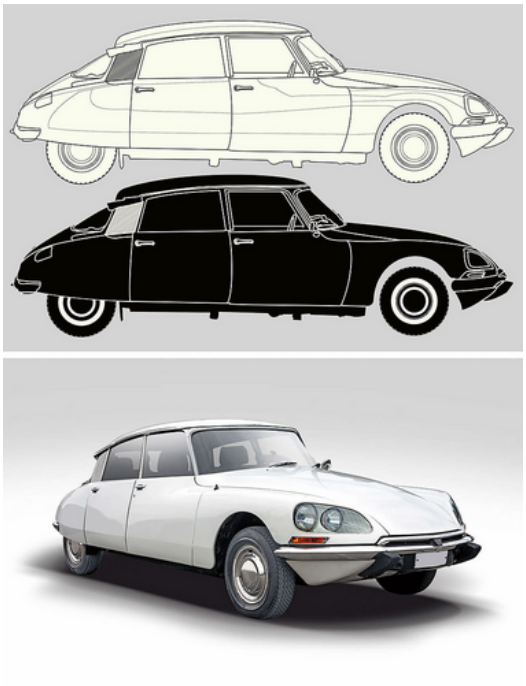
Construction of a 3rd Degree B-Spline and a Tensor Surface



Master in computer science 4th Year
2023-2024

Course: Applied Geometry and visual effects

Supervisor: Arne Lakså

Student: CHANOU Mouhamad

Table des matières

Introduction

We're about to explore how to make shapes on a computer using different types of curves and surfaces. We'll start with simple curves called Bezier curves and then move on to more complex ones called B-Splines. We'll learn how to make these curves using points and a special method called the least square method. We'll also discover a clever way to make curves smoother using a method called the close Lane Riesenvelt algorithm. Exploring blending splines, which are like mixing different colors, will help us understand how to combine curves in interesting ways. Finally, we'll look at blending surfaces, where we put curves together to create beautiful and detailed shapes. Throughout our journey, we'll face challenges that will make us learn and adapt, shaping a project that shows how creative and flexible we can be when designing shape on a computer.

Illustration Table

1. Bezier Curve

In the realm of parametric curves, where mathematical equations guide the movement of a point to craft intricate shapes, Bézier curves stand out as a powerful tool in computer graphics and animation. Defined by a set of control points, Bézier curves enable the creation of smooth and varied shapes by manipulating the positions of these control points. In facts, they are the results of the work of an Engineer working at Citroën in 1959 De Casteljau. With the introducing of computers in engineer design, it was only possible to design with basics shape such as sphere, rectangle but with Bézier curves, new innovative appears in the automobile industries as shown in the following Citroën DS.

