SPLINES AND SUBDIVISION

MODELING AND ANIMATION, LAB 3

Albert Cervin
albce943@student.liu.se

Monday 9th May, 2011

Abstract

In this paper I will present theory and results when implementing splines, curve- and mesh subdivision. This is the third lab in a lab series consisting of six labs in the course TNM079 - Modeling and Animation at Linköping University. Topics that will be covered include a general introduction to splines and subdivision, implementation of both curve and mesh subdivision, localization of spline evaluation and using vertex curvature to do adaptive mesh subdivision. The results will then be presented showing that the subdivision schemes for curves and meshes gives satisfying results. By localizing the analytical spline the efficiency of the calculations increases. The adaptive subdivision scheme based on vertex curvature also works well. The conclusion is that subdivision is very useful in different types of computer graphics and is currently a big area of research.

1 Introduction

The course TNM079 - Modeling and Animation consists of six lab sessions to be performed. Three of these sessions are to be presented in a report presenting what has been done and how it has been done. This report describes my work in the third lab session covering curve and mesh subdivision.

1.1 Curves and splines

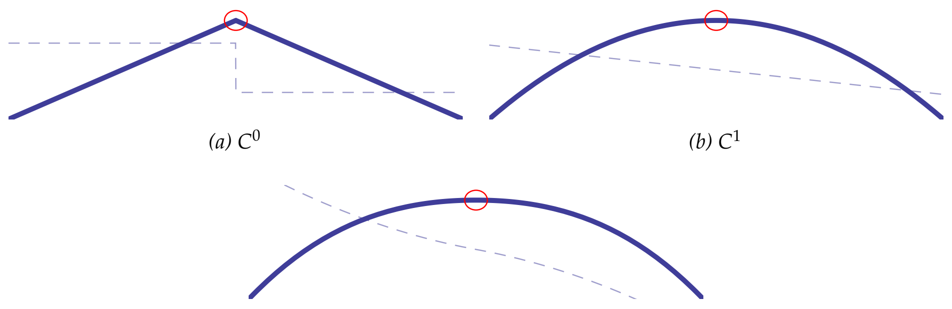
To describe a line in computer graphics it is for natural reasons impossible to use infinite descriptions and a *curve* is a parametric description of a line. A curve $p(t)$ is defined by a set of *coefficients* c_i and a set of *basis functions* b^i.

$$p(t) = \sum_{i=0}^{n} c_i b^i \tag{1}$$

Basis functions of a function f are functions that when linearly combined, gives the function f. That means that the use of basis functions controls the behavior of the resulting function. This gives the opportunity to choose basis functions in a way that fits a particular application of a curve. This is the reason that there exists some different types of curves like linear and cubic Bézier curves.

1.2 Piecewise curves and continuity

To create complex looking curves in computer graphics several curves are combined and adjusted to look like one single line. To achieve this it is important to consider joints between the different curves that make up the line. To measure the smoothness in the joints, continuity is used, denoted C^n. C^{-1} means (in some cases) that the curve is not continuous, the joints are not connected at all. C^0 Means that the joints are connected but no derivative is common between the two connected curve segments. C^1 means that the joints are connected and the first order derivative of the connected curve segments are equal. C^2 analogously, means that the second order derivative of the curve segments are equal. This is illustrated in figure 1.



(a) C^0 *(b)* C^1 *(c)* C^2

Figure 1: Different continuity.

1.2.1 Convex combination

A convex combination is a linear combination of data points. An important property of the convex combination is that all coefficients are positive and sum to one. The reason it is called convex combination is that the resulting convex combination will always be inside the convex hull of the data points.

1.3 Bézier curves

Bézier curves originates from France and from the French engineer Pierre Bézier who used the curves in the 1960s to design car bodies. The curves were developed earlier though, by Paul de Casteljau in the late 1950s. Today Bézier curves are for example used in PostScript fonts and in various desktop vector drawing tools such as Adobe Illustrator.

