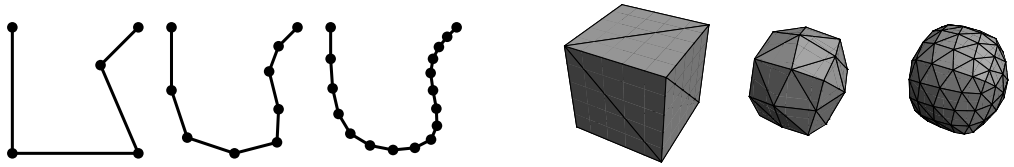
SPLINES AND SUBDIVSION

MODELING AND ANIMATION, LAB 3

Mark Eric Dieckmann Erik Junholm Emma Broman Robin Skånberg

Wednesday 22nd March, 2023



Abstract

This lab covers the topic of smooth surfaces and curves in computer graphics. We look at subdivision curves and surfaces as an approximation thereof. We consider stability issues in discrete computations and how the construction of curves and surfaces by convex combinations of Bézier curves and B-splines leads to stable schemes. We look at the *uniform cubic B-spline*, which is the most commonly applied spline in computer graphics. Uniform B-splines can be constructed from repeated box convolutions which we also investigate in this lab. We investigate how spline theory can be used to design subdivision schemes that create smooth curves and even surfaces via the Loop subdivision scheme.

1 Introduction

Subdivision curves and surfaces are widely used in the area of applied computer graphics. They are fast to compute, because we can keep the order of the splines relatively low. They are easy to interact with, because the impact of operations is local. And the numerical scheme is stable because all calculations are based on stable bases.

The concept of curve- and surface subdivision is well known since a long time. This is true in particular for subdivision curves, which are based on spline theory that was first mentioned in the 40s of the last century. This text covers only some of the basic concepts in the theory. Especially it is based on the assumption of uniform basis functions; this enables more straightforward analysis and construction but limits some of the geometric properties. For a more in-depth description of subdivision see [Zorin and Schröder, 2000], whereas spline theory is thoroughly covered in [Lyche and Mörken, 2007].

1.1 Parametric polynomial curves

In this lab, we will work with *parametric representations*. This type of description maps a real number to a vector output

$$f : \mathbb{R} \to \mathbb{R}^n$$

A parametric curve may be written $\boldsymbol{p}(t) = \{x(t), y(t)\}$ in 2D. We see that the parametric representation is made of vectors of functions, and we

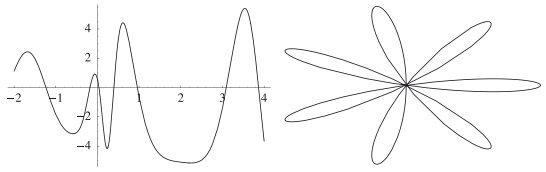


Figure 1: Left: a function and right: a parametric curve.

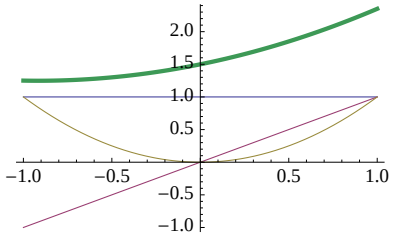


Figure 2: The three first monomial basis functions $(1, x, x^2)$ with coefficients $(1.5, .55, .3)$ give the following function $1.5 + .55x + .3x^2$ (thick).

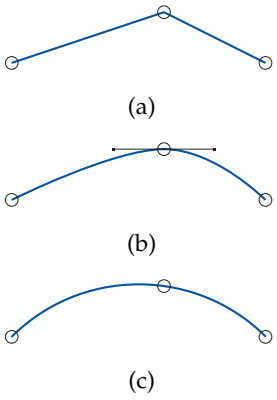


Figure 3: Piecewise curves and their continuity at a joint. (a) Linear curve segments have C^0 continuity at the joint. (b) Second order curves with the same first derivatives at the joint. (c) Cubic curves with the same first and second derivatives at the joint.

will refer to parametric curves simply as curves, see figure 1 for an example of the difference to functions.

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{c}_i t^i = \left\{ \begin{array}{l} x(t) = \sum_{i=0}^{n} a_i t^i \\ y(t) = \sum_{i=0}^{n} b_i t^i \end{array} \right\} \qquad (1)$$

Giving coefficients in the vector form $\boldsymbol{c}_i = (a_i, b_i)$ used above leads to a more compact description of the parametric curves. The curve is infinitely differentiable everywhere[1], and easy to compute. The curve is formed by coefficients $(\boldsymbol{c}_i)$ and polynomial *basis functions* on the form x^i, as seen in equation (1). This type of basis function is referred to as a *power basis* or *monomial basis*. An example of a function in monomial basis is shown in figure 2.

1.2 Piecewise curves and continuity

A common way of making complex shapes without too much computational effort is to stick together low-order curves and patches. The advantage is that the order of the basis functions remains low and the curve therefore becomes fast to compute, and in addition has less oscillations. The disadvantage is that we must explicitly connect the different curve parts and make the joints look sufficiently smooth. This smoothness is measured in terms of continuity, and labeled C^n where n is the number of continuous derivatives at the joint.

C^0 continuity is the lowest possible order for a connected curve. Adjacent curve segments share the same endpoints. See figure 3 (a).

C^1 continuity is very common in computer graphics. Adjacent curve segments share the same endpoints and also share the same first derivative. See figure 3 (b).

C^2 continuity is often needed for motion since the human eye is very sensitive to non-smooth movement. Adjacent curve segments share the same endpoints and first and second derivatives. See figure 3 (c).