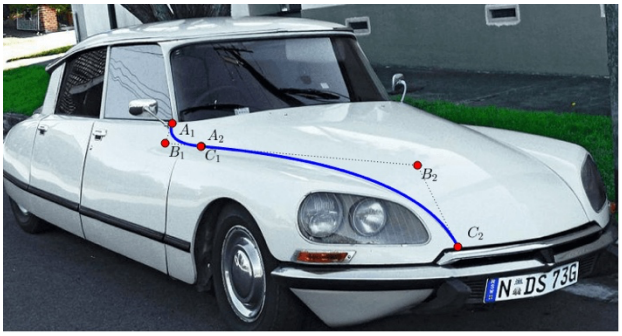


Figure 1 Picture of a Citroën DS

Moving beyond the individual strokes of Bézier curves, we encounter B-Splines, interconnected brushes, working collaboratively to form a more complex curve. In contrast to Bézier curves, B-Splines incorporate knots, acting as influential guidelines that affect the influence of each control point across different curve segments. This additional flexibility allows artists to craft not only simple and smooth curves but also intricate details and variations. As we delve into the technical aspects, we discover that B-Splines, particularly Non-uniform Rational Basis Splines (NURBS), serve as a versatile mathematical model in computer graphics, offering precision for representing both analytic and modeled shapes. The distinguishing factor between B-Splines and Bézier curves lies in the knot vector, a sequence of parameter values that dictates the influence of control points on the NURBS curve. While Bézier curves do not use knot vectors, B-Splines leverage them to refine the control point influence, providing adaptability in shape creation. The forthcoming exploration will delve into the construction, properties, and practical applications of B-Splines, emphasizing their role in computer-aided design (CAD), animation, and other fields requiring precise and adaptable shape manipulation. The journey will culminate in the implementation of a B-Spline with a knot vector, showcasing various construction methods and the precision afforded by control point adjustments and knot vector manipulation.

2. B-Spline of 3rd Degree

2-1. Constructor and Control points

A B-Spline curve is constructed by joining polynomial-based curve pieces at knot values, forming a continuous and smooth curve. Specifically, for a 3rd degree B-Spline curve, each piece within the domain [ti, ti+1), where i ranges from d to n, is defined by the following formula:

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

Here, $w_{d,i}(t)$ represents a function equivalent to the Bernstein algorithm, where the variable t is substituted with a scaled range of values between 0 and 1. This concept of local spline will be further explored in the subsequent discussion on the local spline in our Blending Spline based on a model curve. The Bernstein algorithm is a mathematical method used to compute Bernstein polynomials, which are fundamental in polynomial interpolation and curve construction, particularly in Bézier curves and B-Spline curves. The Bernstein polynomial $(\mathrm{B}_{\mathrm{k,d}}(\mathrm{t})$ is defined as:

$$\mathrm{B}_{\mathrm{k,d}}(\mathrm{t}) = \binom{\mathrm{d}}{\mathrm{k}} \cdot (1 - \mathrm{t})^{\mathrm{d-k}} \cdot \mathrm{t}^{\mathrm{k}}$$

Here, k ranges from 0 to d, and $\binom{\mathrm{d}}{\mathrm{k}}$is the binomial coefficient, representing the number of ways to choose k elements from a set of d elements. The resulting polynomial exhibits properties like non-negativity, symmetry, and the fact that the sum of all $(\mathrm{B}_{\mathrm{k,d}}(\mathrm{t})$ for a fixed d equals 1.
The Bernstein algorithm provides a concise and efficient way to calculate these polynomials, playing a crucial role in the representation of curves in computer graphics and geometric modeling. The use of Bernstein basis functions within the B-Spline formulation contributes to the smooth and adaptable nature of the resulting curve, allowing for precise control over the shape through the manipulation of control points and the knot vector. We scale the knot vector of a B-spline to the interval [0, 1] for several advantages. It simplifies calculations and analysis by providing a standard domain, allows us to reuse existing B-spline tools designed for this range, and keeps the curve definition independent of control point positions. While not strictly necessary, scaling to [0, 1] offers a convenient and efficient starting point for working with B-splines.
We are going to construct in C++ due to the precision with which we can compute all the different shapes that we are going to study. The GMlib library provided in the resources contained a lot of geometric virtual and not classes which will help us to conduct our project.

Here is our class called MySpline with different method which implement all the computation that the representation of the curve.

```
protected:
  // Virtual functions from PCurve, which have to be implemented locally
  //Compute the Parametric equation of the spline
  void            eval(T t, int d,bool /*l*/) const; //override
  //Define the domain of the Spline
  T               getStartP() const override;
  T               getEndP()   const override;
  //Find the interval of the following value of t
  int             findIndex(T t) const;//T is the type selected and is the value for whi
  //Scale the knot vetor between 0 and 1
  T               makeW(int d, int i, T t) const;
  //Generate the knot vector with the number of control points
  void            makeKnotVector(int m) ;
  //Compute the products of the differents coefficient of the parametric equation
  GMlib::Vector<T,4>              makeB(int i, T t) const;
```

Figure 2: Desciprion of MySpline methods

There you can find the graphical representation of a 3rd degree B-Spline with the following controls points.

