Loop Subdivision for Triangle Meshes

Reeder Ward, David Melamed and Majbrit Schöttner

May 28, 2021

Abstract

This documentation describes what subdivision algorithms look like and how the principle works. This is shown with the examples of an open polygon and tetrahedron. This documentation shows Chaikin's algorithm, cubic subdivision and triangle subdivision and how these algorithms work in more detail based on the mathematical foundation and implemented in C++.

1 Subdivision

Subdivision is a practice in geometric modelling that can be used to smoothly represent angular objects. There are a number of approaches to subdivision, one of the best known being the corner-cutting approach of 1974 by George Chaikin.

1.1 Subdivision with curves

In elementary terms, a curve consists of several straight lines. A first step is to calculate a curve, which is considered as an open polygon, more smoothly by the technique of subdivision. With the execution of the subdivision algorithm, a new open polygon containing more points is calculated based on the old open polygon. The new open polygon is used as the basis for calculating another execution of the subdivision algorithm.

1.2 Subdivision with surfaces

To make a 3D polygon smoother by subdivision, the object is first divided into the base surfaces, then divided into polygons, and finally, the polygons are divided into vertexes (points) and edges (edges) which represent the essential elements of the subdivision.

2 Related work

One of the first and most important researches on subdivision was done in 1978 by Edwin Catmull and Jim Clark, after which more researches moved in the direction of graphical data processing. The PIXAR company took up the research and added more functions to the subdivision. This led to further research into simplifying and more efficiently processing surfaces. PIXAR sat down with Microsoft, and together they created the OpenSubdiv initiative. OpenSubdiv was intended to solve the major problem of cross-software incompatibility, as well as to improve Subdivision in general and make it available to everyone.

One of the most famous discoveries, which also found a place in OpenSubdiv, was made by Matthias Nießner. He wrote his dissertation on ”Rendering Subdivision Surfaces using Hardware Tessellation”. His work shows that you don't need big machines to run subdivision. He also talks about how subdivision can be done with less computer effort with the same level of detail as the practice has shown so far. He was referring exclusively to interactive programs like video games. [3]

The first developer who put the subdivision into practice on consumer level was Autodesk Inc. with MAYA. MAYA is a software that combines powerful tools for 3D animation and implements modeling, simulation, rendering and compositing. AutoDesk has also researched in the direction of subdivision in detail. Jos Stam writes in his paper ”Evaluation of Loop Subdivision Surfaces” about an addition to the original idea of the Catmull-Clark Subdivision. Stam describes how by using different parameters, different rules and a different eigenanalysis the idea of ”extraordinary vertices” can be used in a meaningful way, leading to a more efficient application. [2]

3 Known methods

3.1 Mathmatical formulae for curves

Chaikin's algorithm can be represented as a weighted sum of the points P_i^{k-1} of the previous iteration. [1] Thus, Chaikin's algorithm can be represented as follows, where real coefficients = a_{ijk}:

$$\sum_{i=0}^{n_k-1} a_{ijk} P_i^{k-1} \tag{1}$$

The coefficients for Chaikin's algorithm can be seen in matrix 2 and those for cubic interpolation can be seen in matrix 3.

$$a = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & \dots & 0 \\ 0 & \frac{3}{4} & \frac{1}{4} & \dots & 0 \\ 0 & \frac{1}{4} & \frac{3}{4} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{2}$$

$$a = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ \frac{3}{8} & \frac{3}{4} & -\frac{1}{8} & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ -\frac{1}{16} & \frac{9}{16} & \frac{9}{16} & -\frac{1}{16} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{3}$$

[4]

3.2 Mathmatical formulae for surfaces

Where e: edge point, v0 and v1: points spanning the edge, v1 and v2: third point of each of the two triangles involved, this is the equation for the **Edge Mask**:

$$e' = \frac{3}{8}(v_0 + v_1) + \frac{1}{8}(v_2 + v_3) \tag{4}$$

Vertex Mask:

$$v' = \beta(n)\mathbf{v} + \frac{1-\beta(n)}{n} \sum_{i=0}^{n} \mathbf{e_i'} \tag{5}$$

