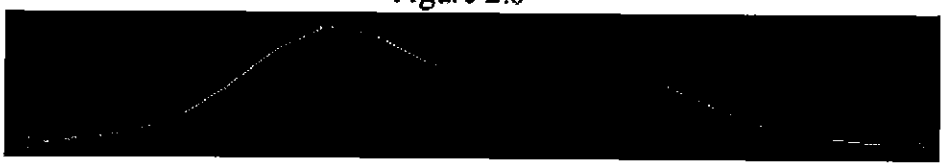
A point Q is defined by

$$Q(t) = [t^3, t^2, t, 1] * \tfrac{1}{2} M_{cs} * G_{cs}$$

Using the forward differences technique discussed earlier, I was able to efficiently interpolate the curve segments between the second and second-to-last control points. By adding two more control points, both invisible to the user, I was able to produce a curve that interpolated all of its visible control points. The first invisible control point was collinear with the first and second control points of the spline, forcing the first segment of the curve to leave the first visible control point tangent to the vector between the first and

second visible control points. Recall that this is exactly the way that a Bezier curve leaves its first control point. Likewise, the second invisible point was collinear with the last and second to last visible control points. A Catmull-Rom spline is pictured in Figure 2.8.

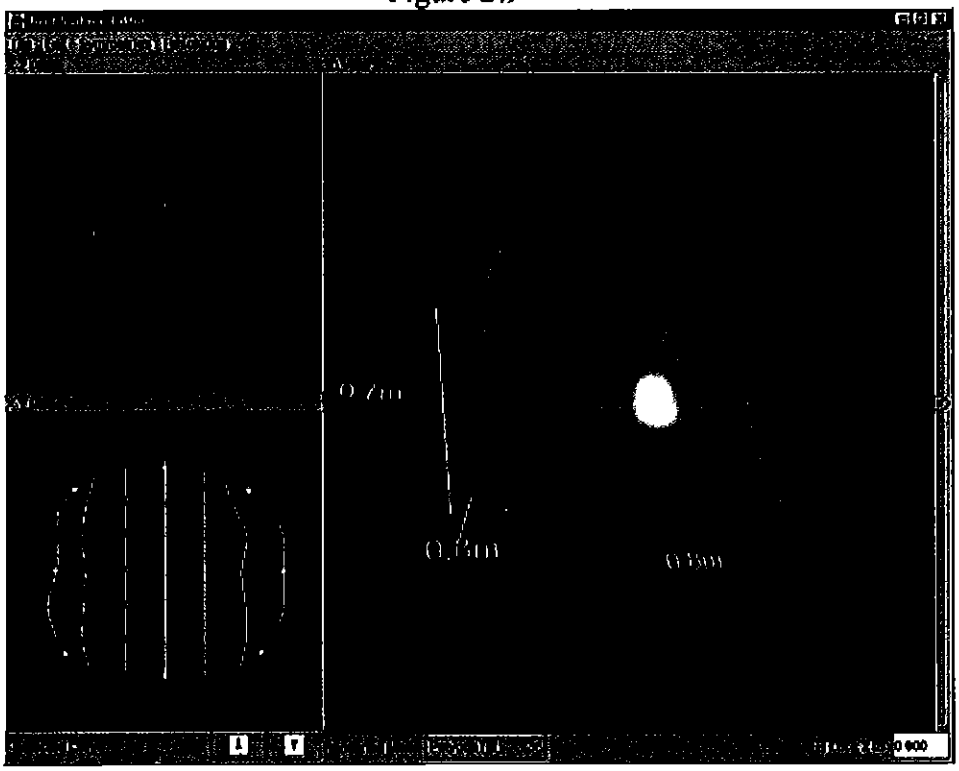
Figure 2.8



The Catmull-Rom splines interpolated all of their visible control points, meaning that each of their visible control points is an inflection point. It seemed that a method for defining the boundary of a surface had finally been found. By making slight modifications to the surface interpolation technique employed in version 3, I was able to create a shape that was interpolated using both Bezier curves and Catmull-Rom splines. Catmull-Rom splines were used to describe all seven of the vertical curves and the two horizontal boundary curves. C1 continuous curves comprised of two Bezier curve segments were used to describe the five interior horizontal curves. By using Catmull-Rom splines to define the boundary, seven inflection points were available per boundary side, and each boundary side remained C1 continuous. By using Catmull-Rom splines for interpolating the five interior vertical curves, the user was assured that manipulation of a point on a horizontal curve would have almost no effect on any other horizontal curve. With a new method for defining the boundary of the surface, as well as a new method for interpolating the surface, it was time to build a new prototype.