C[0]={5.0f, 2.0f, 8.0f}
C[1]={2.0f, 9.0f, 2.0f}
C[2]={2.0f, 3.0f, 4.0f}
C[3]={3.0f, 4.0f, 5.0f}
C[4]={3.0f, 6.0f, 0.0f}
C[5]={1.0f, 4.0f, 0.0f}
C[6]={5.0f, 8.0f, 3.0f}

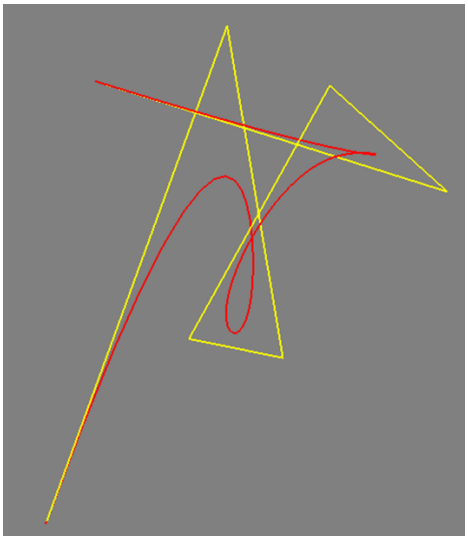


Figure 3 : B-Spline with 7 control points.

2-2. Constructor with the least square method

The least squares method is a precise mathematical technique commonly employed for curve fitting or interpolation (mathematical technique used to estimate values that fall between known, given data points), particularly when the number of interpolation points exceeds the degree of the polynomial being used. In situations where traditional interpolation methods may result in an overdetermined system of equations, the least squares approach minimizes the sum of the squared differences between the observed and calculated values, effectively finding the best-fitting curve.

In our case we will try to create a least square constructor for interpolating 20 points at different position of a circle and we will see how accurate the interpolation is of the actual shape.

With this method, the knot vector is independent of the values of the x_i of the splines. We will make a knot vector such that there is an x_i value inside every B-splines.

Given 20 strictly increasing real numbers x_i, and 20 points p_i. We can construct a B-spline curve of degree 3.

$$c(t) = T^d(t)c,$$
$$Ac = p$$

Here A is a 20(m) x7 (n) where m is always inferior to n implying that A can never be squared as a matrix. To solve the previous equation, we will use instead $min|Ac - p|^2$ and find the derivative with respect to c and find when the derivative is 0.

That is when $\frac{d}{dc}(|A|^2 - 2pAc + |p|^2)$=0 which gives 2 A^TAc-2A^Tp=0.

Thus, to solve the least square expression we must solve Bc = y, where B=A^TA and y = A^T*p while c is the parametric equation of our curve. That also means c=y*B^{-1} and this is what we are trying to implement.

In C++, we will have the following A matrix

```
GMlib::DMatrix<T> A(m, n, T(0));
```

and with this one, we will get all the right matrixes and get the following values for C.

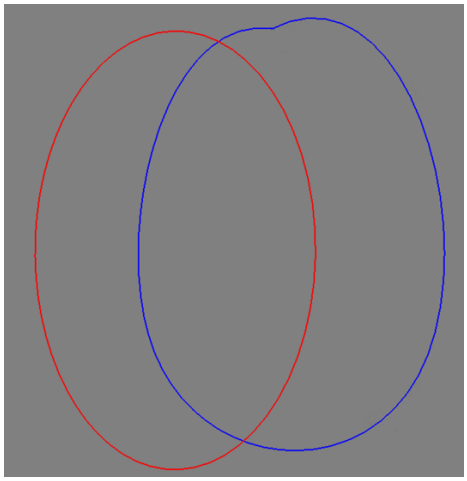


Figure 4: Interpolation of a 20 points circle with 6 control points.