A linear Bézier curve is the simplest form of Bézier curve. It is essentially a linear interpolation between control points. This means that the linear Bézier curve is C^0 continuous.

$$p_1(t) = (1-t)c_0 + tc_1, \quad t \in [0,1] \tag{2}$$

1.4 B-splines

The B-spline is one of the most common types of splines [3]. It is C^2 continuous and will thus give smooth lines. The original basis functions for cubic splines is presented in equation 3.

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

The basis functions given in equation 3 are defined in the interval $[-2, 2]$. Furthermore, it can be shown that it is possible to define B-spline basis functions in terms of convolution. A B-spline of degree d is a convolution of B_{d-1} with B_0 where B_0 is

$$B_{i=0,0}(t) = \begin{cases} 1, & 0 \le t < 1 \\ 0, & otherwise \end{cases} \tag{4}$$

Defining a B-spline in terms of convolutions is very useful since it makes refinement of the spline easy. The refinement equation is

$$B_d(t) = \frac{1}{2^d}\sum_{i=0}^{d+1}\binom{d+1}{i}B_d(2t-i) \tag{5}$$

Equation 5 shows that a B-spline of arbitrary degree d can be written as a linear combination of translated and dilated copies of itself. Each refinement of the spline gives a one degree higher continuity. The refinement for a cubic B-spline is

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

1.5 Subdivision

1.5.1 Subdivision of spline curves

As seen in section 1.4 the basis functions of a spline can be rewritten by means of convolution. When rewriting the basis functions, the coefficients has to be changed too which is really the key to subdivision.

Consider a spline curve with coefficients $[c_0, c_1, ..., c_n]$ in vector form as **C** and the basis functions $[B_n(t), B_n(t-1), ..., B_n(t-n)]$ in a vector **B**. This means that the spline curve can be written as [3]

$$\mathbf{p}(t) = \mathbf{B}(t)\mathbf{C} \tag{7}$$

Putting the refinement coefficients in a matrix S gives that [3]

$$\mathbf{C}_{j+1} = \mathbf{S}\mathbf{C}_j \tag{8}$$

where the subscripts show the number of refinements. The entries of **S** is given from equation 5.

It is not possible however to apply this scheme since there will be problems at boundaries. Boundary conditions has to be introduced and with boundary conditions the full **S** matrix becomes [3]

$$\mathbf{S} = \frac{1}{8}\begin{pmatrix} 8 & 0 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 & 0 \\ 1 & 6 & 1 & 0 & 0 \\ 0 & 4 & 4 & 0 & 0 \\ 0 & 1 & 6 & 1 & 0 \\ 0 & 0 & 4 & 4 & 0 \\ 0 & 0 & 1 & 6 & 1 \\ 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 0 & 8 \end{pmatrix} \tag{9}$$

The above described way of evaluating a spline curve is suitable for analytical evaluation but for more practical implementation, successive applications of subdivision is simpler to implement. Inspecting the result of applying subdivision it can be seen that in practice there are only two rules (except for the boundaries).

$$\begin{array}{rcl} \mathbf{c}'_i & = & \frac{1}{8}\left(1\mathbf{c}_{i-1} + 6\mathbf{c}_i + 1\mathbf{c}_{i+1}\right) \\ \mathbf{c}'_{i+\frac{1}{2}} & = & \frac{1}{8}\left(4\mathbf{c}_i + 4\mathbf{c}_{i+1}\right) \end{array} \tag{10}$$

The boundaries can be handled by

$$\begin{array}{rcl} \mathbf{c}'_0 & = & \mathbf{c}_0 \\ \mathbf{c}'_{end} & = & \mathbf{c}_{end} \end{array} \tag{11}$$

The subscript $\frac{1}{2}$ refers to a new coefficient in between two already existing ones. To apply the scheme, the coefficient list is looped over and already existing coefficients are re-weighted and new coefficients are added. The algorithm is described in more detail below.

1.6 Mesh subdivision

The subdivision method described above only works for 2D curves and to apply subdivision to meshes, other schemes has to be used. In mesh subdivision the term *valence* appears. *Valence* is another word for the number of incident edges to a vertex. A vertex is furthermore called *ordinary* if it has a special valence (6 for triangle meshes and 4 for quad meshes). If this is not true, the vertex is *extraordinary*.