The term geometric continuity (G^n) relates to the fact that higher dimensional curves have multiple partial derivatives, or tangents at the joint. If the

[1] Although the derivative may be zero.

first n derivatives are proportionally equal (the respective partial tangents are parallel but may have different lengths) the curve is G^n continuous.

As a side note, the expression C^{-1} continuity is sometimes used for piecewise curves with end points not matching.

1.3 Stability and convex calculations

Limited precision is a common problem in discrete numerical computations. For example the number 1.125 can be represented without round-off error as a binary number since both 1 and .125 are exact fractions of powers of two ($\frac{1}{2^0}$, $\frac{1}{2^3}$). On the other hand, the common number 0.1 is not an exact fraction of a power of two and therefore can only be *approximated*. It falls between the two closest representable numbers which happen to be 0.0999999046 and 0.1000000238 in single precision. Round-off errors also lead to logical pitfalls such as comparing floating point numbers[2]. For example,

```
69.82 == (69.2 + .62)
```

and

```
a+b+c == c+b+a
```

will both evaluate to false in some cases, depending on the actual numbers. *Subtracting large numbers* can result in severe errors, because the error then creeps up into the more significant digits

```
1e7-(1e7+.00001) -> -1.0000541806221008e-05
```

Summing up floating point numbers in a loop also accumulates errors. Round-off errors are discussed by [Goldberg, 1991; Hecker, 1996]. We can usually not avoid floating point calculations but we can limit the damage. Let us start with a definition:

Definition 1 *A stable calculation in numerical computing is defined as a calculation for which small perturbations in the input result in small perturbations in the output.*

[2]The `map` used to avoid redundant vertices in the HalfEdgeMesh uses floating point comparison through < (operator less).

If we consider calculations formed by a multiplication of some coefficients times a basis, like in equation (1) then it follows that our calculations will be stable if our base is stable.

As you can see in figure 4 the monomial basis (a-c) is not stable. It is somewhat analogous to a ray, with an origin and a direction (a). The direction vector is (in this case) calculated as the slope between $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$, and if there is an error in the positions of $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ (b) the error will propagate to the slope $\tilde{\boldsymbol{k}}$. The total error of the function will then grow with the distance from $\boldsymbol{c}_1$ as illustrated in figure 4 (c). This leads us to the conclusion that the monomial basis is unstable since small errors in the coefficients can lead to large errors in the result.

1.3.1 Convex combinations

A convex combination is a linear combination of data points (which can be vectors, scalars, or more generally points in an affine space) where all coefficients are non-negative and sum up to 1. It is called "convex combination", since all possible convex combinations (given the base vectors) will be within the convex hull of the given data-points. In fact, the set of all convex combinations constitutes the convex hull, see figure 5. Mathematically we can compactly write the convex combination criteria as:

$$f = \lambda_1 c_1 + \lambda_2 c_2 + \cdots + \lambda_n c_n, \quad \text{with } \begin{cases} \sum_i \lambda_i = 1 \\ 0 \leq \lambda_i \leq 1 \end{cases} \tag{2}$$

A special case is with only two data points, where the value of the new point (formed by the convex combination) will lie on a straight line segment between the two points as in figure 4 (d).

As you can see in figure 4 and 5 a convex combination has the desirable property of definition 1, namely that small errors in the input can only produce small errors in the output.

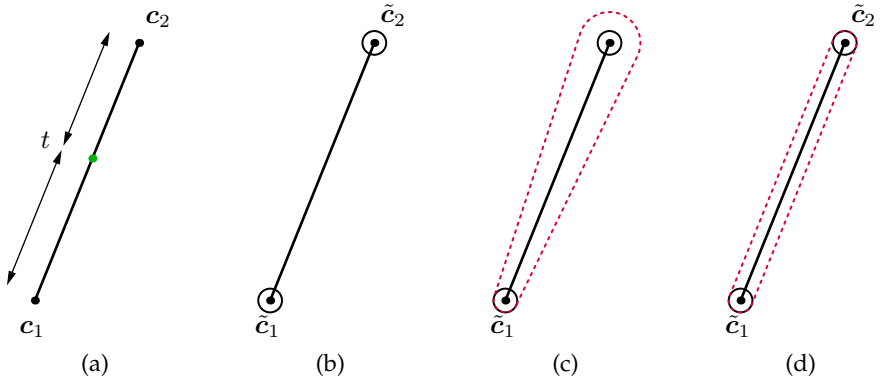


Figure 4: Visualization of the possible error (red) of perturbed coefficients (circles) for a curve $\boldsymbol{p}(t)$. In (a) the curve with true coefficients. In (b) the coefficients are perturbed. The bounding of the error for a curve on (c) monomial form $\boldsymbol{p}(t) = \tilde{\boldsymbol{c}}_1 + \boldsymbol{k}t$, and (d) as a convex combination, $\boldsymbol{p}(t) = (1-t)\tilde{\boldsymbol{c}}_1 + t\tilde{\boldsymbol{c}}_2$.

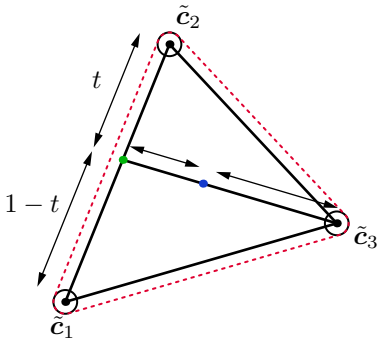


Figure 5: The convex hull of three data points (triangle) and the convex hull of the perturbed points (red).

