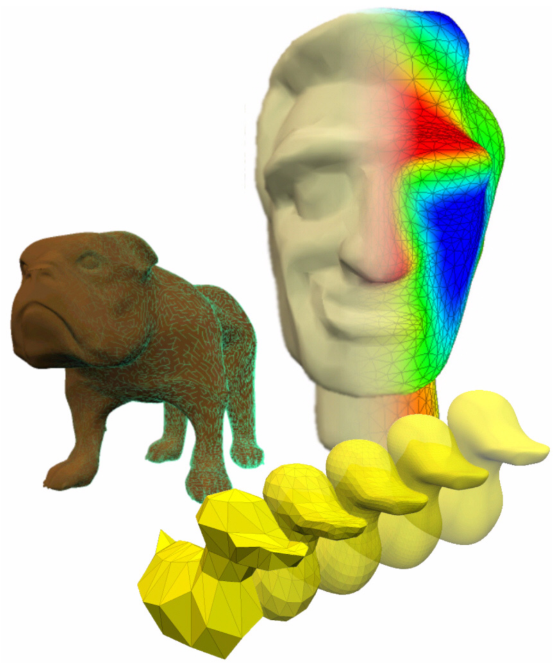
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Improving a demo program for triangle meshes



Semester project in computer science
realized at the computer graphics laboratory
ETH Zürich
2004/2005

submitted by

Michael Sauter

Tutor: Christian Sigg
Supervisor: Prof. Dr. Markus Gross

1

The triangle mesh demo program

In this chapter the demonstration program for triangular mesh processing will be presented in detail. Starting with a general description of the user interface, the focus will move on to the three main parts of this chapter, namely the mesh processing methods subdivision, mesh decimation and fairing. Each such part shall include documentation and illustration of the specific user interface, information about the theoretical background, a critical discussion on the applied methods and a presentation of the achieved results.

1.1 The user interface - General description

This section will provide a general description of the user interface of the triangle mesh demo program and point out some of the basic functionality.

