Loop subdivision for triangle meshes

Reeder Ward, David Melamed and Majbrit Schöttner

May 16, 2021

Abstract

In this documentation is described how a self-implemented version of Chaikin's Algorithm looks like and how the principal works. This is shown with the examples of an open polygon and a three dimensional triangle object.

1 The subdivision

Subdivision is a practice in geometric modelling that can be used to smoothly represent angular objects. In order for subdivision to work, 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.

1.1 Subdivision with curves

A well known subdivision algorithm based on the idea of corner-cutting was explored by George Chaikin in 1974. A polygon is generated from an existing curve by cutting the corners of the original curve with each iteration. In one iteration, the individual lines are divided by constant ratios, which are 1/4 and 3/4, and thus the corner-cutting is calculated using this ratio pattern.

1.1.1 Mathmatical formulae

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

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

1.1.2 C++ implementation

```
// fill center of the matrix
int c = columns-2;
int l = (lines-3)*2;
int i=2;
int j=1;
while( i<l) {
    while(  j<c) {
        matrix[i][j] = 0.75;
        matrix[i][j+1] = 0.25;
        matrix[i+1][j] = 0.25;
        matrix[i+1][j+1] = 0.75;
        j=j+1;
        break;
    }
    i=i+2;
}

// fill first two and last two lines of the matrix
matrix[0][0] = 1.0;
matrix[1][1] = 0.5;
matrix[1][0] = 0.5;
matrix[lines-1][columns-1] = 1.0;
matrix[lines-2][columns-1] = 0.5;
matrix[lines-2][columns-2] = 0.5;
```

1.1.3 The curve

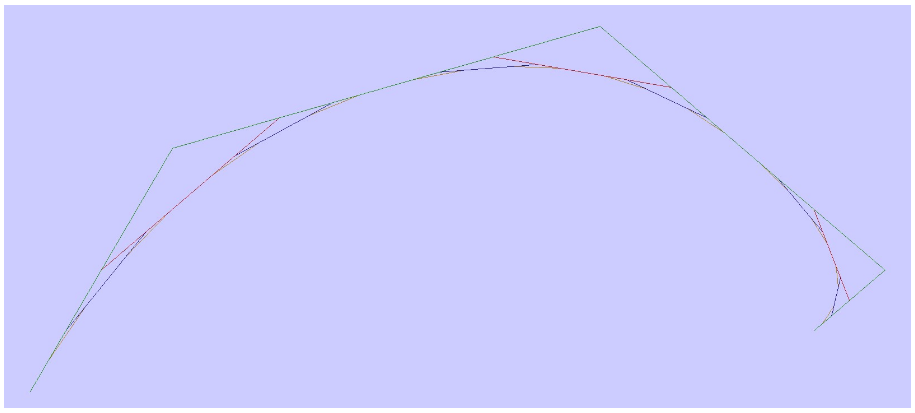

Figure 1: Chaikin's Algorithm curve.

A further development of Chaikin's algorithm is made possible by choosing other constants. The cubic interpolating curve is essentially based on other parameters.

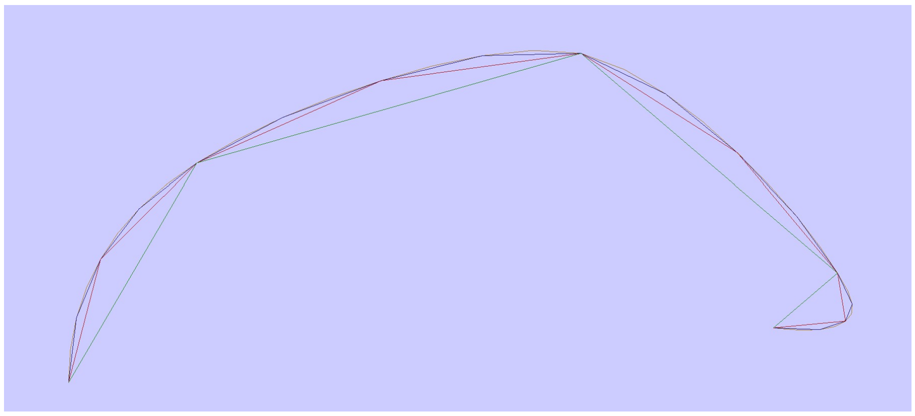

Figure 2: Interpolating cubic subdivision curve.

1.2 Loop subdivision scheme

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:

$$e' = \frac{3}{8}(v_0 + v_1) + \frac{1}{8}(v_2 + v_3) \tag{2}$$

Vertex Mask:

$$v' = \beta(n)\mathbf{v} + \frac{1-\beta(n)}{n}\sum_{i=0}^{n}\mathbf{e'_i} \tag{3}$$

1.3 Subdivision with surfaces

2 Subdivision element count

	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. The points do not show any regularity.

With each subdivision, four new triangles are assigned to a triangle, but a point can also have several triangles assigned to it.

