Half-Edge Mesh Data Structure

— Selected Topics in Modelling and Animation —

Erik Sven Vasconcelos Jansson
<erija578@student.liu.se>
Linköping University, Sweden

March 7, 2018

Abstract

Sacrificing *storage space* for *traversal flexibility* can be achieved by using a *half-edge data structure*. In this technical report the half-edge is described and implemented, along with several useful algorithms: how to traverse *neighboring vertices and faces* using half-edges, and how to calculate the *vertex normal*. Additionally, we describe how the *mesh area, volume, curvature, shell and genus* can be calculated.

Contents

References

[1] C. H. Lee, A. Varshney, and D. W. Jacobs. Mesh saliency. In *ACM transactions on graphics (TOG)*, volume 24, pages 659–666. ACM, 2005.

[2] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science Journal*, 7(2):217–236, 1978.

[3] C.-K. Shene. Computing with geometry, 1998.

1 Problem and Motivation

Polygonal meshes are usually represented by using sets of *vertices*, *edges* and *faces*. For *rendering* the mesh it's usually enough with a *"polygon soup"*, in which little, or no, *connectivity* is explicitly stored.

However, when attempting to *manipulate* or find certain *properties* of the mesh then it's increasingly inefficient. Finding *neighboring vertices* requires a linear traversal in edges for each vertex, $\mathcal{O}(|V| \cdot |E|)$.

Sacrificing *storage space* for *traversal flexibility* can be achieved by using a *half-edge data structure*. It provides $\mathcal{O}(1)$ access to vertices, edges and face for a triangle, and convenient access to its neighbors.

Decreasing complexity by an order of magnitude allows faster *inspection* and *manipulation* of meshes. This report describes several useful algorithms for finding the properties of a mesh by using half-edges.

2 Background and Theory

The **half-edge data structure** was introduced by *Muller and Preparata* [2] in 1978 to make traversal through a mesh more time efficient and convenient. It works by maintaining the *next* and *previous* edges for a given edge of a triangle, giving full edge/vertex access inside a triangle. For traversing to another triangle, there is the *pair* edge, which is "parallel" to the current edge, since it has been split in two. See Figure 1 for a graphical representation of these.

Finding **neighboring vertices and faces** to a vertex is a useful operation, and is used in many of the algorithms described here. By using a half-edge of $v_i \to e(v_i)$, we can find the next edge connected to v_i with $\text{pair}(\text{next}(\text{next}(e(v_i))))$ giving vertex v_j. All neighbor elements of v_i are called the *one-ring*.

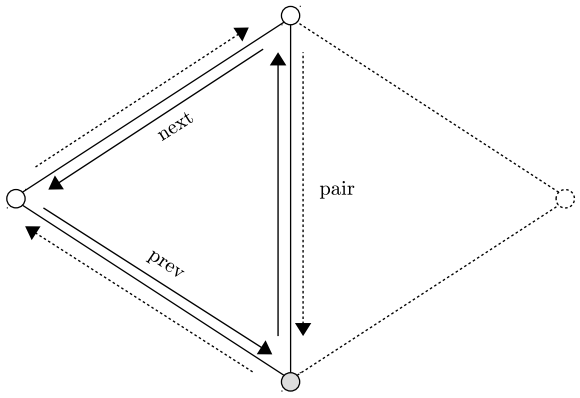


Figure 1: a half-edge data structure in a nutshell. All of the above data are relative to the gray vertex. Only the non-dashed elements are accessed directly; the rest of the mesh can be traversed using the *pairs*.

Estimating the **normal of a vertex** is essential when one renders using *Gouraud* or *Phong shading*. Doing this is quite straightforward, the normal of a vertex v_i is the sum of the *face normals* in the one-ring of v_i, thereafter normalized accordingly as:

$$\hat{n}(v_i) = \sum_{f \in N_i^f} f_{\hat{n}} \div || \sum_{f \in N_i^f} f_{\hat{n}} ||\,,$$

where $f_{\hat{n}}$ is the unit normal of the face f in the N_i^f. See Figure 2 for how to calculate $f_{\hat{n}}$ for an given f.

Another handy property is the **area of a mesh**, which is useful in itself and used in other algorithms. Since our mesh is piecewise flat, and we only want an approximation, we can use a *Riemann summation*:

$$A = \int_S dA \approx \sum_{f \in F} A_f = \frac{1}{2} || \sum_{f \in F} \vec{u}_f \times \vec{v}_f ||\,,$$

where $\vec{u}_f$ and $\vec{v}_f$ are two vectors which connect to the same vertex v_i. The length $||\vec{u}_f \times \vec{v}_f||$ is the *area of the parallelogram*, of which half is the triangle f.

It's trickier to calculate the **volume of a mesh**, but assuming the mesh is a *closed manifold*, we use the *divergence theorem* which relate area to volume:

$$\begin{aligned} 3V &= \int_V \nabla \cdot \mathbf{F}\, d\tau = \int_S \mathbf{F} \cdot \hat{n}\, dA \\ &\approx \sum_{f \in F} \frac{(\vec{v}_1 + \vec{v}_2 + \vec{v}_3)}{3} \cdot f_{\hat{n}} A_f\,, \end{aligned}$$

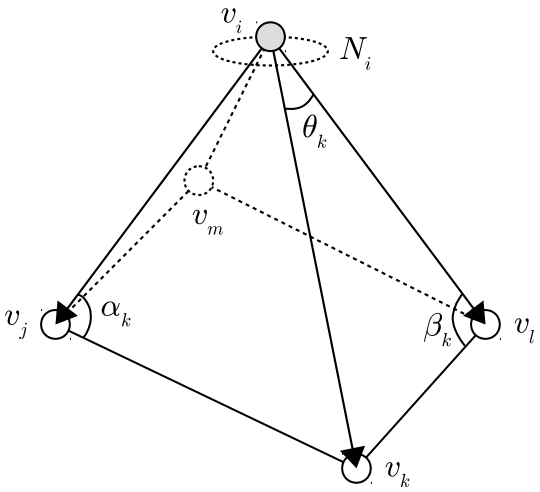


Figure 2: depiction of the *one-ring neighborhood* of v_i containing $N_i^v = \{v_j, v_k, v_l, v_m\}$. Normals of each triangle f are calculated by forming vectors from v_i for each $v_k, v_l \in N_i^v$ in it, and thereafter: $\overrightarrow{v_i v_k} \times \overrightarrow{v_i v_l}$.

where the vector field $\mathbf{F} = (x, y, z)$, and the volume is approximated by using a Riemann sum over faces.