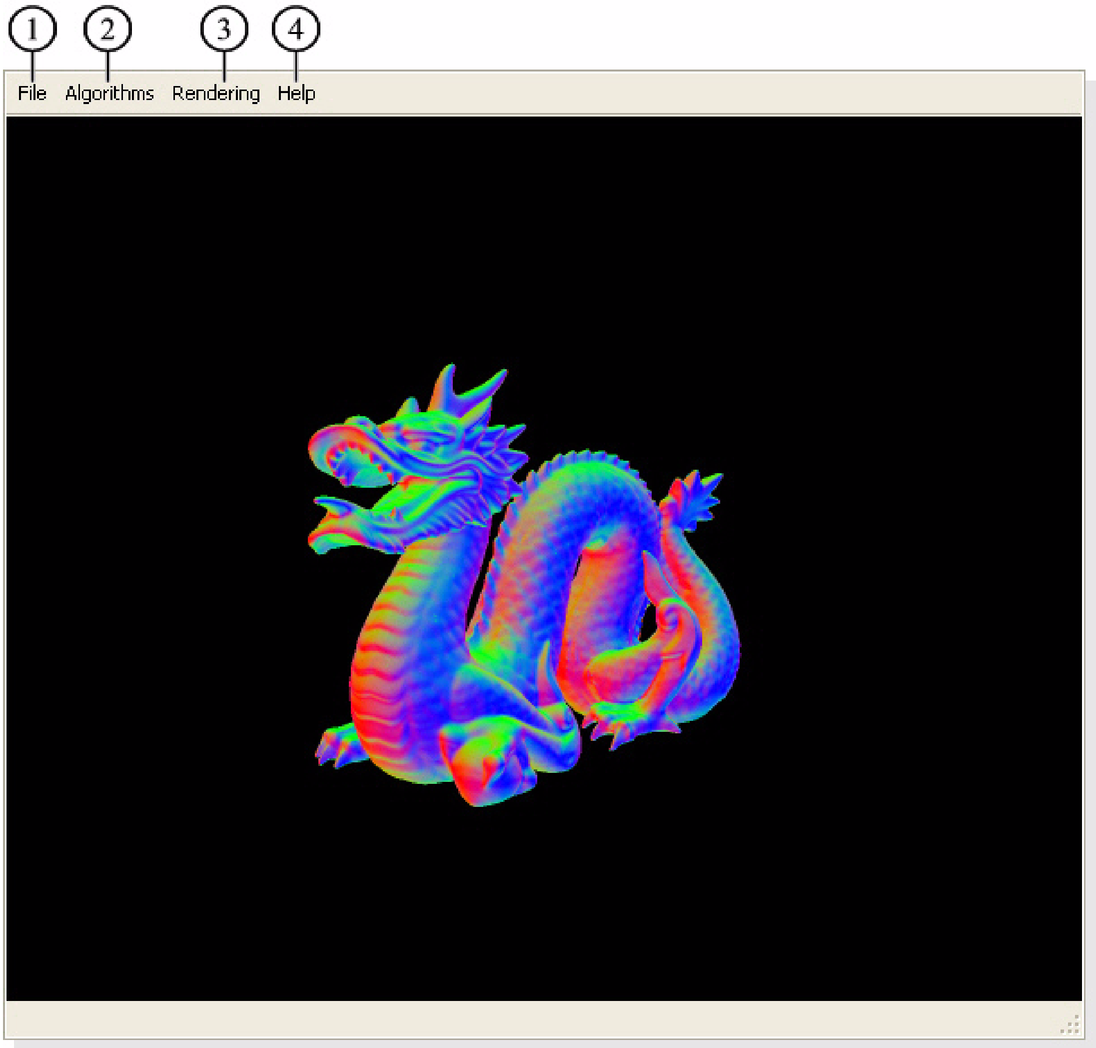


Figure 1.1: The triangle mesh demo program user interface. Dragon model with vertex normal coloring.

Figure 1.1 gives a first impression of what the demo program looks like. Following a very simple structure, the user interface consists of two main parts: A menu bar and a rendering view. The former offers a bunch of options to load, store and process a triangle mesh model, whereas the latter deals with the classical rendering part and supports some of the fundamental viewing actions like rotation or zooming. There is left to say that the triangle mesh demo program starts from a console, where additional information messages are posted during runtime.

1	File menu	Contains procedures for opening, reverting or saving a triangle mesh model. The supported file formats are **.obj*, **.off*, **.om*, **.stla*, **.stlb*. Furthermore, an option to load a texture image of type **.bmp*, **.png*, **.pnm*, **.xbm* or **.xpm* is at hand.
2	Algorithm menu	Holds and activates the mesh surface processing methods *subdivision* ("Subdivider"), *fairing* ("Smoother"), *mesh decimation* ("Decimater") and *mesh frequency decomposition* ("Frequency").
3	Rendering menu	Rendering mode selection. One can choose between wireframe, flat and Gouraud shading, point cloud rendering, hidden-line removal or textured rendering. Additionally there are drawing modes for the visualization of mesh surface properties like vertex normal direction, first and second principle curvature, mean as well as Gaussian curvature and shape index. These drawing modes can be found in the submenu "Colored Vertices". Last but not least, there is the possibility to show face and/or vertex normal vectors respectively by exploring the "Draw Vectors" submenu. Please note that the two mentioned submenus will offer further visualization modes depending on the currently active mesh surface processing method (for more information on that refer to the method specific sections in this chapter). The rendering menu can also be opened by pushing the right mouse button when pointing into the rendering view area.
4	Help menu	As the complexity of the demo program is still limited, there seems no use in offering some guidance to the user. However, for future developments, a help menu was already included, even without providing any real functionality.

Table 1.1: Description of the triangle mesh demo program menus.

