Rendering and Manipulating Quad Meshes and Bezier Surfaces

Field of study: Media informatics
Semester: 4

Date: May 14, 2018

Authors:
Fabian Niehaus
Tuyet Nguyen

Contents

1 Abstract

In this project we will build a C++ application that can read, render and write quad meshes and apply the Catmull-Clark subdivision algorithm. It will also be able to read and compute Bezier surfaces and create rotational sweep surfaces from Bezier curves. We will use QT Creator for programming, QT for window management and OpenGL for 3D rendering.

2 Realisation

2.1 Creating an interface with QT

First off, we create a new QT widget application. This allows us to use QT Creators design feature to set up our application's interface. A new QOpenGLWidget is placed and will be used as a placeholder for a new custom class inheriting QOpenGLWidgets functionality. This class, called OGLWidget, needs to implement the following methods: initializeGL (for setting up OpenGL), paintGL (for doing the actual rendering), resizeGL (for handling resizes of the display window). Additionally, the functions SetMaterialColor and InitLightingAndProjection[1] are used.

2.2 Creating the data structure

The data structure is separated in different classes.
The basic class "Vertex" contains the three coordinates and the valence.
The basic class "Quad" contains the indices of four vertices, the indices of four adjacent quads, the index of its face vertex and the four indices of its edge vertices, the latter two being calculated in the process of the Catmull-Clark subdivision.
Logic and data regarding the computation of quad meshes is stored in a seperate class, as are Bezier surfaces and rotational sweep surfaces.
In order to allow for easier use of a two dimensional matrix of vertices, a wrapper class containing a two dimensional vector of vertices is introduces.

2.3 Reading the data from a file

2.3.1 Quad mesh

After creating the required data structure, a method for reading the vertices and faces from an .obj file is implemented. For this project we can assume that every face is a quad. The distinction between a vertex and a quad is made by the first letter in the line, a "v" for vertices and an "f" for quads.
Each line is read, and depending on the first letter a new Vertex or Quad is created and pushed to the respective vector.

2.3.2 Bezier surface

The initial surface data for Bezier calculations is stored in a data format that separates the x, y and z coordinates in different blocks. Each block is read and its values stored

[1]Taken from Prof. Dr. Martin Hering-Bertrams OpenGL_Example

in a temporary matrix. After alls blocks have been read, a matrix of vertices is created from this data. This matrix is then used to create quads. Vertices and Quads are stored for further use within the object.

2.4 Rendering Quad meshes

Depending on the desired way of rendering the object, different draw methods are implemented. These methods are then being called from the paintGL() function.

2.4.1 Rendering as a wireframe

At first, we want to render the object a wireframe without any specific lighting or normals. The method drawLines() implements this by going through the list of quads and connecting every two points that are next to each other, using GL_Lines. Every line needs two points as input, so every point is mentioned twice considering we need 4 lines per quad in total.

2.4.2 Rendering as a cube

After drawing the object as a wireframe we want to draw it as a solid cube with lighting. This is being achieved in the method drawQuads() which once again iterates over the list of quads. This time using GL_Quads, the four vertices of a quad are connected and the area inbetween is filled. The normal vector for this is calculated using the cross product of the two diagonals vectors.

2.5 Calculation the vertex valence

After making sure that the meshes are read and rendered correctly, the vertex valence is to be calculated. Vertex valence in this context means how many edges a single vertex is connected to. To calculate this, the following algorithm is used:

```
for each verxtex
  for each quad
    for each vertex of this quad
        if index of both vertices is equal
          increase valence of vertex by 1
```

2.6 Determining adjacent quads

In order to determine each quads adjacent quads (or neighbors for short), we need to check if it has a shared edge with another quad. This is calculated by comparing if they share two vertices. The algorithm looks as follows:

```
for each quad
  for i = 0 to 3 // since each quad has 4 vertices
    a = index of vertex at i
    b = index of vertex at (i+1)%4
    for each quad that is not the current one
      compare all vertices to a and b
      if a and b have a match
        add index of quad to current quads list of neighbors at index i
```

2.7 Printing the data

In order to verify the correctness of the results, all data is printed to the console. Additionally, each vertex (now with their valence) and quad (now with its neighbors) is written back to a .obj file.