An important property is the **mesh curvature**, which has many applications, e.g. *mesh saliency* [1] measure, used for finding the *importance of a region*. There are two measures of curvature in 3-D space, a **mean curvature**, and the **Gaussian curvature**:

$$H = (\kappa_1 + \kappa_2) \div 2 \;\text{ and the }\; K = \kappa_1 \kappa_2\,,$$

where κ_1 and κ_2 are the *principal curvatures* at P. Each κ_i describes the *smoothness* at P, and is inversely proportional to an *osculating circle's radius*. We express H and K in terms of angles around N_i^v:

$$K = \frac{1}{A}\left(2\pi - \sum_{k \in N_i^v} \theta_k\right),$$

where θ_k is in radians and adjacent to $\overrightarrow{v_i v_k}$, $v_k \in N_i^v$.

$$H\hat{n} = \frac{1}{4A_{N_i^f}} \sum_{k \in N_i^v} (\cot \alpha_k + \cot \beta_k) \overrightarrow{v_i v_k}$$

Lastly, finding the number of **shells** and **genus** of a mesh is done using Euler-Poincaré's characteristic:

$$G = (-V + E - F + 2S) \div 2,$$

assuming $S = 1$ then $G = (-V + E - F) \div 2$. For further details on genus and topology, see *Shene's* [3].

3 Method and Results

Implementing said data structures and algorithms is quite straightforward, but certain parts are still unclear, e.g.: how do we find the number of shells?

Here follows shortly how these were implemented, the generated results, and some discussion on them.

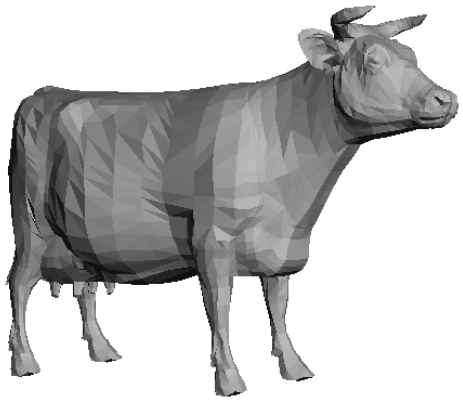
3.1 Creating a Half-Edge Mesh

Represented in `HalfEdgeMesh` using three lists: the *half-edge list*, *vertex list* and also a *face list*. It stores *next*, *previous*, *pair* pointers in each half-edge, and keeps a half-edge pointer in each vertex or face.

Adding a new face is done with `AddFace`, which takes the three vertices which compose one triangle. First, the three vertices are added to the vertex list, if they don't already exist. Thereafter we create the half-edges for these vertices, and remember to connect the "inner triangle" in a counter-clockwise fashion as shown in Figure 1. What about the outer edges you might wonder? The trick is to realise that these will eventually be connected later, assuming we have a closed mesh (the only case we will handle).

Finally, the vertices and faces are given pointers to their half-edges, since we usually specify algorithms which work in vertices or faces, rarely edges. Another property which should be initialized when adding a face is the *face's normal*. This is done simply by using the three vertices which were given, v_1, v_2, v_3, and then just calculating $||\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}||$.

After some fiddling, the complete mesh should be initialized, and fully connected both globally and locally (via the next, prev and pair pointers). When one renders it, in the same way as a "polygon soup", the reader should have a nice looking polygon cow:



3.2 Accessing Neighboring Faces

Now that the mesh is easier to traverse by using a half-edge data structure, the elemental operation of finding all triangles connected to a vertex is desired. We do this by calling `FindNeighboringFaces`, which, given a vertex v_i, returns an ordered list of faces $(f_1, f_2, ..., f_n)$ which share the same vertex v_i.

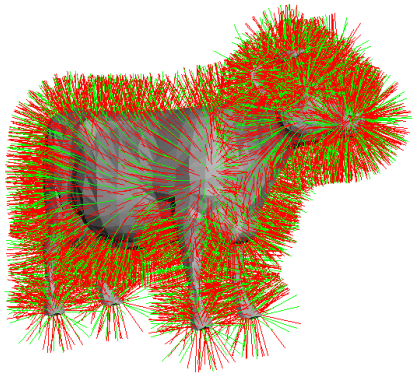
Since we know an initial edge $e(v_i)$ which has v_i, and each half-edge stores a pointer to its face, the base-case is trivial: $f_1 = \text{face}(e(v_i))$. In general, we need to find the next half-edge which has an unique face by traversing the one-ring around v_i. The next edge e_k is found with: $e_k = \text{pair}(\text{next}(\text{next}(e(v_k))))$ which will always give the next face $f_k = \text{face}(e_k)$. Append f_k to the list until we have looped around the one-ring, in other words, when a face $f_{k+1} = f_1$.

3.3 Finding the Vertex's Normal

Our first application of these neighboring faces is to find the normal of a vertex. Since we know that the faces $f_1, f_2, ..., f_n$ are around the vertex v_i, and that each face f_k has a face normal $\hat{n}(f_k)$, which was pre-computed by taking $||\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}||$ of the vertices for f_k, we get the mean normal around v_i:

$$\hat{n}(v_i) = \sum_{f_k \in N_i^f} \hat{n}(f_k) \div || \sum_{f_k \in N_i^f} \hat{n}(f_k)||$$

Applying this to our polygon cow for demonstration purposes, we get the piñata polygon cow below. In this case, the red lines are the face normals and the green lines our calculated vertex normals. It's important to note that the vertex normal is heavily dependent on the placement of the triangles, e.g. a cube will gravitate towards a certain side if uneven.



3.4 Calculating Area of Mesh

As previously described, we calculate the area of a mesh by summing up the individual piece-wise flat areas (triangles in this case). How is the area A_f of a face f calculated, given it's composed of triangles? We already have the list of faces F, and therefore can pick one $f_i \in F$. We can also find the three vertices associated with the face f_i by using the half-edges $e_i = \text{edge}(f_i)$, $\text{next}(e_i)$ and $\text{prev}(e_i)$, then accessing $\text{vert}(e_k)$ for every edge: $\{v_1, v_2, v_2\}$.

Finally, we find the area of the parallelogram with $||\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}||$, done for every face $f_i \in F$. Since the parallelogram are two triangles stitched together, a triangle f_i has then a area $A_{f_i} = ||\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}|| \div 2$. The mesh area A is therefore a sum of $A_{f_i}, \forall f_i \in F$.

Trying this out on 3 spheres composed of triangles of different radii, we get the results below. We see the difference between the analytical one isn't large:

Radius	Analytical Area	Area of Mesh
0.1	0.12566	0.12511
0.5	3.14159	3.1277
1.0	12.5663	12.5111

3.5 Approximating Volume

Similarly, we want to approximate the mesh volume. According to the theory presented earlier, we need to find the centroid of the face, which is simply the three vertices $\{v_1, v_2, v_3\}$ for each face f_i. Then, the centroid is: $(v_1 + v_2 + v_3) \div 3$. We already know how to find the area A_{f_i} of a triangle face f_i, and also the face normal $\hat{n}(f_i)$, see the previous definitions.

Computing the volume is then trivial, we simply calculate $(v_1 + v_2 + v_3 \div 3) \cdot A_{f_i}\hat{n}(f_i)$ for each face $f_i \in F$, then taking the sum of these. Note however that the volume we calculated is $3V$, and a division by 3 is required at the end to get the correct result.

Again, trying this out on a sphere with different radii, we get the results found in the table below. Notice again that the approximated error isn't large.