Professor Stevens and I agreed that the new version of JSed was capable of synthesizing the shapes we desired. With that in mind, we worked on perfecting the user interface. Functionality was added to prompt the user for the width, length, and height of the shape whenever a new shape was created. The width, length, and height information was added to the 3D window in the form of calipers. Several other modifications were made, including changing the layout of the windows, adding point picking and manipulation capabilities to the 3D window, and providing symmetry tools. Figure 2.9 depicts the final version of JSed, using both Catmull-Rom splines and Bezier curves, with the user interface improvements and the symmetry tools.

Figure 2.9



Previous versions of JSed used a x-z view window with width greater than its height. To accurately model a cross-section of a shape, we needed a view with an aspect ratio of one. Due to this constraint, the x-z view window was moved on top of the x-y view window, and the 3D window was enlarged so that all three windows have an equal width and height.

To give the user increased control over the boundary of the surface, point picking and manipulation was added to the three dimensional window. Small spheres were added around the boundary of the surface. These spheres can be picked and dragged around by the user, manipulating the boundary. Changes to the boundary in the three-dimension window are reflected by changes in the x-y view window. The 3D picking was accomplished by modifying the mouse picker code written by fellow undergraduate Niels Albarran.

To assist the user in the rapid synthesis of a surface, two symmetry tools were added. An option was added to enforce symmetry along the x-axis and an option was added to enforce symmetry across the y-axis. With the x-axis symmetry enabled, the regions of the surface on either side of the x-z plane are identical. Likewise, with y-axis

symmetry enabled, the regions of the surface on either side of the y-z plane are identical. With both symmetries enabled, all four quadrants of the surface are identical.

With these modifications, JSed has reached a stable form. JSed can quickly and efficiently model biologically accurate joint surfaces. Functionality has been added to save and load surfaces in .pat format, and to export surface in .obj format. The .obj files outputted by JSed can be read into LightWave, or any Java3D application, such as DinoMorph. With an editor to synthesize joint surfaces, the surfaces and they ways in which they interact can now be studied.

Modeling Biological Joint Surfaces

The initial modeling of the joint surfaces occurred in two phases. The first phase involved creating accurate representations of the condyles, cotyles, and zygapophyses using JSed. For the second phase, the shapes were assembled to create accurate reproductions of vertebrae. With the vertebrae constructed, we were able to study the ways in which the joint surfaces interact under manipulation of a vertebral joint. These two phases occurred in parallel, with surfaces being refined as we visualized them as parts of vertebrae.

When dealing with fossils, it is difficult to articulate pairs of vertebrae because of postdepositional distortion. Professor Stevens and I decided to model the thirteenth and fourteenth cervical vertebra (C13 and C14) of the *Diplodocus* neck because they have little obvious distortion. The condyle of C14 fits well into the cotyle of C13, and the prezygapophyses of C14 fit well with the postzygapophyses of C13. Finding a pair of vertebra that fit together was the first step to understanding they ways in which they interact. Using several photocopies of the vertebrae (Hatcher, 1901; plates III-VI) we set out to model the surfaces.

Modeling the Joint Surfaces using JSed

The first surface modeled was the right postzygapophysis of C13. Using Two photocopies of C13, a lateral view and a posterior view, we were able to estimate the width, length, and height of the postzygapophysis. From the lateral view, we were also able to see the boundary of the surface. From the posterior view, we were able to see the

curvature of the postzygapophysis. Using the two photographs, we were able to model the surface using JSed. Since the postzygapophyses of C14 and the prezygapophyses of C13 have similar surfaces, C14's right postzygapophysis surface was used to represent the prezygapophyses of C13. Figure 3.1 shows the lateral view photocopy of the right postzygapophysis, and a lateral view of the surface created with JSed.