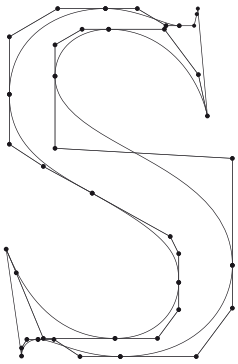


Figure 6: The letter 's' in the Postscript font Times Roman.

2 Curves and basis functions

2.1 Bézier curves

The Bézier curve was developed in 1959 by Paul de Casteljau using the algorithm named after him, which provides a numerically stable method to evaluate these parametric curves. The French engineer Pierre Bézier started to design with it car bodies in 1962. The Postscript language and Adobe Illustrator use cubic Bézier curves, see figure 6, whereas TrueType fonts use quadratic Bézier curves, see figure 7.

Given points $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$, a linear Bézier curve is just a straight line between those two points. The curve is given by

$$\boldsymbol{p}_1(t) = (1-t)\boldsymbol{c}_0 + t\boldsymbol{c}_1 \ , t \in [0,1] \tag{3}$$

and is equivalent to linear interpolation, figure 4. It can be seen as a sum of coefficients ($\boldsymbol{c}_0$ and $\boldsymbol{c}_1$) and the basis functions $\{(1-t), t\}$. Although the linear Bézier curve passes through all its points (it is *interpolating* the points) this is not generally the

case, see figure 6 and 7. A quadratic Bézier curve is the path traced by the function $\boldsymbol{p}(t)$, given points $\boldsymbol{c}_0$, $\boldsymbol{c}_1$, and $\boldsymbol{c}_2$. It is a convex combination of two linear Bézier curves, one from $\boldsymbol{c}_0$ to $\boldsymbol{c}_1$ called $\boldsymbol{p}_{0,1}(t)$, and one from $\boldsymbol{c}_1$ to $\boldsymbol{c}_2$ called $\boldsymbol{p}_{1,1}(t)$.

$$\begin{aligned}\boldsymbol{p}_2(t) &= (1-t)\boldsymbol{p}_{0,1}(t) + t\boldsymbol{p}_{1,1}(t)\ , t \in [0,1] \\ &= (1-t)^2\boldsymbol{c}_0 + 2t(1-t)\boldsymbol{c}_1 + t^2\boldsymbol{c}_2\ , t \in [0,1]\end{aligned} \tag{4}$$

The process is visualized in figure 7 and we see that the curve passes through the endpoints but it misses the middle point. The quadratic Bézier curve is thus only *approximating* the points, which is the general case. From our knowledge about convex combinations we know that the curve will be confined to the convex hull spanned by the points $\boldsymbol{c}_i$.

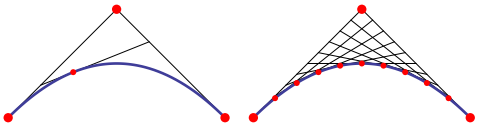


Figure 7: Construction of a quadratic Bezier curve.

2.2 Basis functions

We have introduced the topic of basis functions without really describing what this entails. In this section, the groundwork for further sections will be laid out by inspection of linear and quadratic basis functions. We begin by looking at the simplest case, linear interpolation. The basis functions for linear interpolation are as mentioned above

$$\{t,\ 1-t\}\,, \quad t \in [0,1] \tag{5}$$

We draw these functions in figure 8 and it is clear that for each interval $(0,1)$ or more generally $(i, i+1)$ the support is *local* meaning that only a limited number of basis functions are non-zero, e.g. only two basis functions give support at any t using the basis in figure 8. To construct the value of a function - given a set of coefficients $\boldsymbol{c}$ and basis functions $b_i(t)$

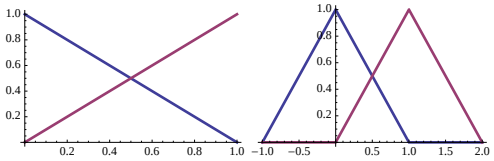


Figure 8: The two pieces of the linear basis functions on $[0,1]$ (left) and the two repeated copies of the general basis function (right).

- we calculate the sum

$$\boldsymbol{p}(t) = \sum_i \boldsymbol{c}_i b_i(t) \tag{6}$$

This calculation is *supported* by the basis functions meaning that if the basis functions become zero the resulting contribution to the sum vanishes.

In the general linear case we have one basis function for each coefficient, in this case the $b_i(t)$ are translated versions of the cardinal linear basis function, also called hat function. We insert a 1-subscript to explicitly denote the degree.

$$b_{i,1}(t) = \begin{cases} t-i, & i \le t < i+1 \\ -t+i+2, & i+1 \le t < i+2 \\ 0, & otherwise \end{cases} \tag{7}$$

Consider instead the cubic basis function on the interval $[-2,2]$ given by

$$B_{i=0,3}(t) = \frac{1}{6}\begin{cases} (t+2)^3, & -2 \le t < -1 \\ -3(t+1)^3 + 3(t+1)^2 + 3(t+1) + 1, & -1 \le t < 0 \\ 3t^3 - 6t^2 + 4, & 0 \le t < 1 \\ -(t-1)^3 + 3(t-1)^2 - 3(t-1) + 1, & 1 \le t \le 2 \\ 0, & otherwise \end{cases} \tag{8}$$

It is depicted in figure 10. Let us see if this basis function fulfills the convex basis criteria in equation (2). By inspection of figure 10 we have that for the interval, all parts are positive and less than 1. If we add all the pieces together we see that they sum up to 1. So this basis can be considered stable. Good! But how can we use it?

