From Simple B-Spline to GERBS Surface, or How I Stopped Sleeping and Learned to Hate My Life

D. A. Salwerowicz

UiT - The Arctic University of Norway, P.O. Box 385, N-8505 Narvik, Norway

Submitted 2018-12-12

Abstract

`//TODO`

Keywords: Parametric Curves; B-Splines; Curve Blending; GERBS

1 Introduction

This project revolves around implementing various geometric objects using GMlib as a basis. Each of these object builds on previous therefore I will describe them seperately. Main focus of this project was to implement a GERBS curve and use affine transformations to create a dynamic animation.

1.1 B-Spline curve

Most basic object that I have implemented is a third degree, fourth order B-Spline curve using both a vector of *control points* and *least square approximation.*

1.2 Blending curve

A more advances object implemented by me is a blending curve that takes in two curves and blends them into one curve using any percent of the original curves it wants. So I can use 50% or 75% of these curves and blend the rest.

1.3 GERBS curves and surfaces

Last and most complex objects implemented are a GERBS curve and surface, where GERBS stands for *Generalized Expo-Rational B-Spline.* These objects are build out of local curves and surfaces that are blended together to form a curve/surface, instead of simple control points.

2 Material & Methods

Here I will describe the most important tools and methods used during development of my project. All of the theory in this section is based on book "Blending technics[sic] for Curve and Surface Constructions" written by Arne Lakså (Lakså, 2012).

2.1 GMlib

It's important to mention that I have based my project and work on the Geometric Modeling library also known as GMlib. It was developed at Arctic University of Norway, UiT Narvik. It handles a lot of necessary background calculation and rendering of objects on screen for me. This way I only needed to focus on implementing necessary functionality for calculating and showing B-Spline curves and other objects.

2.2 B-Spline Curves

2.2.1 B-Spline from control points

One can create a B-Spline curve from a vector of *control points* $\vec{c}$, which is then used to create a *knot vector* $\vec{t}$. For any given B-Spline a knot vector will be of length $n + k$ where n is length of control point vector and k is the order. So in my case knot vector will have lenght 12 if I give it 8 control points. Resulting knot vector will look like one shown in equation (1).

$$\vec{t} = [0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5] \tag{1}$$

This B-Spline will be a *Clamped B-Spline* as its first k and last k knots are equal to each other. Then I can use formula equation (2) to calculate my BSpline.

$$c(t) = \sum_{i=1}^{n} c_i b_{k,i}(t), \quad t \in [t_d, t_n] \tag{2}$$

Here ${c_i}_{i=1}^{n}$ is a vector of controll point and $b_{k,i}(t)$ are basis functions that are defined by t (Lakså, 2012, Chap 5.5.1). (Lakså, 2012, Chap 5.5.3) describes in depth how we calculate these basis functions. Resulting basis functions are shown in equation (3), there are four of them, since the order of my B-Spline is 4.

$$\begin{aligned}
b_{k-3,i} &= [(1 - w_{1,i}(t))\,(1 - w_{2,i-1}(t))]\,[1 - w_{3,i-2}(t)] \\
b_{k-2,i} &= [(1 - w_{1,i}(t))\,(1 - w_{2,i-1}(t))]\,[1 - w_{3,i-1}(t)] \\
&\quad + ([(1 - w_{1,i}(t))\,(w_{2,i-1}(t))] + [w_{1,i}(t)\,(1 - w_{2,i}(t))])\,[1 - w_{3,i-t}(t)] \\
b_{k-1,i} &= ([(1 - w_{1,i}(t))\,(w_{2,i-1}(t))] + [w_{1,i}(t)\,(1 - w_{2,i}(t))])\,[w_{3,i-1}(t)] \\
b_{k,i} &= w_{1,i}(t) w_{2,i}(t) w_{3,i}(t)
\end{aligned} \tag{3}$$

$w_{d,i}(t)$ is a so called linear translation and scaling function used in B-Splines to translate the t to a range $[0, 1]$, and is defined in equation (4).

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

Thus using these basis functions I am able to evaluate my B-Spline and draw it on screen which is shown in the first simulation in my program.

2.2.2 Sampled B-Spline

Another way of creating a B-Spline is by sampling points along an existing curve and then using least square method to create control points from a set of points, p. It is quite usefull as it lets us approximate any kind of freeform curve to a high degree of accuracy.