A couple of rendering options can be accessed by specific keys which are listed below. Remember that the rendering view area must be active (click into the view area) to successfully process the key input.

Key	***Action***
"F"	Turn fog on/off.
"T"	Swap between different texture modes.
"S"	Turn specular reflection on/off.
"I"	Information output about the model and the scene on the console.

Table 1.2: Special keys.

1.2 Fairing

The visual realism of computer graphics applications heavily depends on the quality of the 3D models used, especially on their level of detail. Therefore, a high complexity for 3D models is often desired. However, designing such models from scratch is quite time consuming and can be troublesome. As a solution, the combination of 3D-laser range scanning devices with surface reconstruction techniques allows a rather easy acquisition of highly complex triangle meshes and is widely used in practice. Unfortunately, the scanning process is not free of measuring errors and as a consequence the acquired data may suffer from degradation by noise. To regain the original smoothness, the triangle mesh of the scanned object has to be post-processed and that is where fairing comes into play.

Fairing provides means to get rid of noise on triangular meshes as well as remove undesirable rough features. Fairing can be seen as an extension to lowpass filtering in signal processing and the basic idea behind it is to remove high frequency components from the surface geometry.

1.2.1 Notation and definitions

Throughout the following sections a couple of definitions will be used which shall be stated in this section.

A mesh will be denoted by X, its vertices by x_i (X can be seen as a vector $(x_1, x_2, ..., x_n)^T$ holding the mesh vertices). An edge connecting two vertices x_i and x_j will be called e_{ij}. E will stand for the set of all mesh edges. $N_1(i)$ will refer to the 1-ring neighbors of x_i, i.e., all the vertices x_j such that there exists an edge e_{ij} between x_i and x_j. In analogy, $E_1(i)$ will define the set of all edges adjacent to x_i, i.e., all edges e_{ij} such that x_j is in $N_1(i)$.

$L(X)$ will denote the Laplacian operator of the mesh X:

$$L(X) = X_{uu} + X_{vv} \tag{1.1}$$

where u and v parametrize the mesh surface. However, for practical reasons we will only work with a discretization of the Laplacian, i.e., linear approximations.

There is also a notion of second Laplacian which is defined as follows:

$$L^2(X) = L \bullet L(X) = X_{uuuu} + 2X_{uuvv} + X_{vvvv}\,. \tag{1.2}$$

1.2.2 Mesh frequencies

Defining the notion of frequencies in the context of triangle meshes is not obvious, but can be covered by introducing the so called generalized frequencies. Generalized frequencies are defined as the eigenvectors of the Laplacian and can be used to decompose the mesh into its frequency components. With such a decomposition a triangle mesh could be smoothed by simply discarding the high frequencies, i.e., the eigenvectors of the largest corresponding eigenvalues of the Laplacian. However, numerical instability and high computational cost make this approach infeasible for large meshes[1].

1.2.3 Diffusion equation

A more stable and more efficient way to attenuate noise in a mesh is through a diffusion process:

$$\frac{\partial X}{\partial t} = \lambda L(X)\,. \tag{1.3}$$

Integrating Equ. (1.3) over time will balance the noise throughout the mesh. The high frequencies will be smoothed, while the main shape will only suffer from a slight degradation.

Assuming a discretization of the Laplacian has been defined, the diffusion equation could be handled with a simple explicit Euler forward integration scheme:

$$X^{n+1} = (I + \lambda\, dt L) X^n \tag{1.4}$$

where X^n denotes the mesh after the n-th integration step.

The next matter of concern will be the problem of discretizing the Laplacian along with the question what this discretization should look like. As an answer, various discretization forms have been proposed, each of them resulting in a specific behavior of the smoothing process. In the following sections some of these discretized Laplacians will be presented.