Given a set of points, or more generally coefficients $\boldsymbol{c}_i$, associate one basis function $b_{i,3}(t)$ with

each of the coefficients. Then it is possible to compute the value using equation (6). If the basis in equation (7) that we saw earlier is used we get a linear interpolation. This idea was used in the linear Bézier curve. If we instead apply the cubic basis in equation (8) the basis functions are overlapping (more than one basis is non-zero for every t), the basis functions are moreover smoothly shifting in and out of the support. This results in a smooth blend between the coefficients as seen in figure 9.

Another property can be seen in figure 9. Two Bézier segments are joined together at $x = 1$, but the curve is only C^0 continuous. This is an effect of that the basis functions are not continuous, and therefore the curve can only be smooth if the coefficients are "aligned" and produce equal derivatives. On the right-hand side of the figure, a cubic basis is used that is smooth by construction. The resulting curve is smooth independently of the coefficients.

3 Uniform cubic B-splines

The uniform cubic B-spline is the most common spline. It is a cubic polynomial and allows for C^2 continuity over the joints. The B-spline is represented by repeated translated copies of a cardinal basis function as seen figure 10. Local control can be achieved by manipulation of the coefficients, sometimes called control points. A property of the B-spline is that it is approximating the control points.

B-spline basis functions

The cardinal cubic B-Spline basis function was introduced without a name in equation (8). This B-spline fulfills the convex combination requirements of equation (2). It thus forms a stable basis.

A cubic uniform spline curve can be evaluated similarly to the Bézier curve as the sum of the products formed by the coefficients and the basis functions as seen in figure 9 right.

3.1 B-splines as convolutions

To show some of the properties of B-splines we will use a construction by convolution. A convolution is a mathematical operator which takes two functions f and g and produces a third function that in a sense represents the amount of overlap between f and a reversed and translated version of g. It can be found as the integral of the product of the two functions after one is reversed and shifted. As such, it is a particular kind of integral transform:

$$(f \otimes g)(t) = \int f(s)g(t-s)\,ds \tag{9}$$

For example, the B-spline of degree 1 (B_1) is defined as the convolution of $B_0(t)$ with itself

$$B_1(t) = (B_0 \otimes B_0)(t) = \int B_0(s)B_0(t-s)\,ds \tag{10}$$

Where B_0 is defined as

$$B_{i=0,0}(t) = \begin{cases} 1, & 0 \leq t < 1 \\ 0, & otherwise \end{cases} \tag{11}$$

and seen in figure 11. The value of the convolution is also visualized in figure 11. For a given position of the moving box, the value is given by the area under the product of the boxes. This is just the length of the interval where both boxes are non-zero. At first the two boxes do not have common support. Once the moving box reaches 0, there is a growing overlap between the supports of the graphs. The value of the convolution grows with t until $t = 1$. Then the overlap starts decreasing, and the value of the convolution decreases down to zero at $t = 2$. The linear B-spline is the same as the hat function in figure 8. A second order B-spline is found by applying the convolution one more time

$$B_2(t) = \int B_1(s)B_0(t-s)\,ds \tag{12}$$

and in general, the B-spline of degree d can be found by recursive convolution application

$$B_d(t) = \int B_{d-1}(s)B_0(t-s)\,ds \tag{13}$$

From the convolution properties we get some direct information:

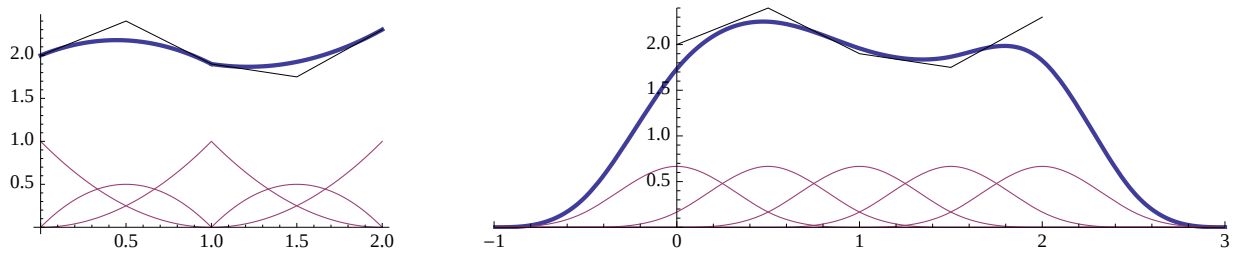


Figure 9: The quadratic bezier basis has non-smooth transitions in the basis functions whereas the degree 3 B-splines are smooth.

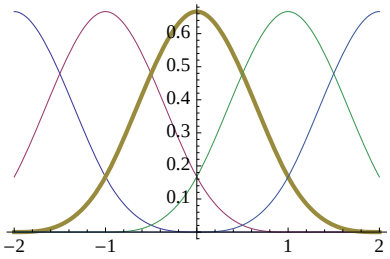


Figure 10: A cubic B-spline basis function (bold) on the interval $[-2, 2]$ together with translated copies of itself.

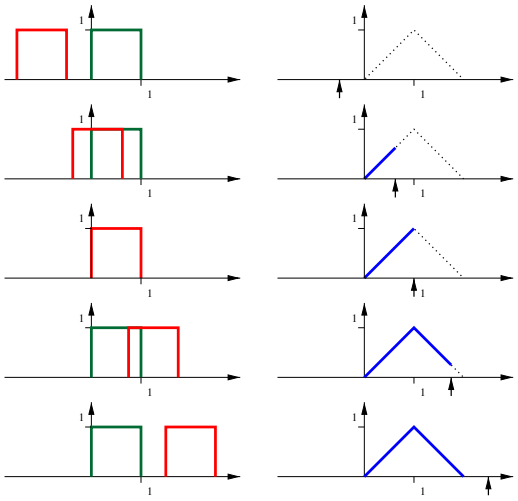


Figure 11: Box convolution.