2.8 Catmull-Clark-Subdivision

2.8.1 Calculating the Face-Mask

In the first step of the Catmull-Clark-Subdivision process, the so called Face-Mask is calculated. It describes the middle point of each quad. Mathematically speaking, it is the average of the four vertices of the quad:

$$\vec{f} = \frac{1}{4} * (\vec{v}_0 + \vec{v}_1 + \vec{v}_2 + \vec{v}_3)$$

This calculation is run for every quad. The resulting vertex is added to the list of vertices, and the index of it is stored within the quad.

2.8.2 Calculating the Edge-Masks

In the next step, the four edge masks are calculated. These vertices are the average between two end vertices of an edge and the face vertices of the two quads sharing this egde.
The formula for a single one of these points is:

$$\vec{e} = \frac{1}{4} * (\vec{v}_{\mathrm{a}} + \vec{v}_{\mathrm{b}} + \vec{f} + \vec{f}_{\mathrm{neighbor}})$$

This calculation is only run once per edge, meaning that only the quad with the lower index calculates the mask. The adjacent quad with the higher index only copies the reference. The following algorithm is used for the entire calculation.

```
for each i = 0 to (size of quads - 1)
  Q = quad at index i
  for j = 0 to 4
    N = adjacent quad at index j in array of neigbors of Q
    if index of q is smaller than index of N
      calculate the edge mask using the formula explained above
    else
      for k = 0 to 4
        if value at index k of Ns adjacent quads = i
          add Ns edge mask of index k to Qs edge masks at index j
```

2.8.3 Calculating the Vertex-Masks

The last mask calculated is the Vertex-Mask. With this mask every original vertex of a quad is repositioned using the following formula:

$$\vec{v}_{\mathrm{new}} = \frac{1}{(n_{\mathrm{v}})^2} * (2\vec{e}_{\mathrm{v_{left}}} + 2\vec{e}_{\mathrm{v_{right}}} - \vec{f}_{\mathrm{v}} + \vec{v}(n_{\mathrm{v}}(n_{\mathrm{v}} - 3)))$$

The implementation is split into the three following steps:

1.) Multiply every corner vertex with a product of its valence: $\vec{v}_{\text{new}} = \vec{v}(n_{\text{v}}(n_{\text{v}} - 3)$

```
for each vertex
  if vertex is corner vertex
    recalculate the vertex using the above formula
```

2.) Add edge vertices and subtract face vertex from each corner vertex of each quad: $\vec{v}_{\text{new}} = \vec{v} + 2\vec{e}_{\text{v}_{\text{left}}} + 2\vec{e}_{\text{v}_{\text{right}}} - \vec{f}_{\text{v}}$

```
for each quad
  for each corner vertex of this quad
    recalculate the vertex using the above formula
```

3.) Divide each corner vertex by its valence squared: $\vec{v}_{\text{new}} = \vec{v} * \frac{1}{(n_{\text{V}})^2}$

```
for each vertex
  if vertex is corner vertex
    recalculate the vertex using the above formula
```

2.8.4 Reconnecting the mesh

In the final step of the Catmull-Clark-Subdivision every quad is divided into 4 smaller quads. These smaller quads are added to a newvector. After all quads have been subdivided, the original vector storing the quads is overwritten with the new vector storing four times the amout of quads.

```
for each quad
  for j = 0 to 4
    v0 = corner vertex of the quad at index j
    v1 = edge vertex of the quad at index j
    v2 = face vertex of the quad
    v3 = edge vertex of the quad at index (j-1)%4 //tuned to always be
   positive
    create new quad with corner vertices v0 - v4 and store it
```

2.9 Bezier surface calculation

In this section, Bezier surface calculation is applied to the surface created when reading the bezier surface. The calculation for a matrix of dimension mxn uses the following forumla

$$f(s,t) = \sum_{i=0}^{m}\sum_{j=0}^{n} b_ij * B_i^m(s) * B_j^n(t)$$

where s and t are defined within [0;1].

B descibes the so called Bernstein polynomial $B_i^n(t) = \binom{n}{i} * t^i * (1-t)^{n-i}$.

A resolution describing the amount of new vertices created is introduced. In this case we chose a value of 0.05 meaning 21x21 new points are calculated.

```
for s = 0 -> 1; increased by resolution {
  for t = 0 -> 1; increased by resolution

    Vertex v = (0, 0, 0)

    for i = 0 -> m {
      for j = 0 -> n

        multiply vector at (i|j) with Bernstein
        polynomials of (i,m,s) and (j,n,t)

        add resulting vector to v
    }

    store v in new matrix at position (s * resolution | t * resolution)
```

2.10 Rotational sweep surface calculation

In this section, a rotational sweep surface is calculated from a single bicubic curve.