1. Nonetheless, such a method was implemented in the demo program. See Section 1.2.11.

1.2.4 Umbrella operator

The umbrella operator approximates the Laplacian in a linear fashion at every mesh vertex:

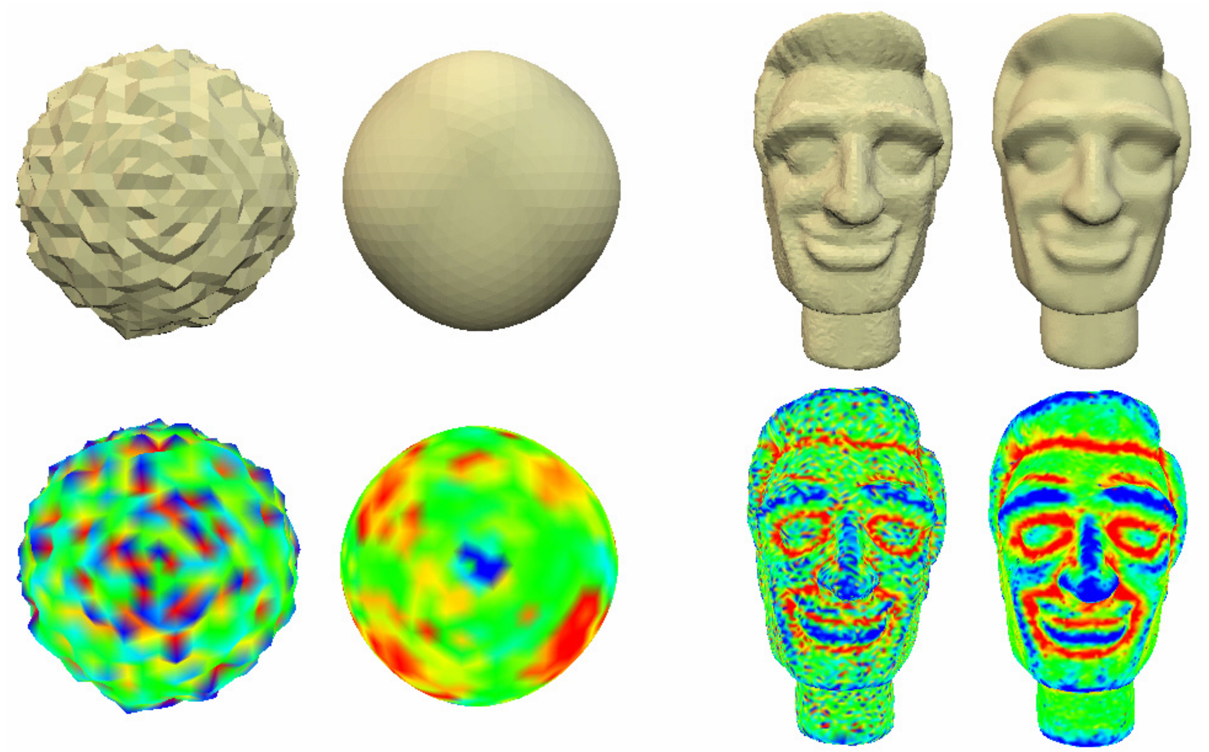
$$L(x_i) = \frac{1}{m} \sum_{j \in N_1(i)} x_j - x_i \tag{1.5}$$

where x_j is a neighbor of the vertex x_i and $m = \#N_1(i)$ is the number of these neighbors (valency of x_i).