Theorem 1 *If $f(t)$ is C^k-continuous, then $(f \otimes B_0)(t)$ is C^{k+1}-continuous*

And we see that a spline of degree d is C^{d-1}-continuous since B_0 is C^{-1}.

Defining B-splines as convolutions is an elegant classification. Another important property that can be deduced is that splines are *refineable*. This provides the connection to subdivision methods. The refinement equation is

$$B_d(t) = \frac{1}{2^d} \sum_{i=0}^{d+1} \binom{d+1}{i} B_d(2t - i) \tag{14}$$

where the binomial is

$$\binom{k}{m} = \frac{k!}{m!(k-m)!} \tag{15}$$

Equation (14) states that the B-spline of degree l can be written as a linear combination of translated (i) and dilated $(2t)$ copies of itself. For the complete proof see [Zorin and Schröder, 2000], for an intuitive verification look at figure 12 which shows the dilated linear and cubic splines. In the figure you see that the support of the new refined (colored) B-splines is the same as for the original (gray), and non-negative; also the sum of the parts seem to be equal to the original. If we look at the resulting refined splines

$$B_1(t) = \frac{1}{2}\Big(1B_1(2t) + 2B_1(2t-1) + 1B_1(2t-2)\Big) \tag{16}$$

we can verify that this is indeed true. The corre-

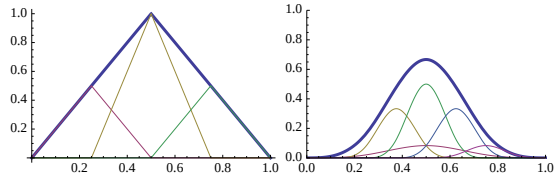


Figure 12: Refinement of a linear (left) and cubic (right) B-spline. Grey is the original (not dilated) spline, and the colored versions the newly dilated ones.

sponding refinement for a cubic B-spline is

$$\begin{aligned} B_3(t) = \frac{1}{8}\Big(& 1B_3(2t) \\ & + 4B_3(2t-1) \\ & + 6B_3(2t-2) \\ & + 4B_3(2t-3) \\ & + 1B_3(2t-4) \\ & \Big) \end{aligned} \tag{17}$$

3.2 Subdividing a spline curve

We showed how to construct a new basis that describes the same support. The new basis is found by dilation and translation. Since the basis changed the coefficients also change. Finding the new coefficients is the key to subdivision. We start by looking at the parametric curve

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{c}_i t^i$$

in monomial basis. For a spline curve this becomes

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} B_i(t)\boldsymbol{c}_i \tag{18}$$

The vector coefficients for the spline curve are

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{c}_0 \\ \boldsymbol{c}_1 \\ \vdots \\ \boldsymbol{c}_n \end{pmatrix}$$

We also write the basis functions as a vector

$$[\ B_n(t) \quad B_n(t-1) \quad B_n(t-2) \quad \dots \quad B_n(t-n)\ ] \tag{19}$$

remembering the translation property ($B_{i,d}(t) = B_d(t-i)$). This makes it possible to write the spline curve on matrix form as

$$\boldsymbol{p}(t) = \boldsymbol{B}(t)\boldsymbol{C} \tag{20}$$

We have also described how to dilate the basis functions and can now express the support using our new basis

$$\boldsymbol{B}(2t) = \quad [\ B_n(2t) \quad B_n(2t-1) \quad B_n(2t-2) \quad \dots\ ] \tag{21}$$

Putting the refinement coefficients in a matrix $\boldsymbol{S}$ we can write

$$\boldsymbol{B}(t) = \boldsymbol{B}(2t)\boldsymbol{S} \tag{22}$$

and also

$$\boldsymbol{p}(t) = \boldsymbol{B}(t)\boldsymbol{C} = \boldsymbol{B}(2t)\boldsymbol{S}\boldsymbol{C} \tag{23}$$

Our geometric curve is now expressed in terms of B-splines whose support is half as wide and which are spaced twice as dense. For this relation we also found the new coefficients to go with the new basis to be $\boldsymbol{S}\boldsymbol{C}$. This process can be repeated to progressively refine the curve representation

$$\begin{array}{rcccc} \boldsymbol{p}(t) & = & \boldsymbol{B}(t)\boldsymbol{C} & = & \boldsymbol{B}(t)\boldsymbol{C}_0 \\ & \rightarrow & \cancel{\boldsymbol{B}(2t)}\boldsymbol{C}_1 & = & \cancel{\boldsymbol{B}(2t)}\boldsymbol{S}\boldsymbol{C}_0 \\ & & \boldsymbol{C}_1 & = & \boldsymbol{S}\boldsymbol{C}_0 \end{array}$$

$$\Rightarrow\ \boldsymbol{C}_{j+1} = \boldsymbol{S}\boldsymbol{C}_j$$

The subscript on the coefficients shows the number of refinements. From this we see how to calculate an arbitrary refined coefficient vector:

$$\begin{cases} \boldsymbol{C}_{i+1} = \boldsymbol{S}\boldsymbol{C}_i \\ \boldsymbol{C}_{i+1} = \boldsymbol{S}^i\boldsymbol{C}_0 \end{cases} \tag{24}$$

The entries of $\boldsymbol{S}$ are given by equation (14) or directly as

$$s_{2i+k,i} = s_k = \frac{1}{2^d}\binom{d+1}{k} \tag{25}$$

where i is the row, k is the column, and d is the degree.