Radius	Real Volume	Volume of Mesh
0.1	0.00419	0.00415
0.5	0.5236	0.51899
1.0	4.1887	4.15192

3.6 The Mesh Curvature

Since there are two types of curvatures in 3-D, we describe how to implement each of these, and thereafter proceed to analyze their practical differences. We start by describing the Gaussian curvature first.

Algorithmically, the Gaussian curvature for a v_i vertex is the difference between 2π and the sum of all angles between edges in the neighborhood of v_i, times the inverse of the area, $1 \div A$, as seen before. An angle $\theta_k = \cos^{-1} \hat{v} \cdot \hat{u}$ between $\overrightarrow{v_iv_k}$ and $\overrightarrow{v_iv_{k+1}}$, where v_{k+1} is the next counter-clockwise vertex in the neighborhood of v_i after v_k, gives one such angle.

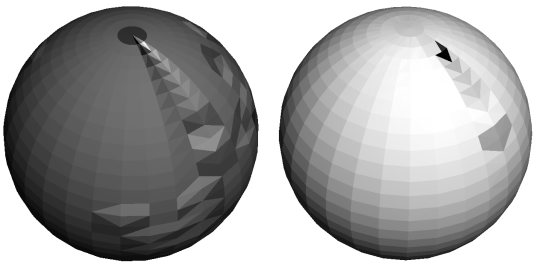
Visualizing the Gaussian curvature can be done by mapping the interval to a black-white gradient, and coloring each vertex or face in its respective curvature map. We have done this for the left figure below, with a sphere of unit radius. Analytically, we should have the same color across the surface since the curvature is inversely proportional to the osculating sphere's radius. We see that this isn't the case, at least not perfectly. As it turns out, the way triangles are placed will affect the curvature, and the Gaussian curvature is more easily swayed because it multiplies principal curvatures κ_1 & κ_2.

Now, we attempt to find mean curvature as well. Since *Voronoi area* provides a better approximation:

$$A_{N_i^f} = \frac{1}{8} \sum_{k \in N_i^v} (\cot \alpha_k + \cot \beta_k) ||\overrightarrow{v_iv_k}||^2 \ ,$$

we use it instead of the previous area approximation. Calculating these angles is similar to how we handled the Gaussian curvature, but instead of only picking the next edge we also pick the previous edge. Finally, calculating the summation for the mean curvature approximation uses basically the same techniques as the Voronoi area, calculating it as before.

Comparing the results, the mean curvature (to the right) produces less noisy/varied curvature. It is more in line with the expected analytical results.

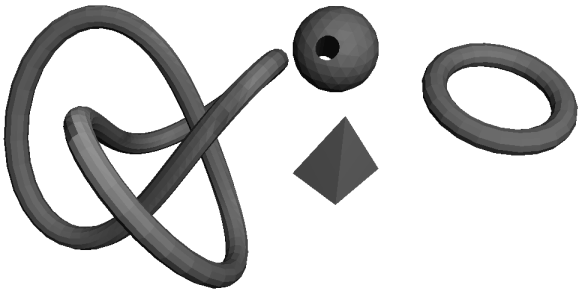


3.7 Shell and Genus

By using the Euler-Poincaré characteristic we can find out an unknown property if we know the rest. Since we already have the number of vertices, edges and faces the remaining ones are shells and genus. Since finding the genus is problematic, we attempt to find the number of shells first, and then plug it into Euler-Poincaré. We describe how to find shells.

Assemble a set V_S which is initially $V_S = V$ and pick some $v_i \in V_S$. Find the neighbors N_i^v of v_i and remove them from V_S but add them to a queue V_Q. Pick next v_i such that $v_i \in V_Q$ and repeat as above, until the queue $V_Q = \varnothing$, then pick another $v_i \in V_S$. Each time we pick new v_i from V_S, it's a new shell. Therefore, in a nutshell, we pick a random vertex and find all possible interconnected neighbors to it by marking vertices as traversed. This should lead to all vertices being traversed, if the mesh is a closed manifold, and the number of shells is one. Each time this doesn't happen, and the meshes are closed manifolds, means that there is more than one shell, and we mark a new shell as being one that isn't completely interconnected (i.e. vertices traversed).

See the example scenario below, our algorithm gets us $S = 4$, and we know V, E, F, therefore we can determine the genus G with Euler-Poincaré. If you are wondering why the $G = 3$, see *Shene's* [3].



In many cases the genus of a mesh can be seen as the amount of "holes" in a mesh. For example, the torus in the scene above has genus one and the pyramid has genus zero. However, this rule of thumb doesn't apply in all cases, for example, the sphere with seemingly "two holes" actually has a genus one.

Vertices	Edges	Faces	Shells	Genus
2013	12066	4022	4	3

Grading

According to the lab guidelines within the course `TNM079`, I've implemented features for: 3, 4 and 5. My aim with this assignment is a grade of scale: 5.

Mesh Simplification Algorithms

— Selected Topics in Modelling and Animation —

Erik Sven Vasconcelos Jansson
<erija578@student.liu.se>
Linköping University, Sweden

June 3, 2017

Abstract

Surfaces are usually described using polygon meshes in computer graphics. Some surfaces may require a staggering amount of triangles to describe in detail. A *mesh simplification* algorithm removes polygons from a mesh while trying to keep its visual details. In this technical report we summarize and compare the *quadric-based* and *absolute curvature-weighted error metrics*, which are used in mesh simplification.

Contents

References

[1] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Conference proceedings on computer graphics and interaction*, pages 209–216. Addison-Wesley, 1997.

[2] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Conference proceedings on computer graphics & interactive techniques*, pages 19–26. ACM, 1993.

[3] L. Li, M. He, and P. Wang. Mesh simplification based on absolute curvature-weighted quadric error metrics. In *Conference on Ind. Electronics and Applications*, pages 399–403. IEEE P, 2010.

1 Problem and Motivation

Representing surfaces using polygonal meshes still remains dominant in the field of computer graphics, since it's flexible (e.g. for artists) and fast on GPUs.

However, the increasing demand for high-fidelity models (e.g. in computer games or 3-D animations) has caused the number of triangles to skyrocket in recent years. While rendering such highly detailed models is possible (even in real-time), it's a waste of resources if these details are superfluous, and don't impact the final visualization (i.e. render) of a scene.

Instead of re-creating a lower resolution mesh by hand, we specify a mesh using their full resolution, and let a *mesh simplification/decimation* algorithm reduce the number of triangles for us, up and until a given threshold. Here we describe such algorithms.

2 Background and Theory

Briefly, *mesh simplification* attempts to, given some mesh $\mathcal{M} = (\mathcal{V}, \mathcal{F})$ with vertices $\mathcal{V}$ and triangles $\mathcal{F}$, find some new mesh $\mathcal{M}' = (\mathcal{V}', \mathcal{F}')$ such that either:

1. $||\mathcal{M}' - \mathcal{M}||$ is minimal, given a target $|\mathcal{V}'| = n$;
2. $|\mathcal{V}'|$ is minimal, given threshold $||\mathcal{M}' - \mathcal{M}|| < \epsilon$,

where $||\mathcal{M}' - \mathcal{M}||$ measures an *approximation error*. Thus, we attempt to reduce $\mathcal{M}$ into $\mathcal{M}'$ by editing $\mathcal{V} \rightarrow_A \mathcal{V}'$ and $\mathcal{F} \rightarrow_A \mathcal{F}'$ with our algorithm A, until we either reach an *error threshold* ϵ, alternatively a *vertex threshold* n, given as parameters to A. Both of these can be used as a stopping condition for A.