Alpha:

$$\alpha = \frac{3}{8} + \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2 \tag{6}$$

Beta:

$$\beta = \frac{8}{5}\alpha(n) - \frac{3}{5} \tag{7}$$

n is the number of triangles in which a certain point occurs. [4]

4 C++ implementation

4.1 Chaikin's algorithm and cubic interpolation

The following is the implementation of cubic interpolation. The code for Chaikin's algorithm looks similar, but contains different numbers. A matrix is calculated, which is created in different sizes depending on how many points there are. Then the new points are calculated using a matrix multiplication of the matrix 3 and the vectors of the old points.

```
// fill center of the matrix
int c = columns-1;
int l = (lines-3)*2;
int i=2;
int j=0;
while( i<l) {
    while(  j<c) {
        matrix[i][j+1]  = 1;
        matrix[i+1][j]  = -0.0625;
        matrix[i+1][j+1]  = 0.5625;
        matrix[i+1][j+2]  = 0.5625;
        matrix[i+1][j+3]  = -0.0625;
        j=j+1;
        break;
    }
    i=i+2;
}
//fill first two and last two lines of the matrix
matrix[0][0]  = 1.0;
matrix[1][0]  = 0.375;
```

```
matrix[1][1] = 0.75;
matrix[1][2] = -0.125;
matrix[lines-1][columns-1] = 1.0;
matrix[lines-2][columns-1] = 0.375;
matrix[lines-2][columns-2] = 0.75;
matrix[lines-2][columns-3] = -0.125;
matrix[lines-3][columns-2] = 1.0;

//---matrix multiplication---
float xproduct = 0;
float yproduct = 0;
float zproduct = 0;
for( int i=0; i<lines; i++) {
    xproduct = 0;
    yproduct = 0;
    zproduct = 0;
    for( int j=0; j<columns; j++) {
        //calculate x,y and z of one new point
        xproduct += matrix[i][j]*xold[j];
        yproduct += matrix[i][j]*yold[j];
        zproduct += matrix[i][j]*zold[j];
    }
    ...
}
```

4.2 Triangle subdivision

To create the triangle mesh, the neighboring triangles are calculated for each subdivision, then the new points are calculated from the old points and for each triangle its three points are calculated. After the last subdivision, the normal vector of each triangle is calculated so that the triangles can be drawn, and it is known where the outer color should be and where the inner one.

To determine the neighboring triangles of a triangle, the vector containing the triangles is through-iterated and tested for each triangle which of the other triangles has two of the three points. Three tests are performed for this purpose. Test1 tests whether the first point of the triangle matches one of the other points of the other triangle. Similarly, this is done for the second and third points in test2 and test3, which can be seen in the following code section. If two of the tests return true and the third false, it is a neighbor triangle and the index of this neighbor triangle is stored in the triangle in the array *it*[].

When going through the vector of triangles, for each point of the respective triangle the counter for the number of triangles of each point is incremented by one.

```
// for every Triangle in tris find the 3 neighbor
   ↪ Triangles
for(unsigned int t=0; t<tris.size(); t++){
    for(unsigned int tn=0; tn<tris.size(); tn++){
        // For the 3 points of the triangle test if they
           ↪ occur in the other triangles
        test1 = (tris[t].iv[0]==tris[tn].iv[0] || tris[t
           ↪ ].iv[0]==tris[tn].iv[1] || tris[t].iv[0]==
           ↪ tris[tn].iv[2]);
        test2 = ...
        test3 = ...
        // find t0!=t containing b and c
        if(test2 && test3 && !test1){
            tris[t].it[0]=tn;
        }
        // find t1!=t containing c and a
        ...
        // find t2!=t containing a and b
        ...
    }
    // increase no. of triangles
    valences[tris[t].iv[0]] += 1;
    valences[tris[t].iv[1]] += 1;
    valences[tris[t].iv[2]] += 1;
}
```

To calculate the edge points, which are located near the edges between two points called v0 and v1, the formula 4 is used. Other points used to calculate the edge points are the third points of each triangle containing v0 and v1. These points are called v2 and v3. In each triangle, the array *iv*[] stores the indices of the three vertices of the triangle. In a matrix *mv*[][] three indices of *iv*[] are stored for each of the three edges, which are respectively the edge connection points and the third point of the one triangle.