3.3 The subdivision matrix

Neglecting the boundaries of the curve for now we find the cubic subdivision matrix $\boldsymbol{S}$ as

$$\boldsymbol{S} = \frac{1}{8}\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ \cdots & 1 & 6 & 1 & 0 & 0 & \cdots \\ \cdots & 0 & 4 & 4 & 0 & 0 & \cdots \\ \cdots & 0 & 1 & 6 & 1 & 0 & \cdots \\ \cdots & 0 & 0 & 4 & 4 & 0 & \cdots \\ \cdots & 0 & 0 & 1 & 6 & 1 & \cdots \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{26}$$

where each column contains the coefficients $\frac{1}{8}(1,4,6,4,1)$ from equation (14) and is shifted two rows per column.

We have shown above that new control points are created by successive applications of the subdivision matrix $\boldsymbol{C}_{i+1} = \boldsymbol{S}\boldsymbol{C}_i$ and therefore can be written $\boldsymbol{S}^n\boldsymbol{C}_0$. We can see from this expression that the properties of the refined coefficients are *fully determined* by the properties of the subdivision matrix. Hence it is sufficient to look at $\boldsymbol{S}$ to find out the characteristics of the subdivision scheme. Ref. [Zorin and Schröder, 2000] shows that it is sufficient to look at the eigenvalues and eigenvectors of $\boldsymbol{S}$ to find out if the subdivision, in the limit, becomes

→ convergent, smooth, and invariant under affine transformations.

This requires that the largest eigenvalue is 1, that the second largest eigenvalue is smaller than 1, and that all other eigenvalues are strictly smaller than the second largest.

$$\begin{cases} \lambda_0 = 1 \\ \lambda_1 < 1 \\ \lambda_i < \lambda_1,\ i = 2,3,\ldots. \end{cases} \tag{27}$$

Sending the matrix from equation (26) above to Matlab we see that all these criteria are fulfilled and conclude that B-spline curve refinement through subdivision is a stable process with the desired characteristics.

3.4 Boundary constraints

Clearly, the subdivision we have seen so far will not work on the boundary since it is a weighted average of control points on both sides of the current point. We need to supply special subdivision rules for the boundaries. This is analogous to the analytical evaluation of a spline curve where each coefficient has an associated basis spline and away from the boundary all the basis splines sum up to one, but at the boundary we do not have full support since there exist no splines for the outside as shown in figure 9. The normal solution to this problem is to define different spline basis functions for the boundaries, so that the sum still becomes 1. This is shown in figure 13. The natural boundary condition is especially convenient when efficiency is important. It can be constructed directly from the uniform cubic splines. Simply pad the coefficient vector with two copies of the first and last values respectively and evaluate the spline at $t \in (1, n-1)$, where n is the number of coefficients to get the same result.

For the subdivision case we need to define rules for how to subdivide close to a boundary. This was first done in [Lane and Riesenfeld, 1980] which gives boundary rules as

$$\begin{aligned} &s_{0,0} = 1 \\ &s_{1,0} = .5 \qquad s_{1,1} = .5 \\ &s_{n-1,m-1} = .5 \quad s_{n-1,m} = .5 \\ &s_{n,m} = 1 \end{aligned} \tag{28}$$

3.5 Evaluation by repeated subdivision

Above we saw how to evaluate a spline curve analytically. Another way of finding the curve is by successive applications of subdivision. We have the full cubic subdivision matrix with boundary con-

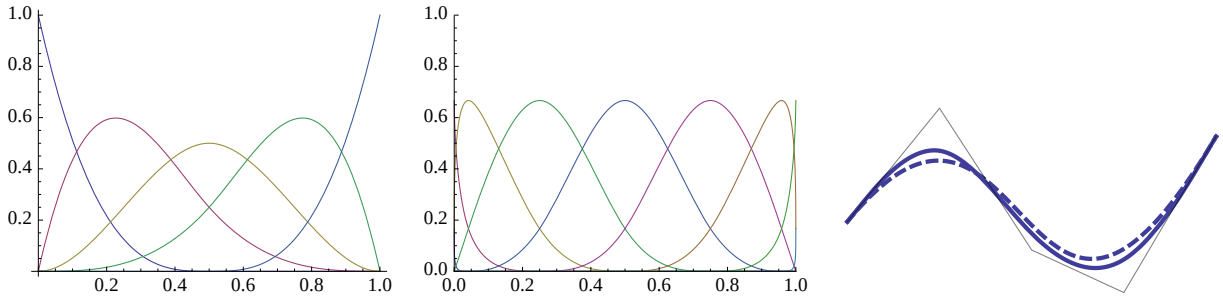


Figure 13: Modified cubic B-splines at a boundary will give full support. (a) "clamped boundary" splines. (b) Natural boundary splines. (c) The resulting curves using the same coefficients as in figure figure 9 with clamped boundary condition (dashed curve) and natural boundary condition (solid curve).

straints as

$$\boldsymbol{S} = \frac{1}{8}\begin{pmatrix} 8 & 0 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 & 0 \\ 1 & 6 & 1 & 0 & 0 \\ 0 & 4 & 4 & 0 & 0 \\ 0 & 1 & 6 & 1 & 0 \\ 0 & 0 & 4 & 4 & 0 \\ 0 & 0 & 1 & 6 & 1 \\ 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 0 & 8 \end{pmatrix} \tag{29}$$

in this case for a coefficient vector of length 5. The result from multiplying this matrix with $\boldsymbol{C}_0$ is $\boldsymbol{C}_1$ which will have size 9. An example of curve subdivision is shown in figure 14. Inspecting the product $\boldsymbol{C}_{i+1} = \boldsymbol{S}\boldsymbol{C}_i$ we see that in practice we do not need to store a full matrix. Except for the boundaries, there are just two rules.