3 Code Snippets and Curves

3.1 Chaikin's Algorithm and cubic interpolation

Cubic interpolation mask implementation:

```
// 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;
```

3.2 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, iterate through the vector containing the triangles and test 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 first, second and third point of triangle test if
            test1 = (tris[t].iv[0]==tris[tn].iv[0] || tris[t].iv[0]=
            test2 = (tris[t].iv[1]==tris[tn].iv[0] || tris[t].iv[1]=
            test3 = (tris[t].iv[2]==tris[tn].iv[0] || tris[t].iv[2]=
            // find t0!=t containing b and c (CG21_1 page 19)
            if(test2 && test3 && !test1){
                tris[t].it[0]=tn;
            }
            // find t1!=t containing c and a (CG21_1 page 19)
            if(test3 && test1 && !test2){
                tris[t].it[1]=tn;
            }
            // find t2!=t containing a and b (CG21_1 page 19)
            if(test1 && test2 && !test3){
                tris[t].it[2]=tn;
            }
        }
        // increase no. of triangles for the point of the triangle
        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 2 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 $mv[][]$ for the three edges stored, which are in each case the edge connecting points and which is 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 index of triangle is smaller than inde
            //otherwise the edge point is already calculated
            if(i<t.it[j])
            {
                //find Vertex in neigbor Triangle that is not in Tri
                test1 = t.iv[i1]==n.iv[0] || t.iv[i2]==n.iv[0];
                test2 = t.iv[i1]==n.iv[1] || t.iv[i2]==n.iv[1];
                test3 = t.iv[i1]==n.iv[2] || t.iv[i2]==n.iv[2];
                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
                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 ? and then beta with the formula ?. Beta is inserted into the formula 3 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[
   pts[tris[i].iv[1]] +=  ( ( (1.0 - beta_n(valences[tris[i].iv[
   pts[tris[i].iv[2]] +=  ( ( (1.0 - beta_n(valences[tris[i].iv[
}
```

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 triangles around the middle triangle
        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.

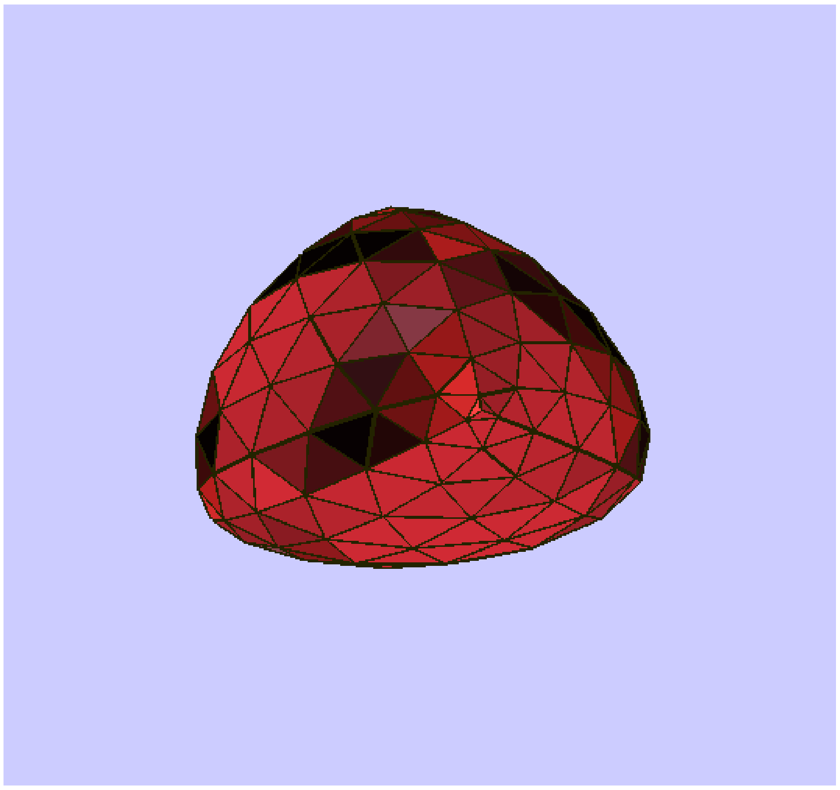

Figure 3:

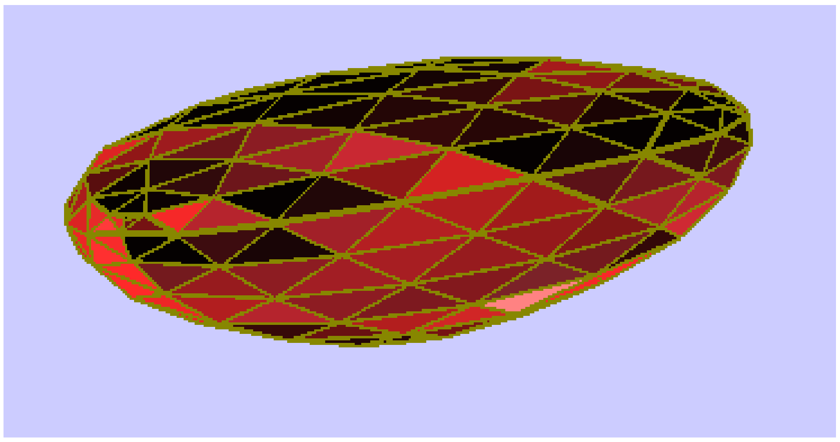

Figure 4: