Applied geometric modelling, STE6247

Daniel G. Razafimandimby, 500311

Master of Technology, 5 Data/IT,
Narvik University College, Narvik, Norway

December 15, 2014

Abstract

1 Introduction

The goal of this project was to create and implement our own ERBS surface class, complete with evaluators for the ERBS and it's subsurfaces. The part about tesselation was not supposed to be implemented, however the theory shall be covered in this report. This project was written for the course STE6247 Applied Geometry and Special Effects.

1.1 GMlib

GMlib is an open source geometric modelling library developed and maintained at Narvik University College, NUC, by the college's teachers and students. The library contains several modules for creating and visualizing parametrized objects, such as curves, surfaces and volumes.

1.2 Coding convention

The code written for this project follows a mix of my own style of coding and the Google C++ Style Guide.

2 Parametric surfaces

2.1 Bezier and the evaluator

A bezier curve is a parametrized curve with knot vectors which range from 0 to 1 and can be defined for any degree n. Higher dimensions of bezier curves are called bezier surfaces and they are defined by a set of control points and the position of a point P is is given by the parametric coordinates u and v.

There are three different types of evaluators used for Bezier curves: de Casteljau algorithm, a Cox/de'Boor algorithm and computing the Bernstein polynomials directly [Lak07]. In this project, I have used a Cox/deBor algorithm: computing from the left. The algorithm starts by filling in the matrix with the Bernstein polynomial between first and the requested degree.

Listing 1: Assigning the top half of the matrix.

```
//Matrix is the incoming matrix, degree is the
//Bernstein polynomial degree, t is the parameter
//value 0 <= t <= 1 and scaling factor delta.
function compute basis matrix(matrix, degree, t, delta)
  matrix[degree-1][0] = 1 - t
  matrix[degree-1][1] = 1
  for(i = rows - 2; i >= 0; i--)
    matrix[i][0] = (1 - t) matrix[i+1][0]
    for(j = 1; j < degree - i; j++)
      matrix[i][j] = t * matrix[i+1][j-1] + (1-t)
                   * matrix[i+1][j]
    end for
    matrix[i][degree-i] = t * matrix[i+1][degree-i-1]
  end for
```

Following this every row except the first one is multiplied with the delta and derivatives.

Listing 2: Assigning the bottom half of the matrix

```
  matrix[degree][0] = -delta
  matrix[degree][1] = delta
  for(k = 2; k <= degree; k++)
    scale = k * delta
    for(i = degree; i > degree-k; i--)
      matrix[i][k] = scale * matrix[i][k-1]
      for(j = 1; j < degree - i; j++)
        matrix[i][j] = scale * (matrix[i][j-1]
                     - matrix[i][j])
      end for
      matrix[i][0] = -scale * matrix[i][0]
    end for
  end for
end function
```

Because the Bezier surface ranges from 0 to 1, a scaling parameter is introduced and multiplied into every row except the first. It is written by course professor Lakså in a paper from 2010: "*If the domain of the Bezier curve is scaled, as is the norm, because of the global/local affine mapping,*

then, in order to compute, for instance, the local Bezier curve $c_i(t)$, the j-th derivatives actually have to be scaled by the global/local "scaling factor" δ_i^j where

$$\delta_i = \frac{1}{t_{i+1} - t_{i-1}} \tag{1}$$

as described in Theorem 1. The numerator in the fraction is 1 because the domain of the Bezier curves is [0,1]. Because the matrix is supposed to be used both in Hermite interpolatino and in the evaluation of local curvse, this matrix has to include the scaling, as described earlier." [LBD10]. The evaluator creates a basis matrix for both u and v, b1 and b2 respectively, and then multiplies these two, where b2 is transposed, with the matrix containing the control points.

$$p = b1 * c_{controlpoints} * b2^T \tag{2}$$

2.2 B-spline

A b-spline is a curve which has knot definitions ranging from 0 to n. Formula 3 shows the formula for a third degree B-spline curve [Lak07].

$$\begin{aligned} c(t) = \begin{pmatrix} 1 - w_{1,i}(t) & w_{1,i}(t) \end{pmatrix} \begin{pmatrix} 1 - w_{2,i}(t) & w_{2,i}(t) & 0 \\ 0 & 1 - w_{2,i}(t) & w_{2,i}(t) \end{pmatrix} \\ \begin{pmatrix} 1 - w_{3,i}(t) & w_{3,i}(t) & 0 & 0 \\ 0 & 1 - w_{3,i}(t) & w_{3,i}(t) & 0 \\ 0 & 0 & 1 - w_{3,i}(t) & w_{3,i}(t) \end{pmatrix} \begin{pmatrix} c_{i-3} \\ c_{i-2} \\ c_{i-1} \\ c_i \end{pmatrix} \end{aligned} \tag{3}$$

The evaluator for a B-spline is very similar to the Bezier algorithm, however as the B-spline has multiple interior knots which makes it necessary to find the corresponding knot-index to t. Therefore instead of using t in the matrix, we introduce a function w which maps the value between 0 and 1. This function is dependant on the knot index, knot value and the Berstein polynomial degree and is described as [Lak12]

$$w_{d,i} = \begin{cases} \frac{t - t_i}{t_{i+d} - t_i}, \text{if} : t_i \leq t < t_{i+d} \\ 0, \text{otherwise} \end{cases} \tag{4}$$

2.3 Expressional rational B spline (ERBS), and ERBS surfaces

An expressional rational B-spline, ERBS, is a new type of B-spline, purposed by Arne Lakså, Børre Bang and Lubomir T. Dechevsky [LBD05]. The concept is shortly explained in their article, Exploring Expo-Rational B-splines for Curves and Surfaces[LBD]. This was created to cover some of the limitations found in regular B-splines.