Because of its linear form, the umbrella operator can be represented by a matrix, where every row is connected to a 1-ring neighborhood in the mesh and specifies the weights of its vertices. Solving Equ. (1.3) with an explicit Euler integration scheme as in Equ. (1.4) would then involve simple matrix-vector products. However, it seems more convenient to do the integration steps directly per vertex:

$$x_i^{n+1} = x_i^n + \lambda\, dt L(x_i^n) \tag{1.6}$$

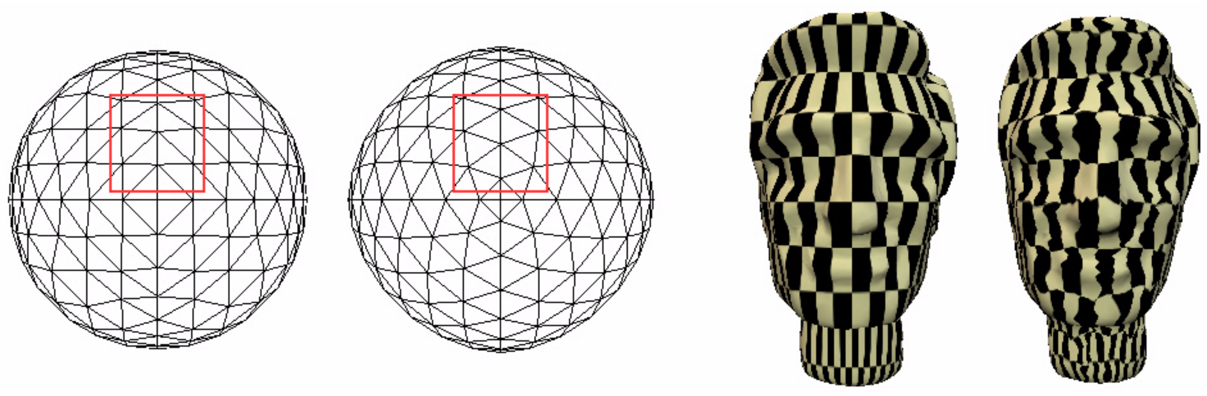
where x_i^n denotes the mesh vertex x_i after the n-th integration step. For reasons of stability, $\lambda\, dt < 1$ has to be fulfilled, otherwise the mesh will be teared apart during the integration process. This restriction can be relaxed by using an implicit integration scheme (see Section 1.2.8).



(a) (b) (c) (d)

Figure 1.2: (a): Noisy sphere (1'026 vertices), flat-shaded and with colored mean curvature. (b): Smoothed version after 8 iterations of the umbrella operator. High fluctuations in mean curvature vanish. (c): Noisy head (17'358 vertices). (d): Smoothed version after 2 iterations of the umbrella operator. The integration step size was set to 1.0 in both cases.

The umbrella operator tends to regularize the mesh, i.e., adapting the edge lengths and the angles in the 1-ring neighborhoods of the vertices. As a consequence, a vertex drift in highly irregular parametrized areas of the mesh occurs, which can distort the geometry severely and may lead to problems in the context of texturing. Hence, efforts were taken to improve the umbrella operator, which led to a more sophisticated version (see next section).



(a) **(b)**

Figure 1.3: Vertex drift using the umbrella operator. (a): 3 smoothing iterations on the sphere model. The regularization tendency of the umbrella operator can be observed, especially in the marked area. (b): 3 smoothing iterations on the head model. Texture gets distorted due to vertex drift.

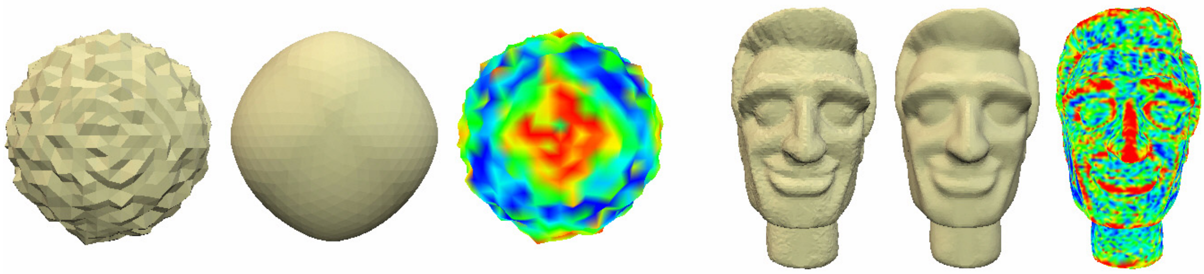
1.2.5 Improved umbrella operator

A relaxation to the inherent regularization approach of the umbrella operator is given by the improved umbrella operator. Not only the connectivity, but also the edge lengths are now taken into account.