In this technical report we'll deal only with (1), and concentrate on *incremental decimation*, which give $\mathcal{M}'_s$'s with reduced vertices in each iteration s.

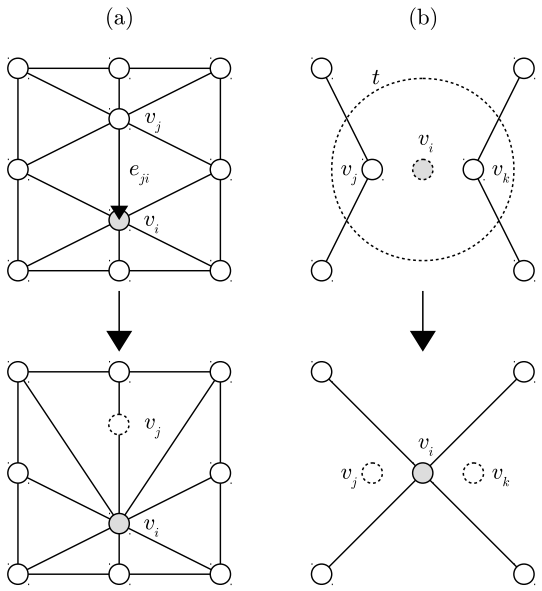


Figure 1: (a) edge contraction & (b) pair contraction

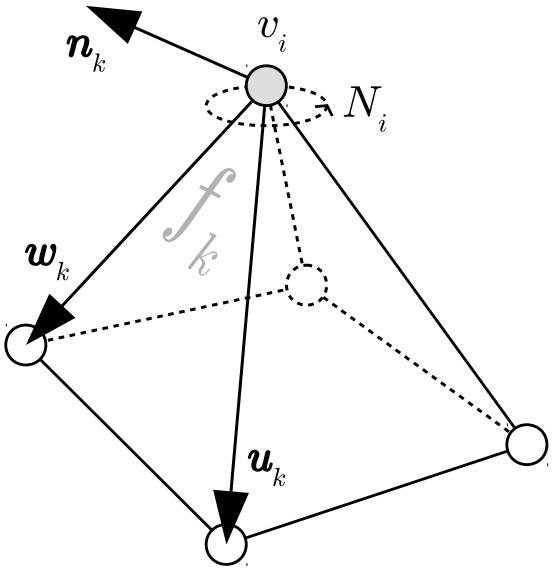


Figure 2: depicts a plane f_k with normal $\mathbf{n}_k$ for v_i

Since there are several such algorithms, we'll only follow the method due to *Garland and Heckbert* [1], called *quadric-based mesh decimation*. But we'll do some variations of it too for comparison purposes, one which will perform worse (i.e. $||\mathcal{M}' - \mathcal{M}||$ will be large), and another which "preserves" curvature.

Before describing the entire algorithm, we'll show the general intuition required. Since we are dealing with an iterative algorithm, the mesh $\mathcal{M}'_s$ at step s of A will only have been a stepping stone towards the final mesh $\mathcal{M}'$. We reach $\mathcal{M}'$ when the stopping condition, $|\mathcal{V}'| = n$, holds for some iteration s in A. Assuming we have a $\mathcal{M}$ with $|\mathcal{V}| > n$, the algorithm needs to *remove vertices* from $\mathcal{M}$ to satisfy $|\mathcal{V}'| = n$. Essentially, each iteration s our algorithm needs to remove one vertex from a previous $\mathcal{M}'_{s-1}$, and given that $\mathcal{M}'_0 = \mathcal{M}$, mesh decimation $\mathcal{M}'_n = \mathcal{M}'$ follows. We choose $\mathcal{M}'_s$ such that $||\mathcal{M}'_s - \mathcal{M}||$ is minimized.

Garland and Heckbert [1] describe two methods for removing one vertex in each iteration. First is *edge contraction*, introduced in *Hoppe et al.* [2]; by choosing an appropriate edge $e_{ji} = (v_j, v_i)$ we can collapse $(v_j, v_i) \to \bar{v}_i$ by deleting triangles and reconnecting v_j's edges to $\bar{v}_i$, shown in Figure 1 (a). They also present *pair contractions*, in Figure 1 (b), which allows a collapse $(v_j, v_i) \to \bar{v}_i$ even if they aren't connected by an edge, but only limited by t, which constrains the collapse to only $||v_j - v_i|| < t$.

We'll be using only edge contractions since it's easier to implement, but *Garland and Heckbert* [1] use pair contractions, which means results may vary.

Now, how does one pick a $(v_j, v_i) \to \bar{v}_i$ such that $||\mathcal{M}'_s - \mathcal{M}||$ is minimal in each step? In their paper, we assign a matrix $\mathbf{Q}_i$ for each v_i which is used to measure the error $\Delta(\mathbf{v}) = \mathbf{v}^\intercal \mathbf{Q}_i \mathbf{v}$ of v_i being moved to its new position $\mathbf{v}$. The optimal position $\bar{\mathbf{v}}$ for a contraction can be found by solving $\nabla\Delta = \mathbf{0}$ and is linear since $\Delta(\mathbf{v})$ is quadratic. Shown in the paper:

$$\bar{\mathbf{v}} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

which can be used to find the *cost* of contracting the edge $(v_j, v_i) \to \bar{v}_i$ and is $\Delta(\bar{\mathbf{v}}) = \bar{\mathbf{v}}^\intercal (\mathbf{Q}_i + \mathbf{Q}_j) \bar{\mathbf{v}}$. We simply put these costs in a min-heap, and each iteration s we contract the edge on top of the heap.

Lastly, calculating $\mathbf{Q}_i$ is done by taking the sum of quadrics derived from planes which intersect v_i. Finding these planes is simple: we take the faces f_k in the neighborhood N_i of v_i which has normal $\mathbf{n}_k$. Therefore $\mathbf{Q}_i = \sum_{f_k \in N_i} \mathbf{p}_k^\intercal \mathbf{p}_k$, $\mathbf{p}_k = \begin{bmatrix} \mathbf{n}_k & \mathbf{n}_k \cdot v_i \end{bmatrix}$. Hence, we can interpret $\Delta(\mathbf{v})$ as the distance from the planes $\mathbf{p}_k$ which intersect v_i, to the position $\mathbf{v}$.

To make things a bit more exciting, we present two other $\Delta(\mathbf{v})$ metrics and heuristics for deciding the position $\bar{\mathbf{v}}$. These will be used for comparing and evaluating quadric-based error metrics, seeing which situations they perform well, and not so much.