$$\begin{aligned} \boldsymbol{c}'_i &= \tfrac{1}{8}\left(1\boldsymbol{c}_{i-1} + 6\boldsymbol{c}_i + 1\boldsymbol{c}_{i+1}\right) \\ \boldsymbol{c}'_{i+\frac{1}{2}} &= \tfrac{1}{8}\left(4\boldsymbol{c}_i + 4\boldsymbol{c}_{i+1}\right) \end{aligned} \tag{30}$$

Each "old" coefficient ($\boldsymbol{c}_i$) will be re-weighted and between every two old coefficients a new one ($\boldsymbol{c}_{i+\frac{1}{2}}$) will be inserted. We can simply loop over the coefficient vector and create the subdivided values on the fly. See figure 15 for a visual depiction. The corresponding boundary refinements simply become:

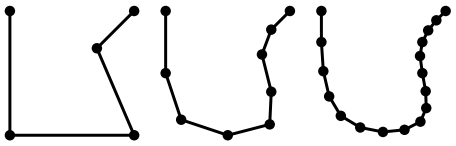


Figure 14: Spline curve subdivison.

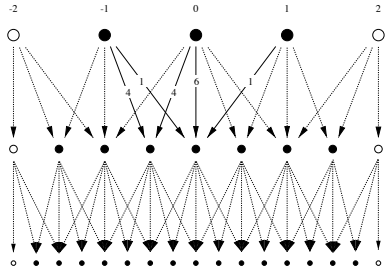


Figure 15: The weights for new coefficients according to cubic spline subdivison.

$$\begin{aligned} \boldsymbol{c}'_0 &= \boldsymbol{c}_0 \\ \boldsymbol{c}'_{end} &= \boldsymbol{c}_{end} \end{aligned} \tag{31}$$

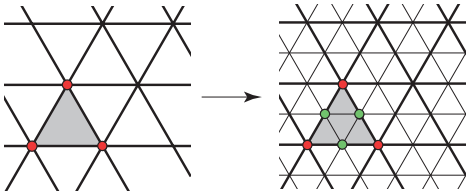


Figure 16: The triangle refinement rules for Loop subdivision.

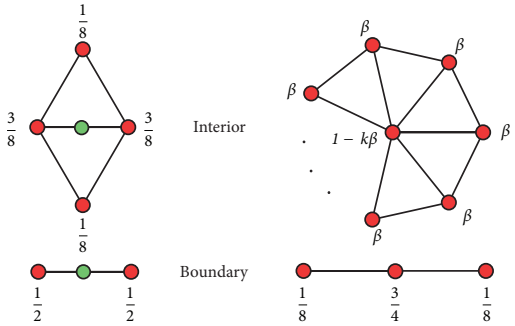


Figure 17: The weights for the new vertex positions in the Loop subdivision scheme. k is the valence (number of incident edges).

4 Mesh subdivision

A subdivision surface is a method of representing a smooth surface via the specification of a coarser piecewise linear polygon mesh. The smooth surface can be calculated from the coarse mesh as the *limit* of an iterative process of subdividing each polygonal face into smaller faces that better approximate the smooth surface.

4.1 Terminology

Valence is another word for the number of incident edges to a vertex. An ordinary vertex has a special valence and different mesh types have different valence for this. For triangular meshes the ordinary valence is 6, and for quad meshes it is 4. An *extraordinary vertex* is a vertex that is not ordinary. The valence dictates how smooth the subdivided mesh becomes, thus it is important that the rules do not generate new extraordinary vertices. Therefore, special rules are made for extraordinary vertices that keep the resulting mesh smooth.

4.2 Loop subdivision

The loop subdivision scheme [Loop, 1987] is a straightforward mesh refinement approach, it is based on a generalization of box splines to irregular meshes of arbitrary topology. It is, in the limit, C^2 continuous at regular vertices and at least G^1 elsewhere. It is straightforward to implement, given access to face and vertex neighborhoods. Each triangle in the original mesh is split into four new triangles (See figure 16). The new vertex positions are weighted averages of the old ones. In figure 17 the weights for both boundary and internal triangles are listed. We use the newer proposed [Hoppe et al., 1994] formula to calculate β as

$$\beta = \begin{cases} \dfrac{3}{8k}, & k > 3 \\ \dfrac{3}{16}, & k = 3 \end{cases} \tag{32}$$

5 Adaptive subdivision

Any subdivision will generate a very large amount of triangles very fast. However, we do not necessarily need the same level of refinement everywhere which brings up the topic of adaptive subdivision. The classification for which regions to subdivide can in principle be depending on any function. In [Amresh et al., 2002] on such scheme is introduced which classifies regions according to their "flatness". Other schemes includes screen space coverage per triangle, curvature, or user-defined constraints. The main issue for adaptive subdivision is how to handle boundaries between regions of different subdivision. In figure 18 a method for triangle splitting to ensure a connected mesh is shown.

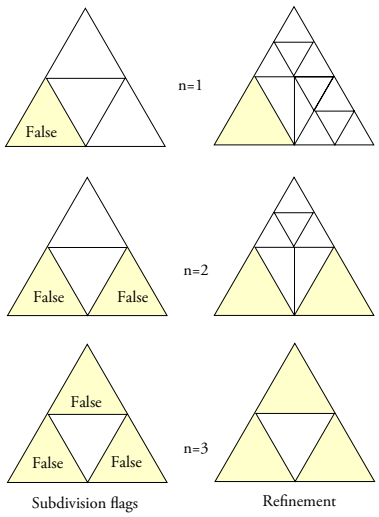


Figure 18: The three possible cases for a triangle that is subdividable (white, true) but has at least one false (yellow) neighbor.