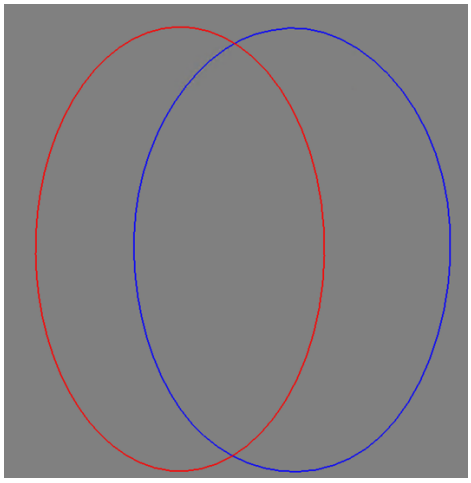


Figure 5: Interpolation of a 20 points circle with 8 control points.

We can notice from the previous figures that the more control point we have the more our interpolation is accurate.

3. Subdivision using close Lane Riesenvelt

Uniform splines based on integer-valued Bézier-splines, introduced by Jacob Schonberg in 1940-1950, offer unique advantages due to their symmetrical structure and identical nature of basic functions merely shifted relative to one another. This property facilitates a streamlined process for knot insertion, the addition of intermediate knots within existing ones, and subsequently allows for reparameterization to maintain integer knots, leading to the development of subdivision curves.

Subdivision techniques generally involve two primary approaches: corner cutting and interpolation. Corner cutting replaces the initial point set with a larger set of points lying along the lines connecting the old points, effectively refining the curve while maintaining its shape. Interpolative subdivision, however, adds new points to the original point set without confining them to lie strictly within the convex hull of those points.

The Lane-Riesenfeld subdivision scheme is an interpolatory subdivision algorithm for uniform B-splines. It was introduced in 1980. This algorithm is particularly notable because it produces closed uniform B-spline curves and open curves that are not clamped. The Lane-Riesenfeld subdivision algorithm involves two steps: doubling the point set and smoothing/degree raising. Doubling is achieved either by adding new points halfway between the existing points or duplicating every point twice. Smoothing consists of calculating new points as the average of adjacent old points, which is repeated until reaching the desired degree of details.

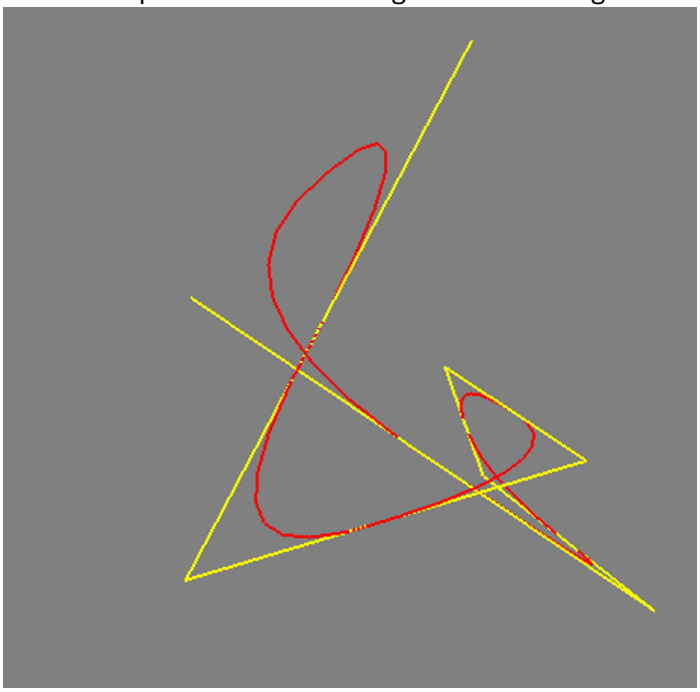


Figure 6: Closed Lane Riesenfeld Subdivision spline.

4. Blending Spline

A blending spline is a construction method where local geometries are smoothly combined using a blending function to create a unified global geometry. This technique allows for the seamless integration of different local shapes or curves into a cohesive and continuous overall structure. Blending splines offer flexibility and favorable smoothness properties, particularly at the endpoints, making them ideal for creating transition curves that maintain continuity and aesthetic appeal in geometric designs.