$$L(x_i) = \frac{2}{E} \sum_{j \in N_1(i)} \frac{x_j - x_i}{|e_{ij}|}, \quad \text{with } E = \sum_{j \in N_1(i)} |e_{ij}| \tag{1.7}$$

where $|e_{ij}|$ is the length of edge e_{ij}.

Note that as opposed to the original umbrella operator, this discretization of the Laplacian is no longer linear because in principle, the edge lengths do not stay constant over the diffusion process. However, fixing the edge lengths for an integration step and thus linearizing the system, seems to work well and does not reveal any drawbacks.

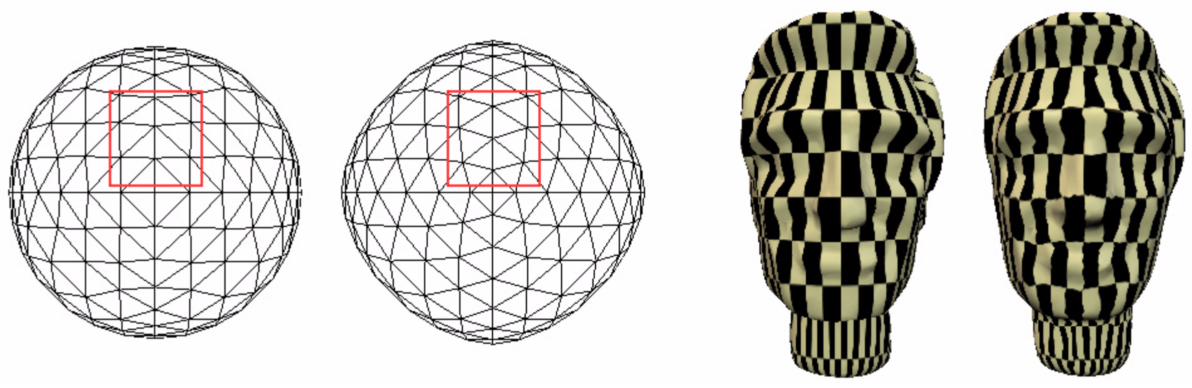


(a) **(b)** **(c)** **(d)**

Figure 1.4: (a): Noisy sphere (1'026 vertices), flat-shaded. (b): Smoothed version after 1 iteration of the improved umbrella operator using an integration step size of 1.0. The colored image visualizes smoothing distances. (c): Noisy head (17'358 vertices). (d): Smoothed version after 1 iteration of the improved umbrella operator. The integration step size was set to 0.0002. Smoothing distances are colorized in the rightmost image. Due to the numerical instability of the improved umbrella operator, implicit integration was used in both cases.

A major disadvantage of the improved umbrella operator is stability. Using an explicit integration scheme will mostly fail unless a really small step size is chosen. As a remedy, one can switch to an implicit integration scheme (see Section 1.2.8), which makes the improved umbrella operator applicable, at least for moderate integration step sizes.

Although the improved umbrella operator shows a better behavior when facing irregular parametrized areas, the undesired effect of sliding vertices can still be observed during the smoothing process, especially on flat surfaces.



(a) **(b)**

Figure 1.5: Vertex drift using the improved umbrella operator. (a): 1 smoothing iteration on the sphere model. The improved umbrella operator still shows the undesirable regularization behavior in the marked area. (b): 1 smoothing iteration with integration step size 0.003 on the head model. The texture gets distorted as well, but the improvement compared to the original umbrella operator is obvious. In both cases implicit integration was used.