It is iterated through each triangle in the triangle vector. An edge point is calculated for each edge of the triangle. For this purpose, the neighboring triangle is needed. If the neighbor triangle already occurred earlier in the triangle vector, i.e. has a lower index, the edge point is not calculated, because it was already calculated. In the other case, it is tested which point of the neighboring triangle did not occur in the triangle that is in turn. The edge point is then calculated from the four points of the triangle and the neighboring triangle. The edge point is stored in the vertex vector and its index is stored in the *ie*[] array of the two triangle objects.

```
//matrix indices of iv
int mv[3][3] = {{1, 2, 0}, {2, 0, 1}, {0, 1, 2}};
...
    //calculate edge point
    if(i<t.it[j]) {
        //find Vertex in neigbor Triangle
        //that is not in Triangle t
        test1 = t.iv[i1]==n.iv[0]
        || t.iv[i2]==n.iv[0];
        test2 = ...
        test3 = ...
        if(!test1) {
            d = pts[n.iv[0]];
        }else if(!test2) {
            d = pts[n.iv[1]];
        }else if(!test3) {
            d = pts[n.iv[2]];
        }
        //with formula
        e = (pts[t.iv[i3]] + 3*pts[t.iv[i1]] + 3*pts[t.iv
           ↪ [i2]] +d) * 0.125;
        tris[i].ie[j] = pts.size();
        pts.push_back(e);
        //fill ie in neighbor Triangle
        if(i==n.it[0]) {
            tris[t.it[j]].ie[0] = pts.size()-1;
        }else if(i==n.it[1]) {
                tris[t.it[j]].ie[1] = pts.size()-1;
        }else if(i==n.it[2]) {
            tris[t.it[j]].ie[2] = pts.size()-1;
        }
```

For all vertices that were already in the vertex vector before the edge points were calculated, new x, y and z coordinates are calculated. For this, alpha is calculated with the formula 6 and then beta with the formula 7. Beta is inserted into the formula 5 and so the new positions of the points are calculated, which can be seen in the code section below.

```
//Vertex mask; for first second and third point of every
  ↪ triangle
for (int i=0; i<(int)tris.size(); i++) {
   pts[tris[i].iv[0]] +=  ( ( (1.0 - beta_n(valences[tris
      ↪ [i].iv[0]])) / (valences[tris[i].iv[0]])) * ((
      ↪ pts[tris[i].ie[1]]+pts[tris[i].ie[2]]) / 2.0));
    pts[tris[i].iv[1]] += ...
    pts[tris[i].iv[2]] += ...
    }
```

Four new triangles are created from one triangle. One of the new triangles consists of the three edge points, the other three triangles consist of two edge points and one corner point of the original triangle. How this is implemented can be seen in the code section below. All new triangles are stored in a triangle vector.

```
//calculate middle triangle
Triangle t2 = Triangle(t.ie[0], t.ie[1], t.ie[2]);
//calculate side triangles
Triangle t0 = Triangle(t.ie[1], t.ie[0], t.iv[2]);
Triangle t1 = Triangle(t.iv[0], t.ie[2], t.ie[1]);
Triangle t3 = Triangle(t.ie[2], t.iv[1], t.ie[0]);
```

When all the subdivisions have been made and the final vertices and triangles are determined, the normal vector of each triangle is calculated. For this, one vertex of the triangle is subtracted from the other two vertices. Thus, one receives the vectors of two edges. The cross product of these two vectors is the normal vector. The normal vectors of all triangles are stored in a vector so that they can be retrieved when drawing.