5.1 Classifying faces

A criterion for adaptive subdivision is described in [Amresh et al., 2002]. This method assumes that we do not need to subdivide flat regions of the mesh. In practice, this is done by classifying all triangles in the mesh as either being *subdividable* or not. A triangle is classified by calculating the angle between its face normal and the normal of each of the neighbor faces to that triangle. If *all* of these angles are less than some critical angle α, the triangle is treated as being not subdividable (false).

Now, for each face that is *subdividable* we calculate the numbers of neighbors that are *not subdividable*. This is one, two or three for a triangle. From this number we then create a case table for how to treat the triangle, this is shown in figure 18. The reason why we need to treat these cases is to see to it that the edges between subdivided and not subdivided triangles match.

5.2 Sharp features

Another early name for the approximating subdivision schemes was "corner cutting" which stems from the smoothing principle of subdivision. We can use this property to our advantage. Given an adaptive subdivision scheme, we can tag sharp faces and edges as "not subdividable" which will keep the sharpness from being smoothed away. See [DeRose et al., 1998] for more information.

6 Assignments

6.1 Grading

The assignments marked as (3) are mandatory to pass the lab, grade 3. Completing the assignment (4) gives you the course grade 4, while completing assignments (4,5) give you a 5.

Implement curve subdivision (3)

We have implemented a simple line geometry in the class `LineStrip` which forms the foundation for this assignment. This class simply describes a line as discrete sample points which are connected linearly. The task of this assignment is to implement `UniformCubicSplineSubdivisionCurve` which, given a line strip, approximates a spline by subdividing each line strip segment. Compare your implementation with the supplied theory to make sure your subdivided curve has the *correct number of points* and correct point weights (especially near the ends). Add a curve using the menu option "Add object / Subdivision curve".

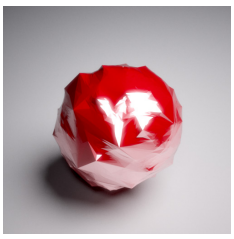
This subdivision curve should, in the limit, converge to the analytical cubic B-spline given the same control points. Use the class `UniformCubicSpline` to visually compare the accuracy of your subdivided curve at different levels of subdivision and discuss any differences. Add an analytical curve using the menu option "Add object / Cubic spline".

Implement mesh subdivision (3)

The class `LoopSubdivisionMesh` implements a simple subdivision scheme that creates new vertices halfway along each edge. Thus, it only adds more triangles without any additional details or smoothness. You have to modify the functionality in `VertexRule()` and `EdgeRule()` to provide better placements of the additional vertices by the Loop subdivision scheme (section 4.2). Before you begin, study the main algorithm in `Subdivide()` to correlate the code with the text in section 4.

Localize evaluation of the spline (4)

The analytical spline implemented in the class `UniformCubicSpline` currently has a naïve implementation of the evaluation in `GetValue()`. Recall that a spline is evaluated by a summation over a set of control points supported (multiplied) by basis functions. Localize the current computations by restricting the summation to only those control points which are supported by the basis functions at a given point t. Note that the analytical spline is defined by a line strip where the points of the line strip are the control points of the spline. The parameter t is defined such that $t \in (0, n)$ where n is the number of control points. The B-spline evaluation function increments a counter for each call, so you can read the output in the console window to see how many times this function is called for each rendering.



Implement a scheme for adaptive mesh subdivision (5)

The purpose of this task is to construct a scheme/rule for the "subdividability". The class `AdaptiveLoopSubdivisionMesh` implements code for adaptive subdivision but is depending on the rule in `Subdividable()` in order to decide whether a face is subdividable or not. Study the code `AdaptiveLoopSubdivisionMesh` to make sure you understand the workings of the algorithm. Then modify the function `Subdividable()` in `StrangeSubdivisionMesh` to something more clever (or even more artistic). Your scheme should be different from what was used in Lab 2 and should work iteratively with a relative criteria (avoid fixed values!). Be prepared to motivate your choice.

7 Acknowledgements

The lab scripts and the labs were originally developed by Gunnar Läthén, Ola Nilsson, Andreas Söderström and Stefan Lindholm. They based in turn the lab on Wikipedia, on Zorins Siggraph course notes [Zorin and Schröder, 2000] and on the spline course notes from Lyche and Mörken [Lyche and Mörken, 2007].

References

Ashish Amresh, Gerald Farin, and Anshuman Razdan. Adaptive subdivision schemes for triangular meshes, 2002.

Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 85–94, New York, NY, USA, 1998. ACM Press.

David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

Chris Hecker. Let's get to the floating point. Feb 1996. URL `http://chrishecker.com/images/f/fb/Gdmfp.pdf`.

Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 295–302, New York, NY, USA, 1994. ACM Press.

J.M. Lane and R.F. Riesenfeld. A theoretical development for the computer display and generation of piecewise polynomial surfaces. In *IEEE Transactions PAMI2*, 1980.

C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, University of Utah, August 1987.

Tom Lyche and Knut Mörken. Spline methods: Course notes. 2007.

Denis Zorin and Peter Schröder. Subdivision for modeling and animation: Siggraph 2000 course notes. In *SIGGRAPH '00: ACM SIGGRAPH 2000 Courses*, New York, NY, USA, 2000. ACM.