Figure 3.1

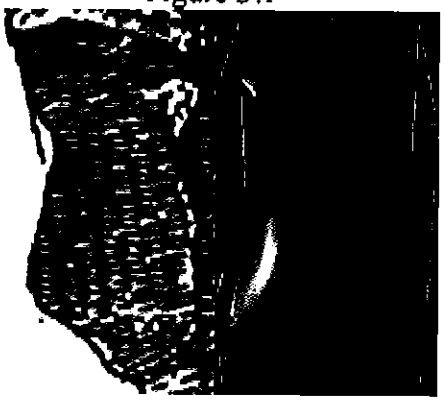


Figure 3.2 depicts the posterior view photocopy of the right postzygapophysis, along with a posterior view of the surface created with JSed.

Figure 3.2

The second shape modeled was the condyle of C14. Using JSed, we modeled the shape in much the same way we modeled right postzygapophysis of C13. By using a photograph of the lateral view of the condyle, we were able to capture the curvature of the shape. With a photograph of the anterior view of the condyle, we were able to visualize the boundary of the surface. Figure 3.3 illustrates the lateral view photograph of the condyle along with a lateral view of the surface created by JSed.

Figure 3.3

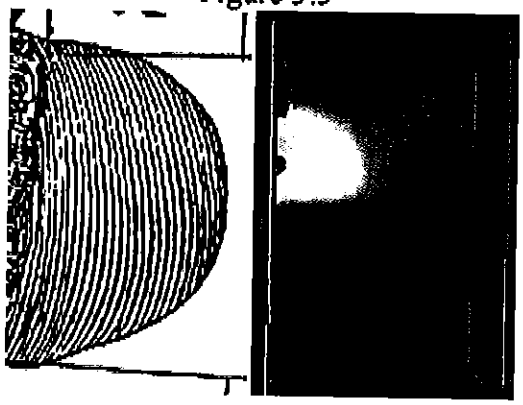
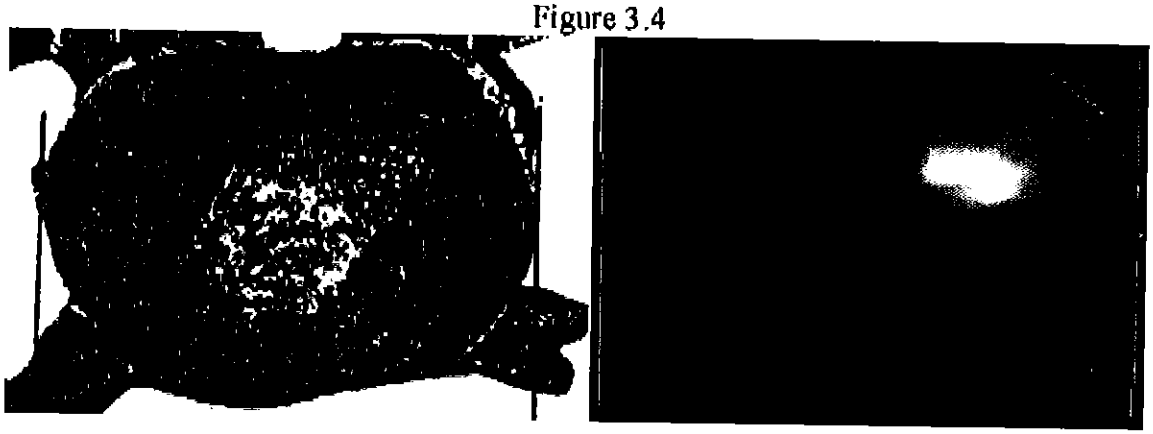


Figure 3.4 shows the anterior view photograph of the condyle next to an anterior view of the surface created using JSed.