5 Numerical Examples

5.1 The curves

The curve made by the Chaikin's algorithm can be seen in figure 1 . A further development of Chaikin's algorithm is made possible by choosing other constants, which can be seen in figure 2. The cubic interpolating curve is essentially based on other parameters.

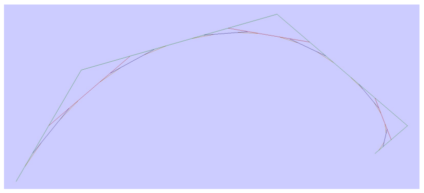
Figure 1: Chaikin's Algorithm curve

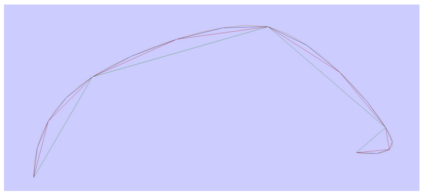
Figure 2: Cubic subdivision curve

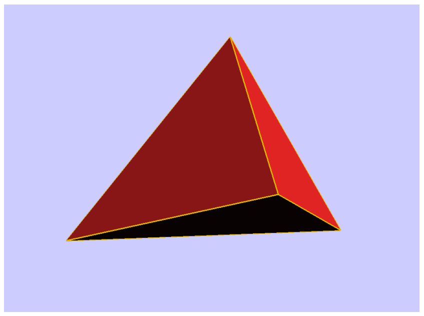
Figure 3: Tetra

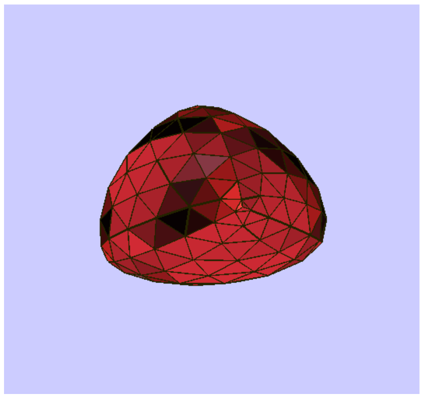
Figure 4: Subdivisions of tetra

5.2 The meshes

The following table shows the number of vertices and faces of a tetra at the beginning and after each subdivision.

	no subdivision	**subdivision 1**	**subdivision 2**	**subdivision 3**
vertices	4	10	34	130
faces	4	16	64	256

As can be seen in the table, there is a regularity to the triangles by which they increase with each subdivision. With each subdivision, four new triangles are assigned to a triangle, but a point can also have several triangles assigned to it.

In Figure 4, an object can be seen that was originally a tetra on which the subdivision algorithm was performed three times. The tetra can be seen in Figure 3.

In figure 5, 7 and 9 further objects are visible, whose subdivisions can be seen in figure 6, 8 and 10.