Knot vector is created the same way as before, I just use the provided n to create it. However in order to calculate a vector of control points I have to perform matrix calculations. I first start by calculating an A matrix. Values in it are defined by formula in equation (5), which gives me a least square approximation of the original curve.

$$\partial t = \frac{t_n - t_d}{m - 1} \tag{5}$$

Where m is the dimension of p. Algorithm used to calculate values in A matrix is as shown in algorithm 1.

Algorithm 1: Calculating values for matrix A

```
Data: $m, n, d$
Result: $A$ matrix of basis functions used in calculating control points.
Create an empty 2D matrix with $\dim_1 = n$ and $\dim_2 = m$;
Calculate $\partial t$ using (5);
for $a_1 = 0;\ a_1 < m;\ ++a_1$ do
    Find $i$ using $t = t_d + a_1 \cdot \partial t$;
    Find $\vec{b}$ using (3) with $k = t_d + a_1 \partial t$;
    for $a_2 = i - d;\ a_2 \leq i;\ ++a_2$ do
        $A_{a_1,a_2} = b_{a_2-i+d}$;
    end
end
```

Using this algorithm we end up with a matrix that has non-zero values along its diagonal, and is zero elsewhere. Using this matrix and property:

$$Ac = p$$

I can easily solve for c. However since A is an asymetrical matrix I need to multiply it with itself transposed. So I multiply both side of the equation with A^T and then substitute $A^T A = D$, lastly I multiply both sides of the equation with inverse of D and get:

$$c = D^{-1} b \tag{6}$$

After calculating this, I get a control point vector that I can then use to draw my B-Spline.

2.3 Curve blending

Blended curve is simply a curve that is result of applying a *B-Function* to two or more curves. B-function per definition is a *permutation function*, where $B(0) = 0$ and $B(1) = 1$, it is as well *monotone*, and can be symetric if it satisfies $B(t) + B(1 - t) = 1$. (Lakså, 2012, Chap 6.1)

There are several viable blending functions and ways of blending two curves. In my implementation I have chosen blending function shown in equation (7) which is a polynomial function of first order. It is simple to calculate and derivate. My program gives possibility to choose how much of the original curve is used before the programs starts to blend the two together. (Lakså, 2012, Chap 6.2.2)

$$B(t) = 3t^2 - 2t^3 \tag{7}$$

To evaluate a curve at blending point I use $c_3(t)$, defined by formula described in equation (8). It uses x as a blending point so that one can choose how much of original curves is used before blending occurs.

$$c_3(t) = c_1(t) + B\left(\frac{t-x}{1-x}\right)(c_2(t-x) - c_1(t)), \quad x \le t < 1 \text{ and } x \in \langle 0, 1\rangle \,. \tag{8}$$

Total curve is then defined by equation (9).

$$f(t) = \begin{cases} c_1(t), & \text{if } 0 \le t < x, \\ c_3(t), & \text{if } x \le t < 1, \\ c_2(t-x) & \text{if } 1 \le t \le 1+x. \end{cases} \tag{9}$$

Which gives me a C_1 smooth curve in domain $[0, 1+x]$.

2.4 GERBS Curve

General Expo-Rational B-Spline curve or GERBS curve for short is a more advanced type of a B-Spline curve that instead of simple control points uses *control curves* or *local curves*. This gives one much more control over the resulting curves as one can not only change the position of control curve, but also its orientation by flipping, rotating or shrinking it. This control comes at a cost, as it is much more expensive and data demanding to evaluate these curves and they are much less numerically stable than simple B-Splines. Final curve is a result of blending these local curves into one, so one can easily see that they build up on two previous curve types. This blending and need for considering orientation of local curves is what makes it harder to work with GERBS.