1.6.1 Loop subdivision

Loop suggested this scheme for triangle mesh subdivision in 1987 [2]. It is a generalization of the spline refinement to irregular meshes. It is in the limit C^2 continuous when dealing with regular vertices and at least G^1 elsewhere. For each triangle in the original mesh, four new are created as shown in figure 2.

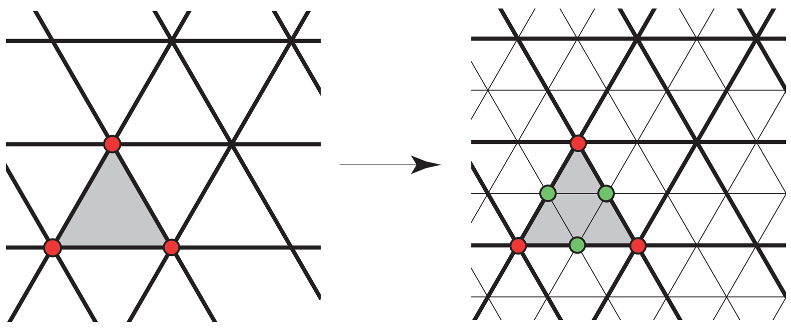


Figure 2: Rules for Loop subdivision (Image courtesy of [3]).

The new vertices are then placed according to figure 3

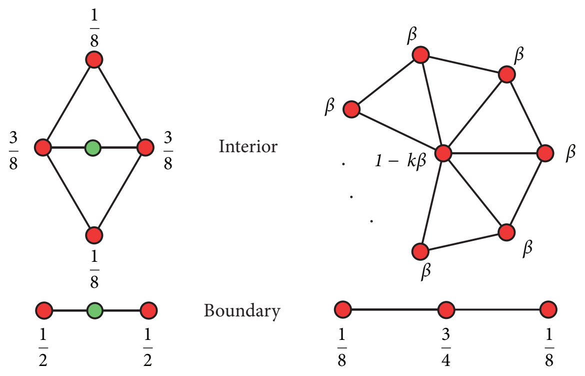


Figure 3: Weights for new vertex placement in Loop subdivision.

where β is calculated from the vertex valence as described by equation 12.

$$\beta = \begin{cases} \dfrac{3}{8k}, & k > 3 \\ \dfrac{3}{16}, & k = 3 \end{cases} \tag{12}$$

1.6.2 Catmull-Clark subdivision

This subdivision scheme suggested by Edwin Catmull and Jim Clark in 1978 can handle arbitrary polyhedrons which means that it is very often used for quad meshes in well known industry software such as 3D Studio Max, Blender and Maya. For more detailed information see [1].

2 Method

2.1 Implement curve subdivision

To subdivide a curve, essentially the only thing that has to be done is to introduce new coefficients in between old ones and place them in a way that gives a smoother curve. The placement of new coefficients are described in equation 10 and equation 11.

Pseudo-code for the algorithm is as follows. The old coefficients are located in an array of size N indexed between $[0..N-1]$.

```
// Construct array twice as big as original coefficient array
new_coeff = array(old_coeff.size()*2)

// Handle the first two
new_coeff.add(old_coeff[0])
new_coeff.add((1/8)*(4*old_coeff[0] + 4*old_coeff[1]))

for old_coeff[1..N-2]
{
    new_coeff.add((1/8)*(old_coeff[i-1] + 6*old_coeff[i]) + old_coeff[i+1])
    new_coeff.add((1/8)*(4*old_coeff[i] + 4*old_coeff[i+1]))
}

// Handle the last one
new_coeff.add(old_coeff[N-1])
```

The above algorithm was implemented in the function `Subdivide` in the file `UniformCubicSplineSubdivisionCurve.cpp`.

2.2 Implement mesh subdivision

The task given was to implement Loop subdivision. The code base for the lab already had the algorithm for adding new vertices implemented. This reduced the problem to finding new vertex positions.