Also shown in *Garland and Heckbert* [1] we have the simple scheme where $\bar{\mathbf{v}} = (v_i + v_j) \div 2$, that is, we contract $(v_i, v_j) \to \bar{v}_i$, such that it's placed at $\bar{\mathbf{v}}$, in-between the vertices v_i and v_j. It also allows for a very simple cost function which is: $\Delta(\bar{\mathbf{v}}) = ||\bar{\mathbf{v}} - \mathbf{v}_i||$. Intuitively, this *should* perform worse than quadrics.

Finally, we present an *absolute curvature-weighted* error metric, such as the one shown in *Li et al.* [3], which builds upon the *quadric-based* error metrics. Since humans find high curvature areas on a surface important (see e.g. *mesh saliency*), as characteristic features of the mesh, we should attempt to penalize removal of vertices which may change the curvature. We already know from the previous technical report how to find the *mean curvature* H and the *Gaussian curvature* K for an arbitrary closed manifold mesh. According to *Li et al.* [3], we can find the *principal curvatures* κ_1 and κ_2 of some vertex v_i using H, K:

$$\kappa_1 = H + \sqrt{H^2 - K} \; ; \; \kappa_2 = H - \sqrt{H^2 - K} \; ,$$

where the *absolute curvature* will be $\kappa = |\kappa_1| + |\kappa_2|$. We then simply use this κ for adding weights for Δ, where we redefine $\Delta(\mathbf{v}) = \kappa(\mathbf{v}^\intercal \mathbf{Q}_i \mathbf{v})$. According to the results from *Li et al.* [3], while the error will be larger for *absolute curvature-weighted* compared to *quadric-based* metrics, it preserves curvature better.

3 Method and Results

Let's cover the practical details now, on how to build a mesh simplification algorithm by using the metrics. We'll evaluate each result and compare them to each other using our trusty polygon cow, which shall be decimated in the name of science (sounds painful).

See the appendix later for images of these results.

3.1 Outlining Mesh Simplification

Briefly speaking an algorithm has these major steps: (1) select all *valid edges* for contraction, (2) compute the *contraction cost and position* for each such edge, (3) insert these into a *min-heap* by *contraction cost*, and finally, (4) pick the least-cost contraction in it, and apply an *edge contraction* into *collapse position*. Rinse and repeat until a *stopping condition* is true.

Simply using $\Delta(\bar{\mathbf{v}}) = ||\bar{\mathbf{v}} - \mathbf{v}||$, $\bar{\mathbf{v}} = (v_i + v_j) \div 2$ gives us a simple decimation scheme, which has an expected "wishy-washy" outcome, seen in Figure 3. However, we use this "skeleton" with better metrics.

3.2 Quadrics-Based Error Metrics

Afterwards, we describe how *quadric error metrics* were integrated into this general algorithm outline.

For computing the *contraction cost* $\Delta(\bar{\mathbf{v}})$ and the *contraction position* $\bar{\mathbf{v}}$, we need the quadric $\mathbf{Q}_i$ for each vertex v_i before doing anything, under $s = 0$. Looping through each vertex v_i in our vertex list $\mathcal{V}$, we find all planes $\mathbf{p}_k$ in the neighborhood N_i of v_i. Since the face $f_k \in N_i$ is on the plane $\mathbf{p}_k$, we just get the normal $\mathbf{n}_k$ of f_k, which is also normal to $\mathbf{p}_k$. But there are infinite such planes with normal $\mathbf{n}_k$, we need to solve $d = -(ax + by + cz)$ for uniqueness. We do this by fixing $\mathbf{p}_k$ to some point on f_k, e.g. v_i.

After this $\mathbf{p}_k = \begin{bmatrix} a & b & c & d \end{bmatrix}$, and can find $\mathbf{p}_k^\intercal \mathbf{p}_k$. Since $\mathbf{Q}_i = \sum \mathbf{p}_k^\intercal \mathbf{p}_k$, we have found quadrics for v_i. Find the *contraction position* $\bar{\mathbf{v}}$ as done in Section 2. Then the *contraction cost* is $\Delta(\bar{\mathbf{v}}) = \bar{\mathbf{v}}^\intercal (\mathbf{Q}_i + \mathbf{Q}_j) \bar{\mathbf{v}}$, for contracting a *valid edge* $e_{ji} = (v_j, v_i) \to \bar{v}_i$ to $\bar{\mathbf{v}}$.

See Figure 4. Comparing it to Figure 3 we notice that quadric-based mesh simplification is better at keeping the meshes overall shape. Notice that legs have been "thinned" in Figure 3, but not in Figure 4. *Garland and Heckbert* [1] mention in their paper that the errors were reduced by up to 50% with quadrics.

3.3 Absolute Curvature-Weighted

Turns out to be quite a straightforward extension of quadric-based error metrics, as we only need to weigh-in the *absolute curvature* κ into a $\Delta(\mathbf{v})$ cost. Hence we only change $\Delta(\bar{\mathbf{v}}) = \kappa(\bar{\mathbf{v}}^\intercal (\mathbf{Q}_i + \mathbf{Q}_j) \bar{\mathbf{v}})$ in the quadrics-based approach, the rest is unchanged.

Looking at Figure 5 and comparing it to Figure 4, we notice some defining areas are evicted in Figure 4, but kept in Figure 5. For example, the eyes' details are still kept by using an absolute curvature metric, and Figure 6 visually demonstrates why this occurs. This metric is suitable when curvature is important, but doesn't guarantee $||\mathcal{M}' - \mathcal{M}||$ will be minimal.

Grading

According to the lab guidelines within the course `TNM079`, I've implemented features for a: 3 and 4. My aim with this assignment is a grade of scale: 4.

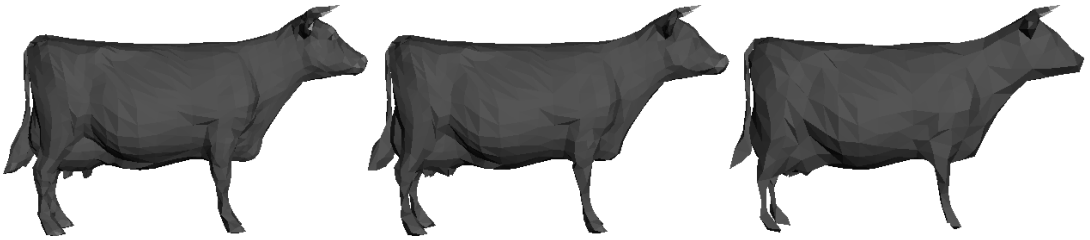


Figure 3: simple decimation for the cow mesh (50%, 35%, 17% triangles of the original mesh)

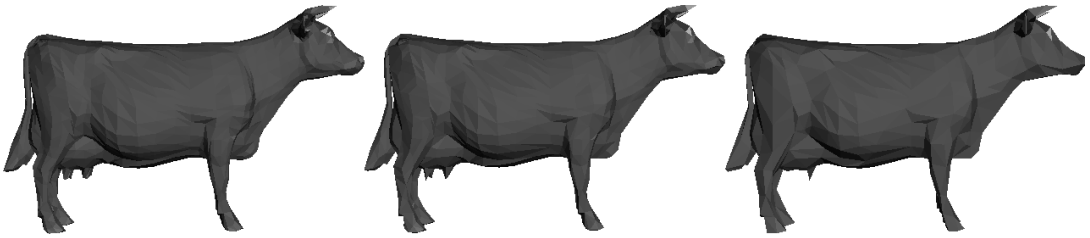


Figure 4: quadric-based decimation for the cow mesh (50%, 35%, 17% triangles of original)