Formula for them is defined in equation (10), where $c_i(t)$, $i = 1, ..., n$ are local curves and $B_i(t)$, $i = 1, ..., n$ are the ERBS basis functions

$$f(t) = \sum_{i=1}^{n} c_i(t) B_i(t) \tag{10}$$

One also needs to take into consideration whether a curve is open or closed as the local curves span several knots and each point on the curve is defined by two curves. In case a curve is closed then I need to adjust the first and last knot vector by formulas described in equation (11), and the last subcurve is the same as the first one.

$$\begin{aligned} t_0 &= t_d - (t_{n+d} - t_n) \\ t_{n+k} &= t_{n+d} + (t_k - t_d) \end{aligned} \tag{11}$$

Besides that knot vector is made quite similar to the way that it is made for B-Splines, however it must start and end at values defined by model curve. In my case the model curve I chose starts at 0 and ends at 2π and is known as one of the many heart curves, but it gives the nicest looking heart. It is defined by equation (12) and I found this definition in (Weisstein, 2018).

$$\begin{aligned} x &= 16\sin^3(t) \\ y &= 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t), \quad t \in [0, 2\pi] \end{aligned} \tag{12}$$

After creating a knot vector it's relatively easy to create local curves based on t. I am using *PSubCurve* for it, however a more common subcurve type are the *Bezier curves*. In my case there are 10 subcurves and as such the knot vector contains 13 knots. In order to evaluate the curve I simply blend the two curves that define each point. Using those simple subcurves has made my work much easier and therefore let me focus more on the animation of final curve, than to spend time setting up the Bezier curves.

After my curve is created I apply simple affine transformations to each local curve creating an animation that looks like a beating heart. Transformations used here are translation and rotation.

2.5 GERBS Surface

GERBS surfaces are similar to GERBS curves as they are also effect of blending of local surfaces, however now there are 4 surfaces defining each point on the surface and it requires two knot vectors to define it. It also requires that I check if the surface is closed in opposite direction of the knot vector so that I can adjust the knot vector the same way I did for the GERBS curve.

Instead of a vector of subcurves, a GERBS surface uses a matrix of subsurfaces. creating it is not easy as subsurfaces in last column and row must be created or copied from first column/row depending on whether or not the model surface is closed or not.

In order to show that my code is working properly I have created three GERBS surfaces, one open, one closed in one direction, and one closed in both directions. They are as follows a plane, a cylinder and a torus.

Evaluating GERBS surface is understandably hard, general formula for GERBS surfaces is defined in equation (13)

$$s(u,v) = \sum_{i=0}^{n_1-1} \left(\sum_{j=0}^{n_2-1} c_{i,j}(u,v) b_j(u) \right) b_i(v), \quad b_j(u) = B \circ w_{1,j}(u), b_i(v) = B \circ w_{1,i}(v). \tag{13}$$

Which in turn can be turned into:

$$S(u,v) = \begin{pmatrix} 1-b_j & b \end{pmatrix} \begin{pmatrix} c_{i-1,j-1} & c_{i,j-1} \\ c_{i-1,j} & c_{i,j} \end{pmatrix} \begin{pmatrix} 1-b_i \\ b \end{pmatrix} \tag{14}$$

Thus getting general formulation for position of each point on the surface:

$$S(u,v) = (1-b_i)(1-b_j)c_{i-1,j-1} + (1-b_i)b_j c_{i-1,j} + b_i(1-b_j)c_{i,j-1} + b_i b_j c_{i,j} \tag{15}$$

I also need to calculate derivatives of equation (15), S_u S_v and S_{uv}, and put them all in a matrix to define all the information needed for every point on the surface to get proper shading and lighting on the surface.

$$\begin{pmatrix} pos & S_v \\ S_u & S_{uv} \end{pmatrix} \tag{16}$$

This matrix is calculated and recorded for each point on the surface.

2.6 Implementation

In order to implement the B-Spline curves I have based my class on `PCurve` template class defined in GMlib. I used `DVector` of `Vector` objects for keeping track of control points, `DVector` is a dynamic vector, and as name suggest can be dynamically expanded. A `DMatrix` is used for the A matrix used in calculating control points.

I have also used `PCurve` to represent blending curves, model curve and GERBS curve. Whereas `PSurf` is used for the GERBS Surface. As I have mentioned earlier the I used `PSubCurve` for local curves in my GERBS curve and I used `SimpleSubSurface` for local surfaces in the GERBS surface.