Vertex positions are weighted according to figure 3. This means that vertices created on edges in the subdivided triangle gets a position described by equation 13.

$$v_{edge} = \frac{1}{8}(3v_0 + 3v_1 + v_2 + v_3) \tag{13}$$

Numeric errors in computers can be a problem in implementing equation 13 [3]. To address this problem, the equation is rewritten as

$$v_{edge} = 0.375(v_0 + v_1) + 0.125(v_2 + v_3) \tag{14}$$

before it is implemented.

Vertices that already existed in the original triangle also have to be re-evaluated. The new position of vertex i ($v_{i,new}$) is calculated according to equation 15

$$v_{i,new} = (1 - k\beta)v_i + \sum_{i \in N_1(i)} \beta v_{i,j} \tag{15}$$

In equation 15 $v_{i,j}$ is the j:th neighbor vertex to v_i.

Pseudo-code for the implementation of relocating a previously existing vertex v is as follows.

```
neighbors = neighboring vertices of v
valence = neighbors.size()

if (valence == 6)
    beta = 1/16
else if (valence == 3)
    beta = 3/16
else
    beta = 3/(8 * valence)

v_new = v

for all neighbors
    v_new += beta*neighbors[i]
```

The code for handling edge vertices was implemented in the function `EdgeRule` and the code for handling already existing vertices was implemented in the function `VertexRule`, both located in `LoopSubdivisionMesh.cpp`

2.3 Localize evaluation of the analytical spline

At most four basis functions can have effect on a spline at one point as can be seen in figure 4.

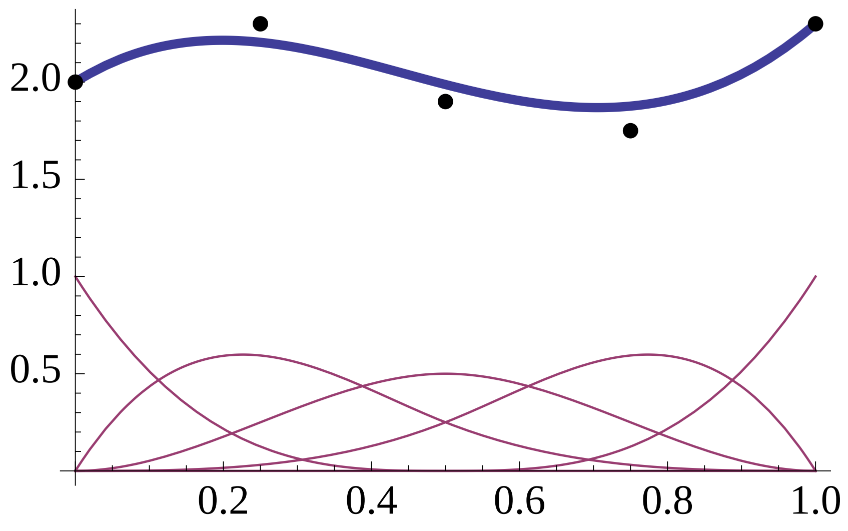


Figure 4: A cubic b-spline with basis functions in purple.

The evaluation of the analytical spline had a naive implementation where all control points are evaluated at each point on the curve, even though they are not supported by a basis function. Localizing the evaluation is only a matter of finding out which control points that are actually supported by basis functions at a given point *t*.

Pseudo-code for the implementation is as follows (The control points is located in an array of size *N* and *t* is the point at which the spline is evaluated).

```
index = floor(t)

bval = GetBSplineValue(index, t)
val = coefficients[i]*bval

if (i > 0)
{
    bval = GetBSplineValue(i-1, t)
    val += coefficients[i-1]*bval
}

if (i < N-1)
{
    bval = GetBSplineValue(i+1, t)
    val += coefficients.[i+1]*bval
}
```

```
if (i < N-2)
{
    bval = GetBSplineValue(i+2, t)
    val += coefficients.[i+2]*bval
}

return val
```

This algorithm was implemented in the function `GetValue` in `UniformCubicSpline.cpp`.

2.4 Implement a scheme for adaptive mesh subdivision

The task was to implement a "clever" scheme for doing adaptive mesh subdivision. The code for subdivision given in the source code for the lab session used the function `Subdividable` in `StrangeSubdivisionMesh.h`. This function returns a boolean indicating if the face shall be subdivided or not. The implementation supplied was rather unusable since it returned `faceIndex % 4 == 0` so it subdivided every fourth face.