Figure 3.4



The cotyle of C13 has a curvature almost identical to that of the condyle of C14; the cotyle of C13 was synthesized by simple modification to the boundary of the condyle of C14. A photograph of the posterior view of C14 aided in the modification to the boundary. Figure 3.5 shows the posterior view photograph of the cotyle of C14, along with a posterior view of the surface created using JSed.

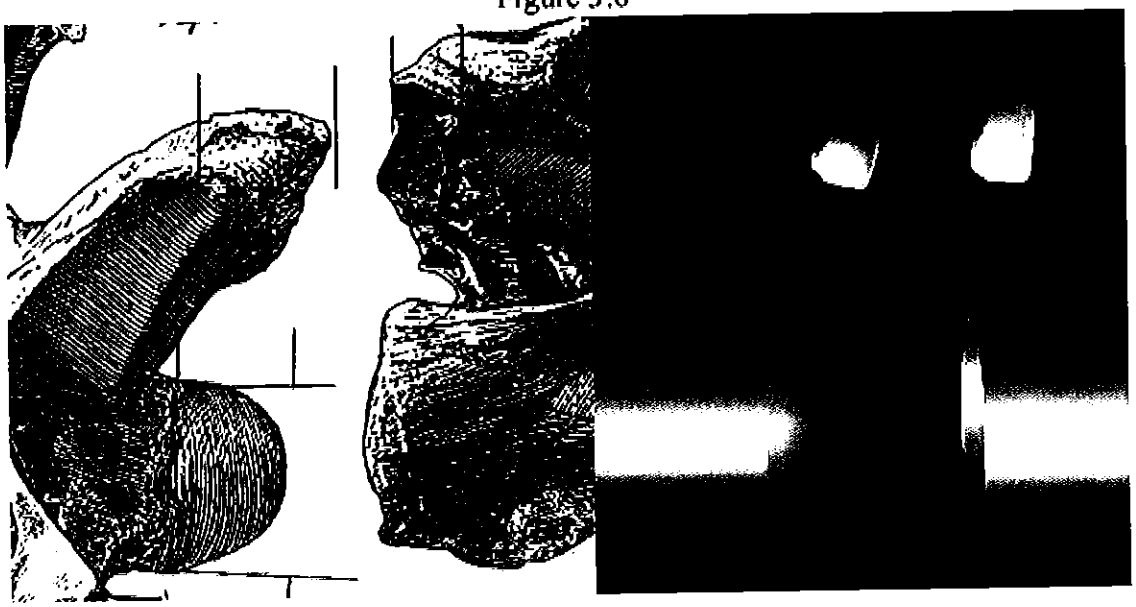
Figure 3.5

Figures 3.1 through 3.5 demonstrate Jsed's capability to synthesize surfaces that accurately represent their biological surface counterparts. With surfaces to portray the prezygapophyses and condyle of C14, and the postzygapophyses and cotyle of C13, we were ready to model the vertebrae C13 and C14.

Analyses of the Joint Surfaces using DinoMorph

Modeling vertebrae containing the joint surfaces was not too complicated once the surfaces had been created using JSed. By augmenting the existing length, width, and height data for each vertebra, along with data on the position and orientation of the prezygapophyses and postzygapophyses, Professor Stevens and I were able to put the shapes together in 3-space. The right prezygapophysis and postzygapophysis were reflected across the y-z plane to create the left prezygapophysis and postzygapophysis. Using a cylinder to represent the centrum of the vertebra, we were able to visualize a vertebra using DinoMorph. Figure 3.6 shows the lateral view photograph of the anterior of C14 and the posterior of C13, along with the vertebrae visualized using DinoMorph.