2.10.1 Cubic bezier curve calculation

First, Bezier transformation is applied to the two dimensional curve using the formula below:

$$p(t) = (1-t)^3 * p_1 + 3t * (1-t)^2 * p_2 + 3t^2 * (1-t) * p_3 + t^3 * p_4$$

$$t = [0; 1]$$

A resolution is of 0.05 is used for t, resulting in 21 new points.

```
for t = 0 -> 1; increased by resolution

  calculate and store new point according to
  the formula above
```

2.10.2 Rotational sweep surface creation

Using the Bezier curve calculated in the last step, a new surface is created. Since the curve is in two dimensional space we need translate every point into 3D space using the following logic:

$$x_{new} = x$$

$$y_{new} = y * sin(t * \pi)$$

$$z_{new} = y * cos(t * \pi)$$

$$t = [-1; 1]$$

Again, a resoultion of 0.05 is used resulting in a total of 21x21 new vertices.

```
for t = 0 -> 1; increased by resolution
  for i = 0 -> 20 // since there are 21 points

    calculate new vertex according to the formula above
    using the vertex at index i as input

    store new vertex in matrix at position (t * resolution | i)
```

After calculating the vertices for the new surface, quads are also created in order to draw it using drawQuad().

```
for i = 0 -> 20 - 1; // since we are also accessing i + 1 and j + 1
  for j = 0 -> 20 - 1;

    create new Quad from (
      vertex at (i|j)
      vertex at (i|j+1)
      vertex at (i+1|j+1)
      vertex at (i+1|j)
    )

    store new Quad
```

3 Results

3.1 Number of vertices, faces (quads) and edges after subdivision

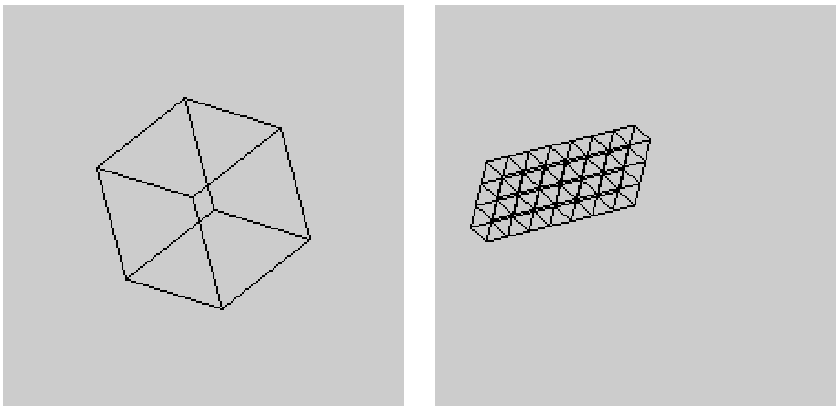
Data used: cube.obj

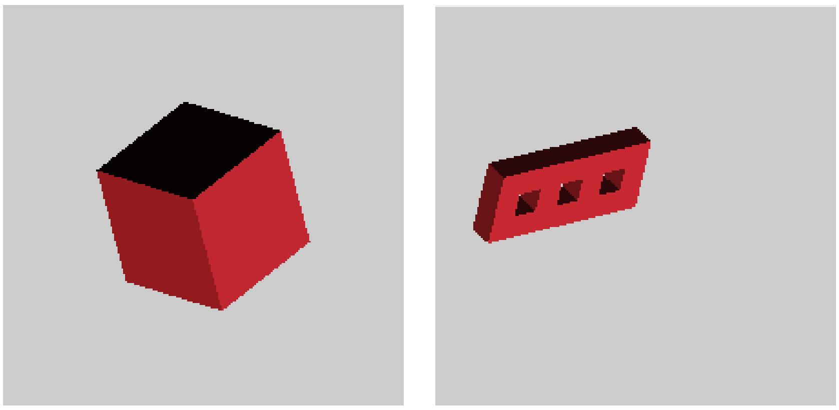
Subdivisions	Vertices	Faces	Edges
0	8	6	12
1	32	24	48
2	128	96	192
3	512	384	768