1.2.6 Curvature flow

To tackle the problem of drifting vertices (see previous sections), the curvature flow operator has been developed. Its goal is to remove noise without depending on the parametrization of the mesh surface. Smoothing will only take place along the surface normal $\boldsymbol{n}$ and the moving speed will be determined by the mean curvature $\bar{\kappa}$:

$$\frac{\partial x_i}{\partial t} = -\bar{\kappa}_i \boldsymbol{n}_i \tag{1.8}$$

where $\bar{\kappa}_i$ and $\boldsymbol{n}_i$ denote the mean curvature and the surface normal at vertex x_i respectively.

This approach leads to the curvature flow operator (the derivation of the formula can be found in [1]):

$$L(x_i) = \frac{1}{\sum\limits_{j \in N_1(i)} (\cot\alpha_j + \cot\beta_j)} \cdot \sum_{j \in N_1(i)} (\cot\alpha_j + \cot\beta_j)(x_j - x_i) \tag{1.9}$$

where α_j and β_j form the triangle angles opposite to the common edge e_{ij}. Figure 1.6 illustrates the setting for the curvature flow operator.

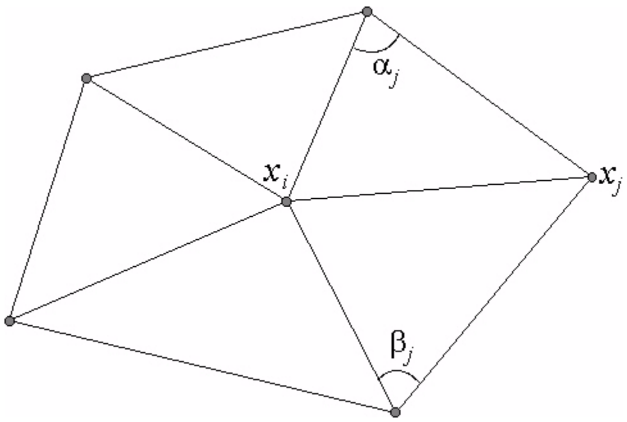


Figure 1.6: Illustration of curvature flow operator.

As stated at the beginning of this section, the curvature flow operator does not depend on the parametrization of the mesh surface, hence no drifting vertices can be observed. The advantage of using the curvature flow operator is visualized in Figure 1.8(b). Unlike the umbrella operator (Figure 1.3(b)), the texture of the head model is not distorted.