4-1 Model Curve

To implement the blending spline, we will implement a model curve with its parametric equation. We will just choose a model curve with the following parametric equation.

$$x(t) = \cos(t) + 0.5\cos(3t)$$
$$y(t) = \sin(t) + 0.5\sin(3t - 1)$$
$$z(t) = 0.3\sin(5t) + \sin(t)$$

And we will have the following curve.

Figure 7: Model curve.

4-2 B-Splines with B-function

Now we will implement our own versions of a versions curve with a Beta function and local curves. Beta functions, in the context of B-splines, are a type of basis function used to construct B-spline curves. They are defined by a recurrence relation and are crucial for creating complex shapes and surfaces using a set of control points. Beta functions are part of a broader class of B-spline basis functions, which include cardinal B-splines, rational B-splines, and expo-rational B-splines. These functions are used extensively in shape optimization methods and are fundamental tools for shape manipulation and design optimization. Among all the existing B-

functions, here we can take a look at 3 of them of the first order which are: the linear function B(t)=t, the trigonometric function $B(t) = sin^2 \frac{\pi t}{2}$ and the polynomial function $B(t) = -2t^3 + 3t^2$. We are going to use a polynomial beta function of the first because it is an easy function to implement, and its derivative still provides us a lot of information.

Another important detail of the implementation of our present Blending curve is the usage of local curves which are useful in shape manipulation for example we could be able to apply an affine transformation to local curves of our main curve to simulate shape deformation. To have local curves at the emplacement of each of our control points in the GMlib library we will use the class PSubCurves. In that way we will be able to apply specific shape deformation by changing the number of control points, the affine transformation and many more.

Here is an example of combination of translation and rotation applied to the local curves. For all curves with an even index, we will apply a translation by the vector.
$vec = (0.025 \cdot \sin(t), 0.025 \cdot \cos(t), 0)$ multiplied by 0.025, followed by a rotation around the x-axis by an angle of $2 \cdot dt$

This combination of transformations effectively translates the curves along a sinusoidal path in the x-y plane and rotates them around the x-axis.

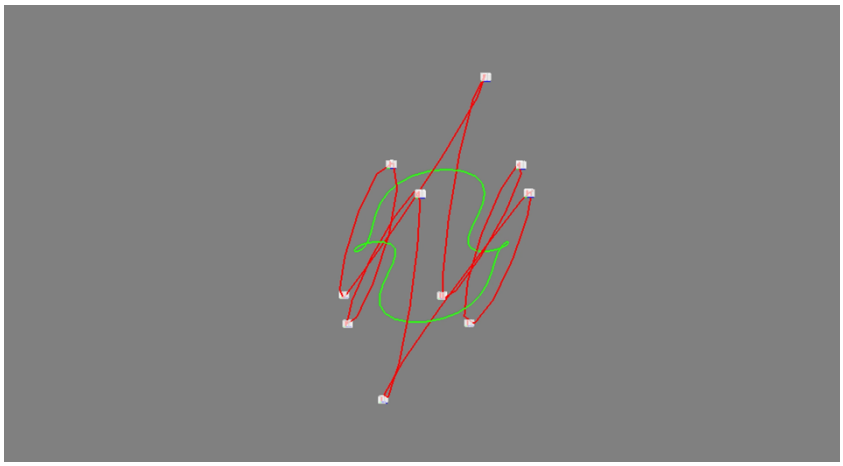


Figure 8: Picture of the Shape deformation of my model curve.