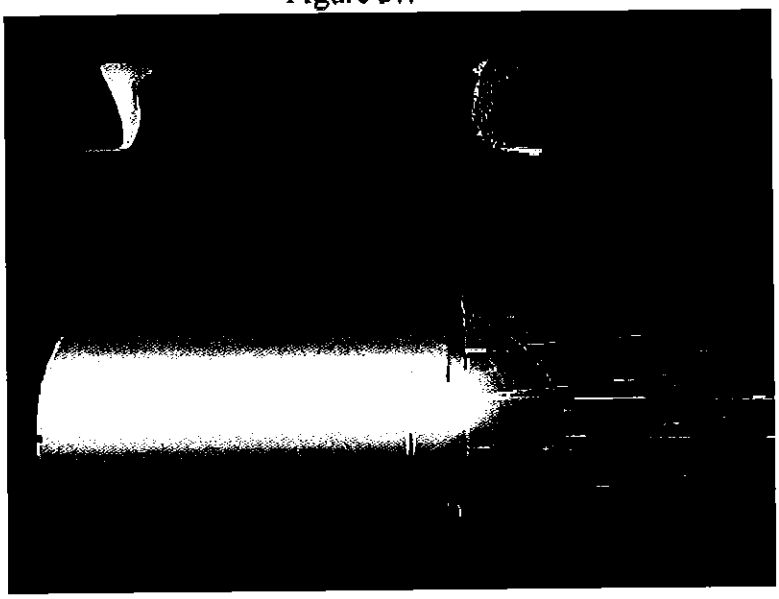
Figure 3.6



Once the vertebrae were loaded into DinoMorph, we were able to adjust the proximal and distal attachments of each vertebra to attach the vertebrae at a joint. We realized that rotating the cotyle about the condyle at the joint caused the condyle and cotyle to collide with even a minor deflection. By moving the centers of rotation about the x and z axis back from the joint, we were able to create a sweeping motion of the

cotyle over the condyle. Tilting the z-axis forward allowed the postzygapophyses to slide over the prezygapophyses. In the process of adjusting the axes, we had created a system that allowed the joint a reasonable amount of deflection. Figure 3.7 shows the lateral view of the coupled vertebrae (C13 is rendered in wireframe) and the adjusted axes (the x-axis is red, the y-axis is green, and the z-axis is blue).

Figure 3.7



With a system for joint rotation established, it was possible to make initial observations about the behavior of a joint. By watching for collisions between the surfaces, we were able to attain some basic information about how the surfaces slide across each other, and the amount of deflection allowed by the joint. Figure 3.8 shows a lateral cross-section of the condyle and cotyle colliding with -6 degrees of deflection about the x-axis.

Figure 3.8

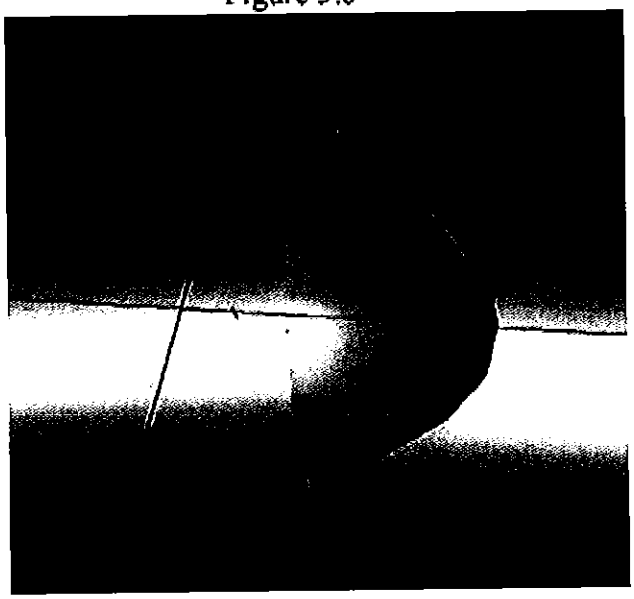


Figure 3.9 illustrates a posterodorsal view of contact between the right prezygapophysis of C14 and the right postzygapophysis of C13 with -6 degrees of deflection about the x-axis and 2 degrees of deflection about the y-axis.