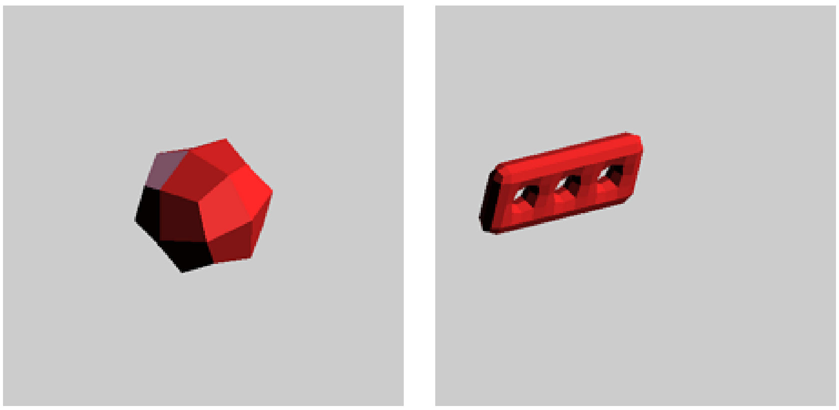
Data used: threehole.obj

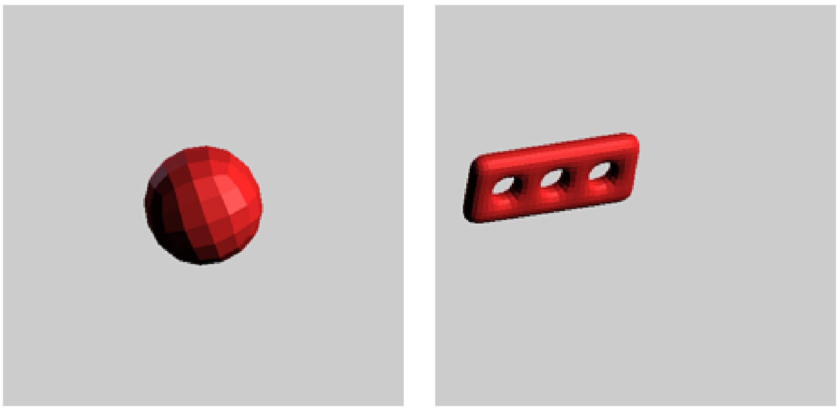
Subdivisions	Vertices	Faces	Edges
0	64	68	136
1	268	272	536
2	1084	1088	2176
3	4348	4348	8696

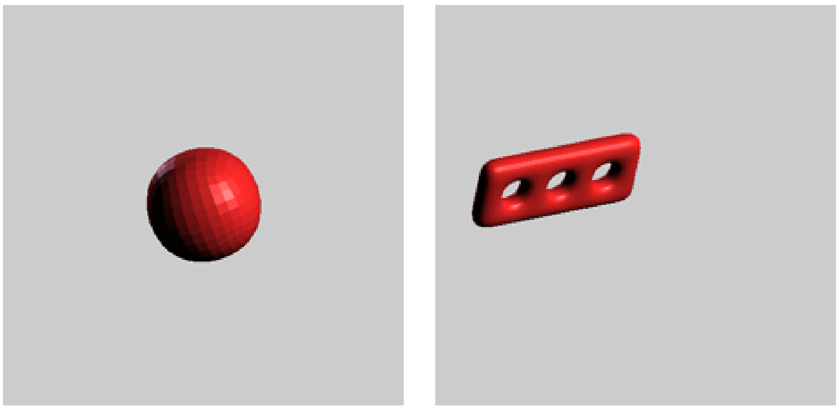
3.2 Rendering and subdivision


Wireframe Cube and Threeholes


Solid Cube and Threeholes


1 CC-Subdivision Cube and Threeholes


2 CC-Subdivisions Cube and Threeholes



3 CC-Subdivisions Cube and Threeholes

4 Conclusion

We were able to successfully build an application that can read and write any kind of boundary-free quad meshes, render them and perform Catmull-Clark-Subdivision on them. We gained a deeper understanding of how OpenGL and QT work and of the mathematics behing the specific subdivision concept. We will be able to extend the code for further applications and to reuse parts of it for other purposes e.g. calculation of Bezier surfaces.

5 Acknowledgements

- User Martin B on StackOverflow for supplying the function for always positive modulos in C++
- The QT Company, GitHub, the developers of TeXStudio and the developers of MiKTEX for letting us use their software free of charge
- Any of the poor souls that had to discuss the maths and structure of this project with us
- Gut & Guenstig Coffee for the long nights of debugging

6 References

- Prof. Dr. Martin Hering-Bertram, Lecture CG18_1, HSB, 2018