5. Blending Surfaces

Among all the surfaces, we are going to study tensor product surfaces. The name tensor product refers to a product of two vector spaces. It is the product between two vector u and v.
In practical terms, a tensor product combines two vectors of basic functions to form a matrix of basis functions. Each basis function is associated with coefficients to create a parametric surface. This surface is typically embedded in R^3, where the coefficients represent points or vectors. The general formula for a tensor product surface based on

polynomials involves summations over polynomial degrees and parameter domains in the u and v directions.
Essentially, constructing a tensor product surface involves blending curves to form a coherent surface. This approach allows for the application of curve algorithms to tensor product surfaces, showcasing the versatility and interconnected nature of these mathematical constructs in creating complex surfaces from simpler curve elements. This is exactly what we are going to do for the implementation.
We are going to repeat the step as in "MySpline" (**Figure3**) for the u and for the v directions. We will have a u knot vector, a v knot vector and for all the step, we will do the same computation in these two directions and according to the kind of shape we have will need to change the domain and the number of patches in the knot vector as display in the following table.

u - direction	v - direction	The parameter domain	Number of patches
Open	Open	$(u,v) \in [u_1, u_{n_u}] \times [v_1, v_{n_v}]$	$(n_u - 1) \times (n_v - 1)$
Open	Closed	$(u,v) \in [u_1, u_{n_u}] \times [v_1, v_{n_v+1})$	$(n_u - 1) \times n_v$
Closed	Open	$(u,v) \in [u_1, u_{n_u+1}) \times [v_1, v_{n_v}]$	$n_u \times (n_v - 1)$
Closed	Closed	$(u,v) \in [u_1, u_{n_u+1}) \times [v_1, v_{n_v+1})$	$n_u \times n_v$

In our code, the surface point (x, y, z) is expressed in terms of the blending functions and their derivatives, as well as the local surfaces:

- Blending Functions:

```
GMlib::Vector<T,2> bu = makeBlendSurf(i,u, _u);
GMlib::Vector<T,2> bv = makeBlendSurf(j,v, _v);

GMlib::Vector<T,2> dBu = makeBlendSurfD(i,u, _u);
GMlib::Vector<T,2> dBv = makeBlendSurfD(j,v, _v);
```

- Local Surface Evaluations:

```
GMlib::DMatrix<GMlib::Vector<T,3>> S1 = this->localSurf(i-1)(j-1)->evaluateParent(u, v,d1, d2);
GMlib::DMatrix<GMlib::Vector<T,3>> S2 = this->localSurf(i)(j-1)->evaluateParent(u, v,d1, d2);
GMlib::DMatrix<GMlib::Vector<T,3>> S3 = this->localSurf(i-1)(j)->evaluateParent(u, v,d1, d2);
GMlib::DMatrix<GMlib::Vector<T,3>> S4 = this->localSurf(i)(j)->evaluateParent(u, v,d1, d2);
```

3. Surface Point and Derivatives:

```
//Surface component on x
this->_p[0][0] = bv[0]*(bu[0]* S1[0][0] + bu[1]* S2[0][0]) + bv[1] *( bu[0]* S3[0][0] + bu[1]* S4[0][0]);
//Surface component on y
this->_p[0][1] = bv[0] * (dBu[0] * S1[0][0] + bu[0] *S1[0][1] + dBu[1] * S2[0][0] +bu[1] *S2[0][1]) +
                 bv[0] * (dBu[0] * S3[0][0] + bu[0] *S3[0][1] + dBu[1] * S4[0][0] +bu[1] *S4[0][1]) ;
//Surface component on z
this->_p[1][0] = dBv[0] * (bu[0]* S1[0][0] + bu[1]* S2[0][0]) + bv[0] * (bu[0] * S1[1][0] + bv[1]* S2[1][0]) +
                 dBv[1] * (bu[0]* S3[0][0] + bu[1]* S4[0][0]) + bv[0] * (bu[0] * S3[1][0] + bv[1]* S4[1][0]) ;
```

Here, bu, dBu, bv and dBv represent the components of the blending functions and their derivatives, and (S_{ij}) represents the components of the local surfaces. The indices i and j are determined based on the parameter values u and v and the knot vectors _u and _v. These equations collectively describe how the blending surface point x, y, z, and its partial derivatives are computed at a given parameter point (u, v).
We can then show the blending surface for a plane (open, open) a cylinder (open, closed) torus (closed, closed).