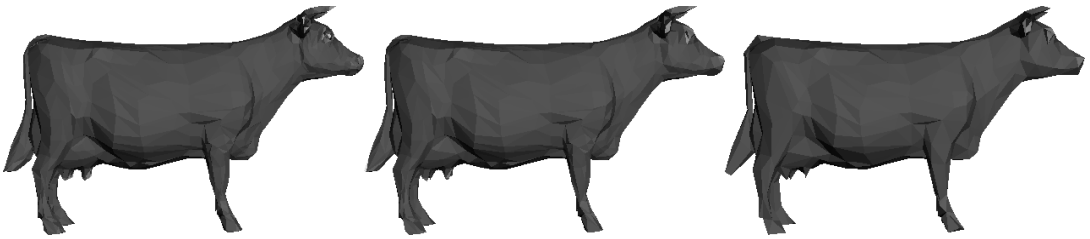


Figure 5: absolute curvature-weighted decimation for the cow mesh (50%, 35% and 17%)

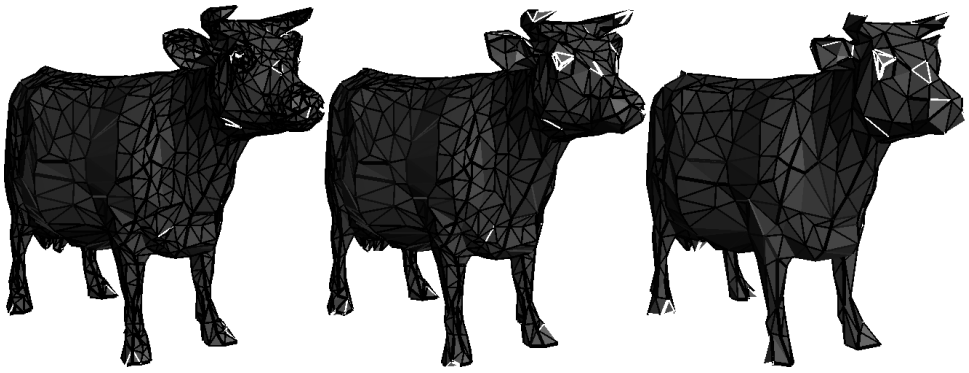


Figure 6: mapping of the cost for curvature-weighted decimation in each step

Splines & Subdivision Surfaces

— Selected Topics in Modelling and Animation —

Erik Sven Vasconcelos Jansson
<erija578@student.liu.se>
Linköping University, Sweden

March 7, 2018

Abstract

Parametric curves can describe smooth geometry and motion; important parts in computer graphics. However, using a high-degree polynomial is both computationally inefficient and a hassle to specify. Instead, we use splines, which are stitched together piecewise polynomial curves. In this article, we describe the *uniform cubic b-spline* and its properties. More specifically, we make use of its *local support* to speed up our b-spline evaluations. We then show how these can equivalently be represented by using *curve subdivision*, which iteratively makes the curve smoother. Finally, we generalize this concept towards *surface subdivision*, and present the well-known *Loop subdivision scheme* for triangle meshes.

Contents

References

[1] De Boor, Carl. *A Practical Guide to Splines*, volume 27. Springer-Verlag Pub. New York, 1978.

[2] Fisher, Matthew. Subdivison (Catmull-Clark vs Loop's Surface Subdivison with Details). 2014.

[3] Loop, Charles. Smooth Subdivision Surfaces Based on Triangles (*Loop's SS Algorithm*). 1987.

1 Problem and Motivation

Representing *smooth curves and surfaces* have been important achievements within computer graphics; since they can represent *geometry* (e.g. font glyphs), and even *motions* (e.g. animation paths), *smoothly*.

Unfortunately, using *high-degree polynomials* is not feasible, both *computationally* and also in terms of *versatility* (difficult for artists to control results). By using a *piecewise polynomial function*, the *spline*, we can model most *high-degree polynomials* using a set of *"stitched together" lower-degree polynomials*. Leading to low compute cost, and also *local control*, enabling artists to modify the curve's *control points*:

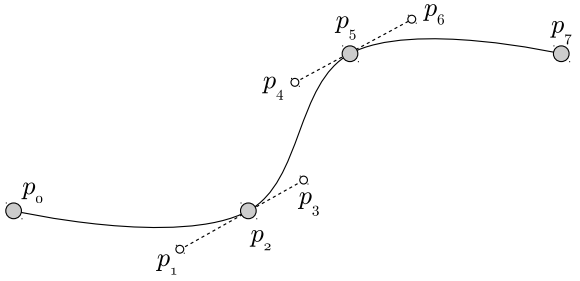


Figure 1: example quadratic Bézier spline curves.

Here we explain the *uniform cubic B-spline* only, and discuss how to evaluate it efficiently by using a *local support* property inherent to B-splines. Also, we show how these can equivalently be represented by *iterative curve subdivision*. Finally, we extend these concepts to *B-spline patches (i.e. surfaces)* by *iterative surface subdivision* using *Loop's algorithm*. Other splines (e.g. *Bézier curves*) and subdivision schemes (e.g. *Catmull-Clark*) won't be shown here.

2 Background and Theory

Here we define *B-splines* as sets of *control points* $\boldsymbol{p}_i$ weighted against the *basis functions* $B_{i,k}(x)$, which are measured at x *along the curve* with a *degree* n:

$$\boldsymbol{S}_n(x) = \sum_i B_{i,n}(x)\boldsymbol{p}_i.$$

Since we only deal with *uniform cubic B-splines*, the degree n is always 3, and the distance between the *knots* is the same everywhere. *Knots* are where the basis functions meet, and give C^2 *continuity* at those points. See Figure 2 for the plot over $B_{i,3}(x)$.

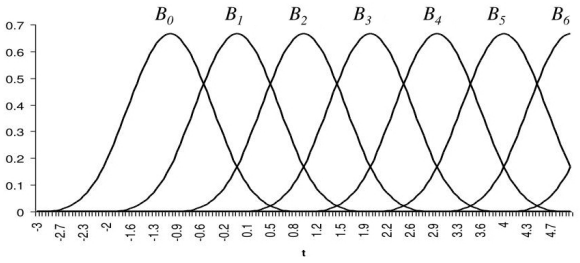


Figure 2: plot over some of the basis functions $B_{i,3}$, found in: *Stephen Chenney's CS559 Lecture Notes.*

Notice how $B_{i,3}(x)$ is only non-zero in some short intervals, while it evaluates to zero everywhere else. This is a property of B-splines, called *local support*, means we only need to sum the contributions of the $n-1$ neighboring basis functions & control points.