2.7 Tesselation

One topic important to mention is the *Tesselation*. Tesselation is used in rendering of curves and surfaces as a way of sampling and displaying approximated data. It is a piecewise linear approximation of freeform geometry. It means that it samples the curve and displays it on the screen as a set of straight lines and it displays surface as a set of triangles. Thanks to that computer avoids calculating position for every pixel on the surface and instead draws surface as a set of polygons that is much easier to compute.

There are three main types of tesselation, *regular*, *irregular*, and *quasi-regular*. An example of regular tesselation is a plane that is divided into a set of identical triangles ordered in regular rows and columns. *Quad Trees* is an example of quasi-regular tessalation, this is a structure where each node in the tree has at most four children (exactly 4 in internal nodes). It's quasi regular as some parts of the surface can have more subdivision than others, but it's always done in a regular fashion.

Irregular tesselation does not keep any order or fashion of how the surface is divided, however it still follows some rules. Most commonly used irregular tesselation is based on *Delaunay triangulation*, as described by (Berg et al., 1997, Chap 9.2) it is a triangulation for a set of points P where no point in P is inside the circumcircle of any triangle in the triangulation, this is clearly shown in figure 1. It is an optimal triangulation algorithm that maximizes the smallest angle for each triangle in it, as to avoid sliver triangles. It also corresponds to the dual graph of *Voronoi diagram* described in (Berg et al., 1997, Chap 7.1). Resulting triangulation will have $2n - 2 - k$ triangles and $3n - 3 - k$ edges, where k stands for number of points on the boundary.

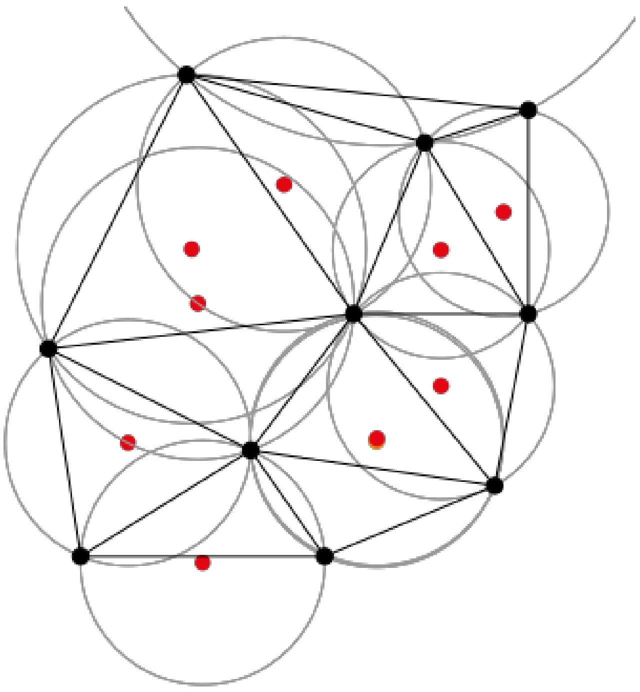

Figure 1. Illustration of the the Delaunay triangulation with all the circumcircles shown (Es, 2016).

My program uses it in the background for displaying my objects on the screen, and algorithm 2 describes how this triangulation is made. It is based on algorithm described in (Berg et al., 1997, Chap 9.3).

Algorithm 2: Delanuay triangulation algorithm

```
**Data:** A set $P$ of $n$ points
**Result:** A Delaunay triangulation of $P$
Find the convex hull for the $P$;
Make 3 new points (the triangle);
Make 3 new edges and 1 triangle;
for Each $p$ in $P$ do
    Find triangle that $p$ is inside of;
    if $p$ is inside the triangle then
        Split the triangle into 3;
        Check and perform swap if necessary;
    else
        Split the edge that $p$ lies on in 2 and each triangle into 2;
        Check and perform swap if necessary;
    end
end
Remove the 3 new points;
Remove the edges;
```

3 References

Berg, M. d., Kreveld, M. v., Overmars, M., and Schwarzkopf, O. (1997). *Computational geometry: algorithms and applications.* Springer-Verlag.

Es, N. (2016). Delaunay circumcircles centers.png. [Online; accessed 12-December-2018].

Lakså, A. (2012). *Blending technics[sic] for Curve and Surface Constructions.*

Weisstein, E. W. (2018). Heart curve. http://mathworld.wolfram.com/HeartCurve.html. Accesed on 2018-12-11.