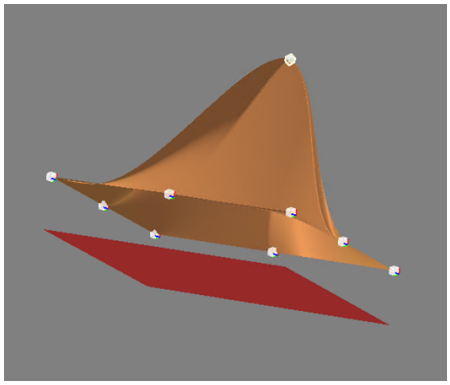


Figure 8: Simulation of a plane.

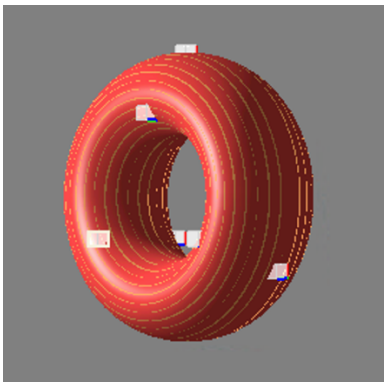


Figure 9: Simulation of a Torus.

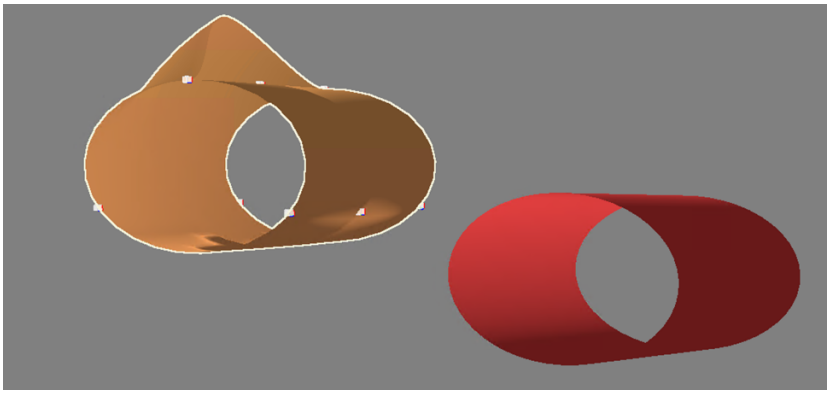


Figure 10: Simulation of a Cylinder.

6. Difficulties faced.

This project was genuinely interesting, but I encountered some difficulties, especially regarding the programming environment. I use a Mac operating system, and most of the software used for this implementation doesn't exist in the same version for Mac. This could have posed a problem for every action I performed, trying to find the equivalent on Mac. Consequently, I decided to rent a Windows virtual private server with a graphical card monthly.

In addition to that, it was initially challenging for me to code the first formula, especially the parametric equations. However, after doing it for a spline, everything started looking clearer.

Conclusion

In conclusion, this project has been an exciting adventure into the world visual effects and computation graphics. From simple Bezier curves to more complex B-Splines, each step taught us something new about making shapes look better. Mixing different curves with blending splines added an artistic touch to our work, showing how we can create detailed designs. Despite some difficulties, especially in using different computer programs, this project demonstrates our ability to keep trying and solve problems. Through our exploration of math in design, we not only learned more about drawing shapes but also discovered endless possibilities in computer graphics and animation. This project celebrates our creativity, determination, and the amazing things we can create by combining math with art in computer design and it is more than surprising to see how mathematics can be applied in film making and 3D visual effects.

Bibliography and project resources

- Arne Lakså. Blending Techniques in Curve and Surface Construction. GEOFO 2023
- The code of the project can be found in the following link: https://github.com/mchanou/projectVisualEffect