Unfortunately, there is no closed-form of $B_{i,n}(x)$. However, using *Cox-de Boor's* recursive definitions:

$$B_{i,0}(x) = \begin{cases} 1 & \textbf{if} \quad t_i \le x \le t_{i+1} \\ 0 & \textbf{otherwise} \end{cases},$$

$$B_{i,n}(x) = T_i B_{i,n-1} + T_{i+1} B_{i+1,n-1},$$

$$T_{i+1} := \frac{t_{i+n+1} - x}{t_{i+n+1} - t_{i+1}},$$

$$T_i := \frac{t_i - x}{t_{i+n} - t_i},$$

gives us a way to compute $B_{i,n}$ by using the knots: $t_i, t_{i+1}, t_{i+n}, t_{i+n+1}$, and $B_{i,n-1}, B_{i+1,n-1}$ functions. Finally, by assuming *local support* and that we can find $B_{i,n}$ by using *Cox-de Boor*, we get a curve $\boldsymbol{S}_n$:

$$\boldsymbol{S}_n(x) = \sum_{i=l}^{k} B_{i,n}(x)\boldsymbol{p}_i, \; l = k-n, \text{ and } x \in [t_k, t_{k+1}],$$

requiring less computational power than initial one.

However, calculating *Cox-de Boor* repeatedly is quite expensive, and might not be suitable in e.g. real-time applications. Luckily, we can represent the same operations with *iterative curve subdivision*. It uses the control points $P_0 = \begin{bmatrix} \boldsymbol{p}_0 & \boldsymbol{p}_1 & \cdots & \boldsymbol{p}_4 \end{bmatrix}$ and a *subdivision matrix* S; for a uniform cubic B-spline:

$$S = \frac{1}{8}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 8 \\ 0 & 0 & 0 & 0 & 1 & 4 & 6 & 4 & 0 \\ 0 & 0 & 1 & 4 & 6 & 4 & 1 & 0 & 0 \\ 0 & 4 & 6 & 4 & 1 & 0 & 0 & 0 & 0 \\ 8 & 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^{\mathsf{T}}.$$

Calculating the next iteration $P_i = SP_{i-1}$ gives us more control points, a smoother curve than P_{i-1}. As $i \to \infty$ we'll get closer to the analytical S_n value. We can simply this even further by using the two rules below, which will re-weight existing $\boldsymbol{p}'_i$, adding new $\boldsymbol{p}'_i$ between every other pairs of control points.

$$\boldsymbol{p}'_i = \frac{1}{8}\begin{cases} \boldsymbol{p}_{i-1} + 6\boldsymbol{p}_i + \boldsymbol{p}_{i+1} & \textbf{if} \quad i \text{ isn't new,} \\ 4\boldsymbol{p}_i + 4\boldsymbol{p}_{i+1} & \textbf{otherwise} \end{cases}$$

Another related task is the *subdivision of surfaces*, which attempts to make surfaces appear smoother. There are two well known techniques, *Catmull-Clark* and *Loop's surface subdivision algorithm* [3]. We'll present *Loop's* since it uses triangles as primitives.

Simply divide each triangle into four new triangles and then assign them weights as shown in Figure 3. After applying Loop's algorithm, the mesh will have C^2 continuity. Notice that our mesh is assumed to be closed, and don't have to deal with boundaries.

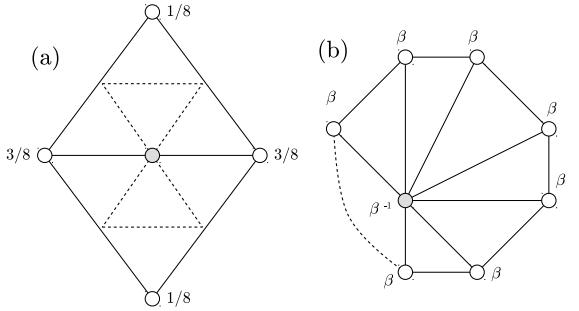


Figure 3: (a) Distribution of the weights for the new vertex (gray) based on existing vertex data (white). (b) Re-weighing factors for an existing vertex (gray).

$$\beta(k) = \begin{cases} 3 \div 8k & \textbf{if} \quad k > 3 \\ 3 \div 16 & \textbf{when} \quad k = 3 \end{cases}, \; \beta(k)^{-1} = \frac{1}{\beta(k)}$$

3 Method and Results

After briefly going through all of this theory, we now outline how to implement these techniques in practice. We'll also show some images of our results and provide relevant discussion around each technique.

3.1 Evaluating B-Spline Curves

Essentially, we want to evaluate the curve $\boldsymbol{S}_n(x)$ in some sampling points $x_1, x_2, ..., x_s$ along the curve. We could set this to e.g. $x_i = x_{i-1} + \Delta$, where Δ is a finite step forward taken along the curve. Since we are displaying the curve on a screen, it's natural that Δ relates to the amount moving us one pixel along our curve. Hence, we evaluate for each pixel.

But we are getting a bit ahead of ourselves here, how do we even evaluate $\boldsymbol{S}_n(x)$? If we can do this, then we can sample the curve and display our pixels. We are only given a bunch of control points $\{\boldsymbol{p}_i\}$, which roughly follow the general "feel" of the curve. We see in Figure 4 that the red dots on the blue line are the control points; notice that the blue line roughly follows the general "tendency" of the curve.

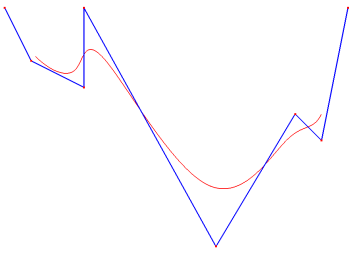


Figure 4: analytical evaluation of the cubic b-spline, where we also use the spline local support property. We see that in this case we are not including the boundaries, which can be done by wrapping around.

By using the basis functions $B_{i,n}$, we calculate how much each control points contributes to the evaluation at x along the curve. Summing over these gives us $\boldsymbol{S}_n$, which makes us happy. We get $B_{i,n}$ by recursively solving *Cox-de Boor*. After we have $B_{i,n}$, we can re-use and don't need to re-calculate it (e.g. Figure 2), we only need find which functions to use. Finally, and most importantly, we don't need to take the sum over all $\boldsymbol{p}_i$ and $B_{i,n}$. By assuming local support, $s = \lfloor x \rfloor$ gives x's closest basis, and, $\max\{0, s-1\}$ the *start*, $\min\{k-1, s+2\}$ the *end*. By summing from the *start* up till *end*, we get $\boldsymbol{S}_n$!

3.2 B-Spline Curve Subdivision

Instead of computing our cubic b-splines analytically by using *Cox-de Boor's* algorithm, we can instead approximate the curve $\boldsymbol{S}_n(x)$ by iteratively subdividing the previous curve, which we call $\boldsymbol{S}'_n(x)$.