Our idea is to use vertex curvature to define subdividability. If the vertex curvature is large, the face is important for fine details and thus should be subdivided. This will also result in big flat surfaces to not be subdivided to the same extent as curved surfaces.

The function `Subdividable` gets a face index as input and to get an estimate of vertex curvature we use the mean value of the three (in this case) vertex curvatures. After experimentation, we found a reasonable limit to be to subdivide when $|c| > 200.0$ where c is the mean vertex curvature.

3 Results

3.1 Implement curve subdivision

The subdivision works well but a few subdivisions is needed before it approximates the analytical spline in a good way. The result of 1, 2 and 10 subdivisions is shown in figure 5 in green and the analytical spline in red.

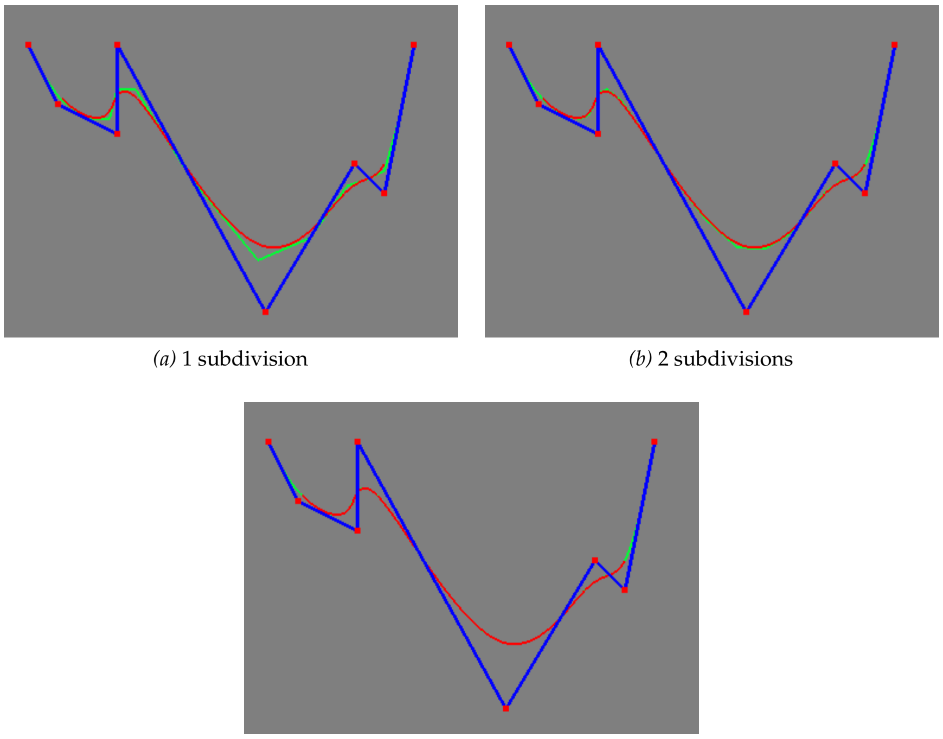
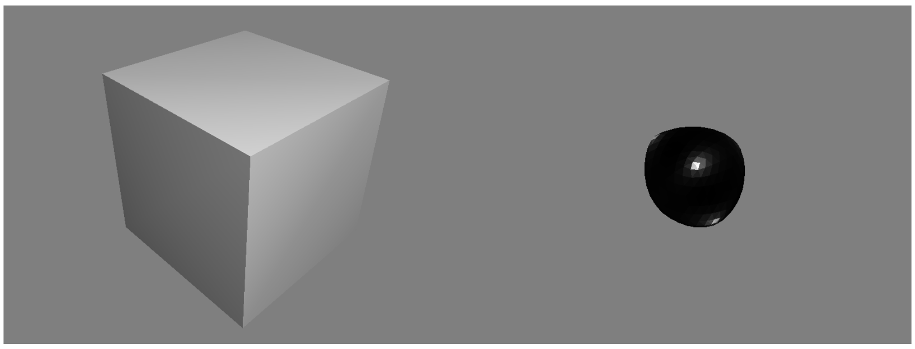

(a) 1 subdivision *(b)* 2 subdivisions *(c)* 10 subdivisions

Figure 5: Subdivision spline (green) and analytical spline (red).