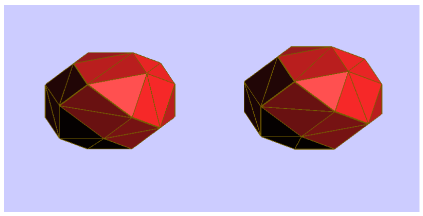
Figure 5: Billiard balls

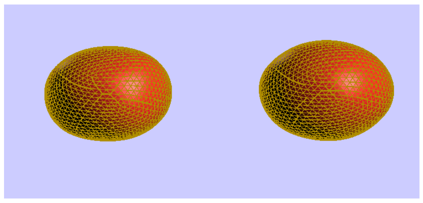
Figure 6: Subdivision of billiard balls

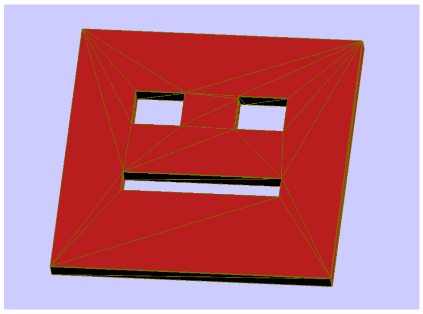
Figure 7: Mad face

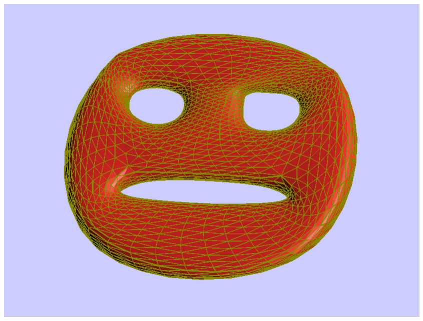
Figure 8: Subdivision of mad face

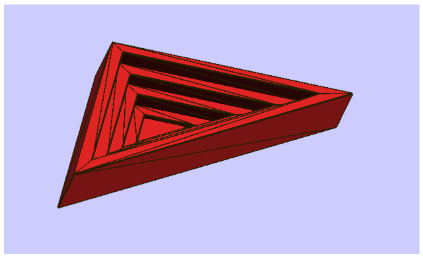
Figure 9: Crazy tetra

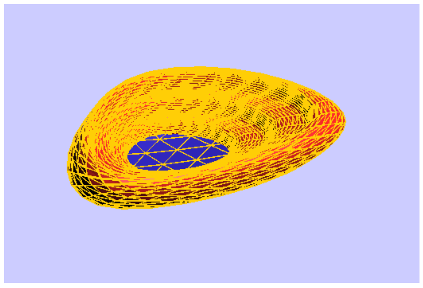
Figure 10: Subdivision of crazy tetra

6 Conclusion and future work

Different types of subdivision have been described. For the representation of curves, Chaikin's algorithm and cubic interpolation were used. To subdivide surfaces, triangular subdivision was used. A new point was calculated for each edge that is part of two triangles. New positions for the old triangle points were calculated. Four new triangles were composed of the repositioned triangle points and the new edge points, each replacing an old triangle. If this subdivision is performed several times, curves of the objects are displayed more and more finely and clearly. The subdivision algorithm could be successfully performed on a tetra and other objects.

In the future it should be found out why the subdivision of ”crazy tetra” shows blue spots, i.e. the inner surface of the object can be seen, although it is not the case without subdivision. Furthermore, it should be tested whether the algorithm can also subdivide more complicated objects with more triangles. The code should be optimized so that a faster calculation can be performed to better calculate a larger number of subdivisions, which is needed to produce a finer representation of the rounding.

7 Acknowledgements

We would like to extend a big thank you to the proofreader, Adam Ferber. Another big thank you goes to Kerstin Schöttner, who extensively checked our content for typos and grammatical errors. Finally, we would also like to thank our computer science professor, Martin Hering-Bertram, for being helpful in a timely manner whenever we had questions.

References

[1] David Salomon. *The Computer Graphics Manual.* `https://doi.org/10.1007/978-0-85729-886-7_15`, (visited on 05/27/2021). Northridge, USA, 2011.

[2] Jos Stam. *Evaluation of Loop Subdivision Surfaces.* `https://damassets.autodesk.net/content/dam/autodesk/www/autodesk-reasearch/Publications/pdf/evaluation-of-loop-subdivision.pdf`, (visited on 05/26/2021). Seattle, USA, n.d.

[3] Matthias Nießner. "Rendering Subdivision Surfaces using Hardware Tessellation". `https://niessnerlab.org/papers/2013/1thesis/niessner2013thesis.pdf`, (visited on 05/27/2021). PhD thesis. Erlangen, Germany, 2013.

[4] Prof. Dr. Martin Hering-Bertram. *Computer Graphics 2021 (CG21_0 and CG21_1).* `https://aulis.hs-bremen.de/ilias.php?ref_id=1393672&cmdClass=ilrepositorygui&cmdNode=wg&baseClass=ilrepositorygui`, (visited on 05/27/2021). Bremen, Germany, 2021.