Illinois

Figure 3.9

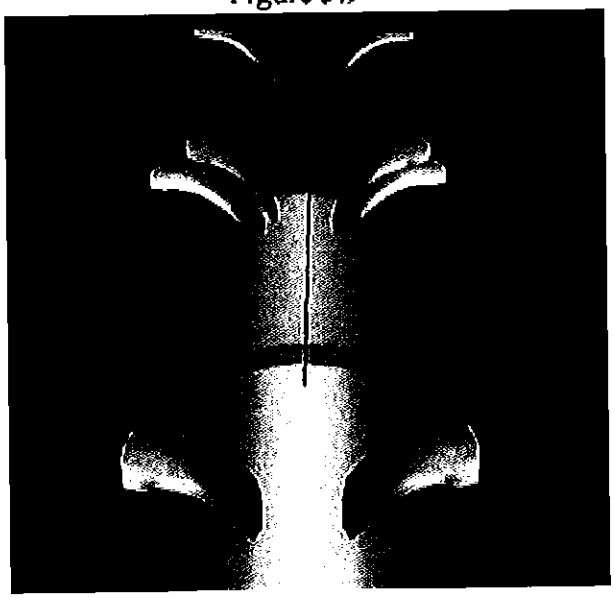
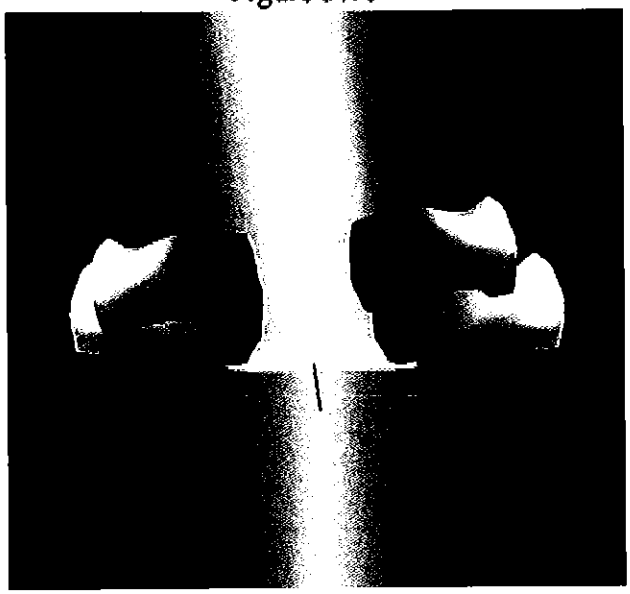


Figure 3.10 depicts a dorsal view of contact between the left prezygapophysis of C14 and the left postzygapophysis of C13 with –6 degrees of deflection about the x-axis and 6 degrees of deflection about the z-axis.

Figure 3.10



Observations such as these are just the first step in understanding the ways in which the joint surfaces interact. Many questions remain about how these surfaces govern the movement of joints. The shape, angle, and position of a vertebra's prezygapophyses and postzygapophyses seem to have as much control over possible deflection as the the vertebra's condyle and cotyle shapes. Further research will show the significance of these shapes, angles, and positions.

The development of JSed allowed us to model the articular surfaces of actual vertebrae using multiple views from the original published monographs. Modifications to DinoMorph allowed us to assemble vertebrae in 3D and to manipulate each vertebral pair. In addition, DinoMorph was modified so that the center of rotation of each axis could positioned and tilted in any direction in order to approximate the actual axes about which rotations occurred. Now that tools exist for accurately creating joint surfaces and for building vertebrae using those surfaces and articulating them, the biomechanical principles underlying their geometry can be further studied.

References

Foley, James, van Dam, Andreas, Feiner, Steven K. and Hughes, John F 1990 Computer Graphics: Practices and Principals. New York: Addison-Wesley.

Hatcher, J.B. 1901 Diplodocus (Marsh): Its osteology, taxonomy, and probable habits, with a restoration of the skeleton. *Memoirs of the Carnegie Museum* **1(1)**.

Stevens, Kent A.1998 *Introduction to the DimoMorph Project.* http://www.cs.uoregon.edu/~kent/dinoMorph.html. Internet

Stevens, K.A. and J.M. Parrish, 1999 Neck posture and feeding habits of two Jurassic sauropod dinosaurs. *Science* **284** 798-800.