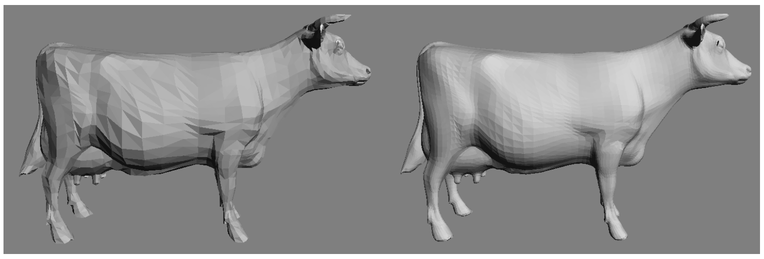
3.2 Implement mesh subdivision

The implementation of mesh subdivision works well and the object gets smoother for each subdivision. The results for the two objects `cube.obj` and `cow.obj` can be seen in figures 6 and 7, respectively. It can be seen in figure 6 that the result of subdividing the cube is rather unexpected. This is due to the fact that the valence are different for different vertices in the cube. To explain this, consider a quad which is essentially two triangles. Since the diagonal edge in the quad gives two of the vertices in the quad a higher valence, the valence of vertices in a quad are different. Therefore, the vertices get different weights in the Loop subdivision scheme and gets deformed as seen in figure 6b.

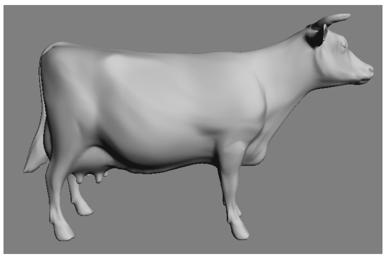


(a) No subdivision *(b)* 4 subdivisions

Figure 6: Subdivision results for `cube.obj`. The rather unexpected shape on the subdivided cube is due to varying valence between vertices.



(a) No subdivision *(b)* 1 subdivision



(c) 3 subdivisions

Figure 7: Subdivision results for `cow.obj`.

3.2.1 Performance

Times for subdividing the object `cow.obj` is presented in table 1. All times are measured with `ctime` under Microsoft Visual C++ and the time for updating the new mesh have been discarded.

Subdivision iteration	Elapsed time (s)
1	1.411
2	1.84
3	4.257

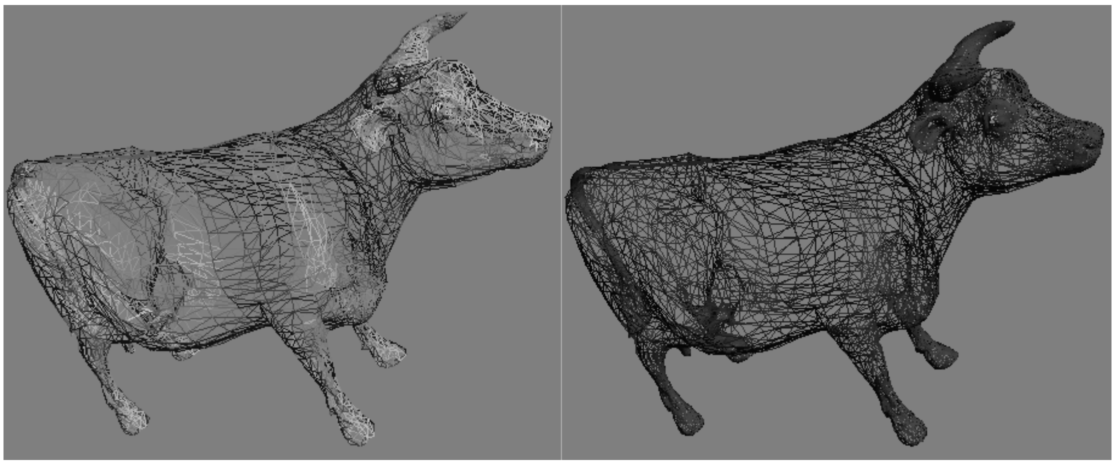
Table 1: Time elapsed subdividing `cow.obj`.