Intuitively, by looking at Figure 5, we see that we initially only have a *linear interpolation* between the control points $\boldsymbol{p}_i$. Thereafter, in each iteration, we create additional control points between the initial ones based on the weights of the other neighboring control points. These weights are pre-defined, and are used to calculate the new weight of existing control points and the weight of newly added control points. As we have shown before, they are weighted:

$$\boldsymbol{p}'_i = \frac{1}{8}\begin{cases} \boldsymbol{p}_{i-1} + 6\boldsymbol{p}_i + \boldsymbol{p}_{i+1} & \textbf{if} \quad i \text{ isn't new,} \\ 4\boldsymbol{p}_i + 4\boldsymbol{p}_{i+1} & \textbf{otherwise} \end{cases}.$$

While this will never give smooth results, since we essentially have a piecewise flat curve in each step, it's usually enough to give nice results, especially if the sampling rate matches the curve's complexity. In the implementation we simply create a new control point in-between each existing control point, and re-weight them according to the relation above. We apply this concept repeatedly, until we have converged to some acceptable level of detail. We then linearly interpolate between neighboring control points. As can be seen below, we get reasonably detailed results after the third iteration. It somewhat even matches our other analytical result.

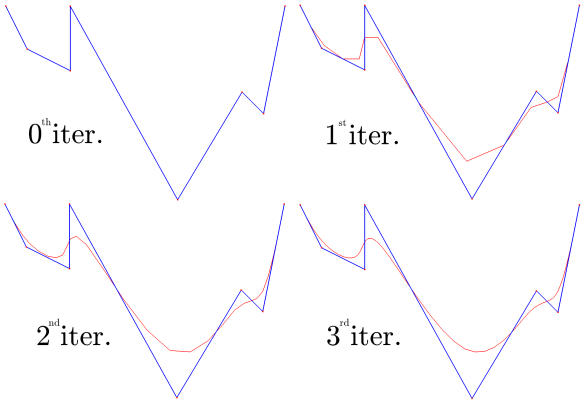


Figure 5: evaluation of the same curve as Figure 4, but instead uses curve subdivision. Notice that the first few iterations give fairly poor results, but we see in the 3^{rd} iteration we're getting smoother results.

3.3 Loop's Surface Subdivision

At last, we'll describe how to implement *Loop's subdivision scheme*. First we'll assume that the given mesh is *closed*. This enables us to remove a couple of special cases from the scheme, making it easier. As input, it takes a mesh $\mathcal{M}$, such as our faithful cow at the left-most part of Figure 6. Our goal is the output mesh in the right-most part of Figure 6, which increases the polygon count of the original cow whilst attempting to make it appear smoother.

In each iteration of the algorithm, we intend to produce a smoother mesh $\mathcal{M}_i$ by using a previous mesh $\mathcal{M}_{i-1}$. We define a subdivision operator $\mathcal{S}$ which will give us $\mathcal{M}_i = \mathcal{S}(\mathcal{M}_{i-1})$, where $\mathcal{M}_0 = \mathcal{M}$. In $\mathcal{S}$, our intention is to create more triangles and place them in such a way that the mesh appears smoother. We'll use the nearby vertices to do this.

First, we produce four triangles from each of the existing triangle in the mesh. You can see this visually in Figure 3 (a). This is done for all triangles in the mesh. While this will create more polygons, it will not give us smoother and smoother meshes in each step of the iteration. To create a smoother result, we need to move around both old and new vertices created in this step. We can divide them into the two rules: the *vertex rule* and the *edge rule*. Both are detailed in Algorithm 1 and Algorithm 2:

Algorithm 1 Vertex Rules in Loop's Algorithm

Require: some existing vertex v_i positioned at $\vec{x}_i$.
Ensure: it gives the re-weighted position $\vec{x}'_i$ for v_i.
  $w \leftarrow \beta(|N_i^v|)$
  $\vec{x}'_i \leftarrow \vec{x}_i(1 - w|N_i^v|)$
  for all $v_k \in N_i^v$ **do**
    $\vec{x}'_i \leftarrow \vec{x}'_i + w\vec{x}_k$
  end for
  return $\vec{x}'_i$

Algorithm 2 Edge Rules for Loop's Algorithm

Require: existing edge $e_{ij} = (v_i, v_j)$ in the mesh.
Ensure: it places a vertex v_n at $\vec{x}_n$ inbetween e_{ij}.
  $v_i \leftarrow \text{vertex}(e_{ij})$
  $e_{ji} \leftarrow \text{pair}(e_{ij})$; $v_j \leftarrow \text{vertex}(e_{ji})$
  $e_{kj} \leftarrow \text{prev}(e_{ij})$; $v_k \leftarrow \text{vertex}(e_{kj})$
  $e_{li} \leftarrow \text{next}(e_{ji})$; $v_l \leftarrow \text{vertex}(e_{li})$
  $\vec{x}_n \leftarrow \frac{3}{8}(\vec{x}_i + \vec{x}_j) + \frac{1}{8}(\vec{x}_k + \vec{x}_l)$
  return $\vec{x}_n$

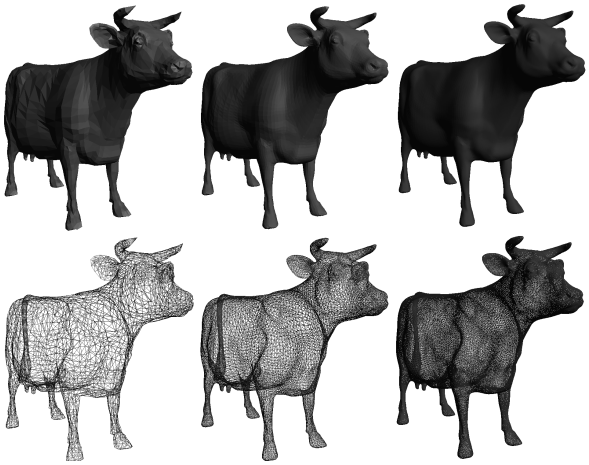


Figure 6: applying Loop's subdivision to our beloved cow mesh. The leftmost image is the original mesh, followed by one, then two, iterations of Loop's. We have also included the wireframe for each iteration.

Second, we'll show when to apply these rules. But first, perhaps an intuitive understanding is better. When we create our sub-triangles, we'll need a new vertex, i.e. the grey one in Figure 3 (a). But where do we place it? Since we want to preserve the overall structure of the mesh, it's natural for it to be some weighted combination of it's neighbors. Vertices which are directly connected via an edge should be more important, and are given the weight $3 \div 8$ while the other neighboring vertices are $1 \div 8$. This is called the *edge rule*, and is applied to the vertex which is created in-between any of the edges. Now the problem is that the old vertices won't be placed correctly, and will contribute to the mesh not being smooth. We thus *re-weight* the mesh by applying the *vertex rule* to all vertices. It takes into account the neighborhood N_i^v, and gives them weights by using the β-function we describe earlier.

After this, you should have a nice smooth mesh. As can be seen in Figure 6, it gets progressively smoother, but also increases in polygon count substantially. It's important to note that it will always be piecewise flat, and won't be analytically smooth. But that is usually enough for computer graphics...

Grading

According to the lab guidelines within the course `TNM079`, I've implemented features for a: 3 and 4. My aim with this assignment is a grade of scale: 4.