An ERBS surface is a surface which consists of an amount of local patches which are created by either using bezier surfaces or sub surfaces. These local patches are created by deciding the size of the grid and if the surface is closed in u- and/or v-direction. This information is then used to construct the knot-vectors which will be the basis for the surface.

The evaulator starts by finding which knot index the u and v value belongs to by checking if $u_{indexU} \leq u < u_{indexU+1}$ and $v_{indexV} \leq v < v_{indexV+1}$, where indexU and indexV are the knot indices. Following this it creates two b vectors, b1 and b2, by using GMlib's own ERBSEvaluator class which gives the value and derivative at the given knot. Then if the local surfaces are Bezier surfaces it maps the u and v parameters between 0 and 1 by the following function:

$$u_{mapped} = \frac{u - u_{indexU-1}}{u_{indexU+1} - u_{indexU-1}} \tag{5}$$

Using these mapped value the $s_{2,2}$, $su_{2,2}$ and $sv_{2,2}$ matrices, which contain positions and derivatives in u- and v-direction, are created. These matrices are then transposed and multiplied with the b vectors to create the positions and their belonging partial derivatives,

Listing 3: Assigning the p matrix

```
this->_p[0][0] = bv * (s ^ bu);
this->_p[0][1] = bv * (s ^ bud) + bv * (su ^ bu);
this->_p[1][0] = bvd * (s ^ bu) + bv * (sv ^ bu);
```

2.4 Tesselation

Tesselation is a technique where we till a plane into different shapes. It is not permitted to have overlapping edges or gaps between the lines which define the shape. Regular tesselations are made up of regular polygons, all meeting vertex to vertex. The most common ways of tesselating is to use a network of triangles or squares.

Irregular tesselations occur when there are no restrictions on the order of the polygons around the vertices.

The Delaunay triangulation algorithm goes as follows:

1. Find convex hull, which is the polygon with teh smallest area which includes every point.
2. Add 3 to 4 new points to create one or two boundary triangles.
3. For each point perform Delaunay.
4. Remove the extra newly added points with their associated edges and triangles.

A Quad tree is a rooted tree where the internal nodes has four children. The algorithm goes as follows:

1. Recursively partition until there are only a given number of points left in the nodes
2. Balance the tree.
3. Add a Steiner point inside the nodes with two neighbours in any direction (east, south, west, north).
4. Make lines to form triangles between the corners and the Steiner point.

This algorithm requires the tree to be balanced, which occurs when every node in the tree has no more than two neighbouring squares in any direction. If the tree is unbalanced, additional nodes must be created [dBCvKO08].

3 My project

3.1 MyERBSSurface

MyERBSSurface is capable of receiving a surface to copy and convert it to an ERBS surface. It can currently use either subsurfaces or bezier surfaces to generate the surface.

MySubsurface is one of the two supported classes which the ERBS surface can use to create local patches. It is constructed using a parametric surface and the evaluation is only done between the knot value of the previous and the next index. When evaluating with this class, it utilizes the surface's evaluateParent function.

MyBezierSurface is the other class which the ERBS surface can use to make the local surfaces. It uses the evaluator function which was described in Section 2.1, and only evaluates between 0 and 1.

3.2 Help classes and containers

MyKnotVector is a simple data container which contains the information needed to create a knot vector.

Animation is a helper class which performs the animations of surfaces. The various types of animations are seperated into individual classes which inherit from a common class.

3.3 My animation/effect

One part of the task was to create a special effect using the self implemented ERBS surface by finding a surface and doing affine operations upon this surface. I found the formula for a fish [Bou02].

$$\begin{aligned} x &= \frac{(cos(u) - cos(2u))cos(v)}{4} \\ y &= \frac{(sin(u) - sin(2u))sin(v)}{4} \\ z &= cos(u) \\ 0 \leq u \leq \pi, 0 \leq v \leq 2\pi \end{aligned} \tag{6}$$

This fish is then to be animated to look, from the front, like it's swimming. Then to add to the effect I will make a flat plane and animate it with waves, to look like water.

4 Conclusion

The implementation of the ERBS-surface with both sub surface and bezier went mostly without problems. The biggest problems I encountered were caused by typos in my code, like by mistake typing u for v and vice versa. The creation of the animation, however was a much bigger challenge than anticipated, as the control points were somewhat difficult to manipulate with the surface I had created. I found the root of the problem to be the fact that the surface is closed in v-direction, which caused the ends of the surface to be translated twice.

The water animation also gave me problems, which I have been unable to find the cause for. The simple function I had created to create the waves were insufficient to make the animation, as the plane would continue to escalate and grow, instead of having a smooth wave pattern. Because of this I simplified it down to just a sinus wave.

The source code for this project can be found on: https://github.com/oxycoon/GeoMod2-BezierBSpline.

References

[Bou02] Paul Bourke. Fish surface, 2002.

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications third edition.* 2008.

[Lak07] Arne Lakså. *Basic properties of Expo-Rational B-splines and practical use in Computer Aided Geometric Design.* 2007.

[Lak12] Arne Lakså. *Blending technics for Curve and Surface constructions.* 2012.

[LBD] Arne Lakså, Børre Bang, and Lubomir T. Dechevsky. Exploring expo-rational b-splines for curves and surfaces.

[LBD05] Arne Lakså, Børre Bang, and Lubomir T. Dechevsky. Expo-rational b-splines, 2005.

[LBD10] Arne Lakså, Børre Bang, and Lubomir T. Dechevsky. Geometric modelling with beta-function b-splines, i: Parametric curves. *International Journal of Pure and Applied Mathematics*, 65, 2010.