3.3 Localize evaluation of the analytical spline

The localization works well and the visual result is identical to the original implementation

3.4 Implement a scheme for adaptive mesh subdivision

The scheme suggested in section 2.4 gives good results and the experimental value of 200 seems to work fairly good for arbitrary meshes. Result for `cow.obj` can be seen in figure 8.



(a) No subdivision *(b)* 1 subdivision

Figure 8: Adaptive mesh subdivision applied to `cow.obj`. Note that the horns get more subdivided while the number of triangles remain basically the same on flat parts as the side of the cow.

A comparison between Loop subdivision and the adaptive mesh subdivision can be seen in figure 9.

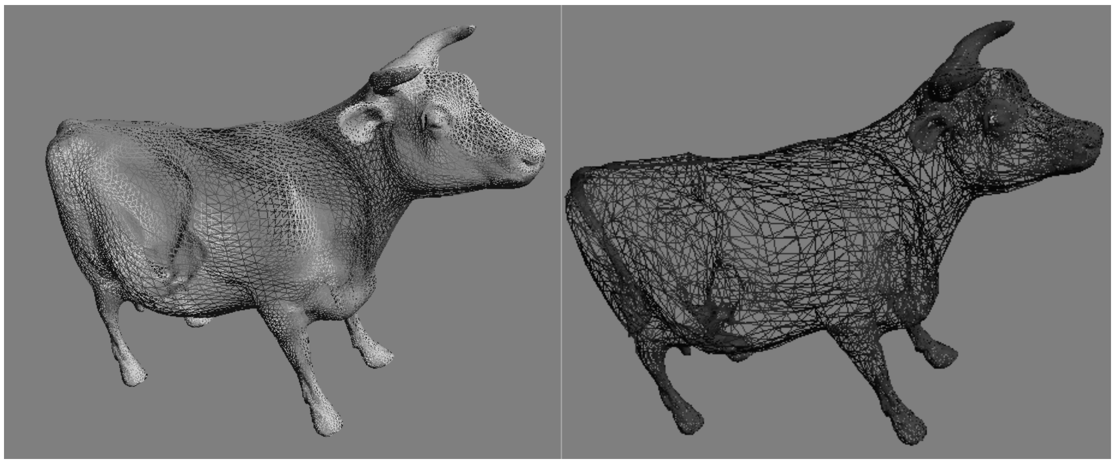

(a) Loop subdivision *(b)* Adaptive subdivision

Figure 9: Loop subdivision compared with adaptive subdivision (1 iteration).

4 Conclusion

From the results in section 3 and from the theory presented earlier it can be seen that it is easy to implement curve subdivision by repeated application of refinement. Some iterations is needed before the result is visually identical.

Loop subdivision also becomes much simpler when topology information is given as it is in the half edge mesh data structure. In figure 6 it is easy to note that the overall shape of a cube is lost and it looks rather deformed. This shows that the subdivision scheme has problems subdividing sharp geometries as a cube due to varying valence in the mesh.

The localization of the analytical spline evaluation achieves the same result as before as expected but increases efficiency of the calculations. This is since unnecessary calculations with no contribution are stripped out.

The adaptive subdivision scheme works well and the result is desirable. To refine details and at the same time keep computational cost down. The limit for curvature should be adapted to the range of curvature of the object if it were to be applied in real applications. However it works reasonably well for the purpose of this lab.

The overall conclusion for this lab is that subdivision is a very important aspect of computer graphics and a big area of research.

Lab partner and grade

I did this lab together with Nathalie Ek (natek725). All tasks marked with * was completed, as well as the task marked with ** and also the task marked with ***, which should qualify me for grade 5.

References

[1] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, 1978.

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

[3] Gunnar Läthén, Ola Nilsson, and Andreas Söderström. Splines and subdivision. Technical report, ITN, 2011.