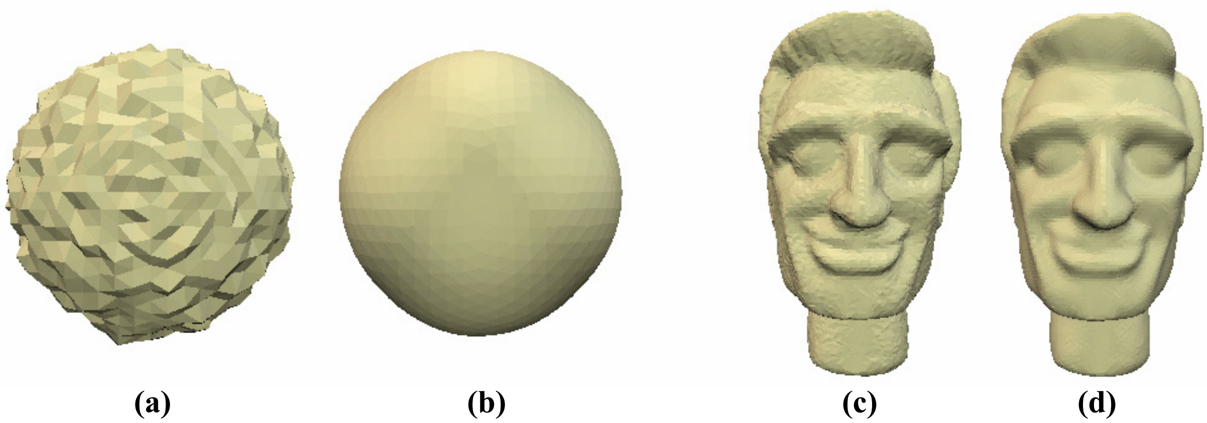


Figure 1.7: (a): Noisy sphere (1'026 vertices), flat-shaded. (b): Smoothed version after 8 iterations of the curvature flow operator. (c): Noisy head (17'358 vertices). (d): Smoothed version after 2 iterations of the curvature flow operator. The integration step size was set to 1.0 in both cases.

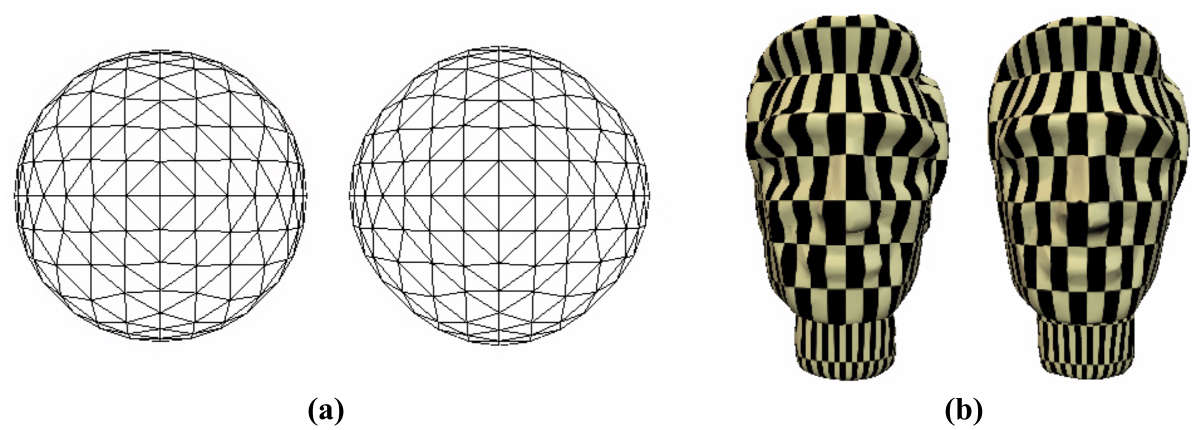


Figure 1.8: No vertex drift using the curvature flow operator. (a): 5 smoothing iterations on the sphere model. The curvature flow operator does not touch the underlying parametrization, the vertices "stay" in place. (b): 5 smoothing iterations on the head model. No texture distortion at all.

1.2.7 Second order difference operator

A further approach to construct a discrete fairing operator is based on second order differences. Second order differences are defined as the difference between two normals on neighboring triangles and can be associated with their common edge. They will be referred to by $D(e)$, where e denotes the common edge (see Figure 1.9).

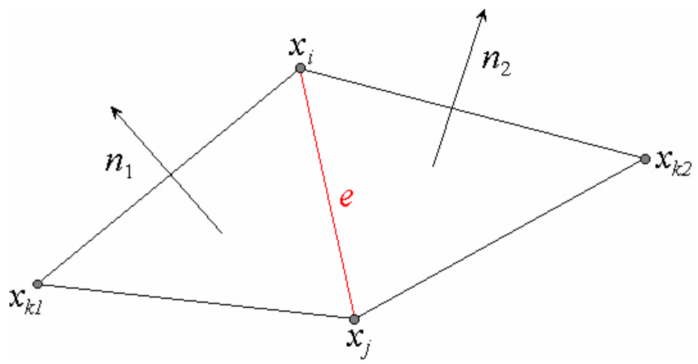


Figure 1.9: Second order difference $D(e) = \|n_1 - n_2\|$.

Fairing is done by minimizing the energy H, defined as the sum of the squared second order differences over all mesh edges.

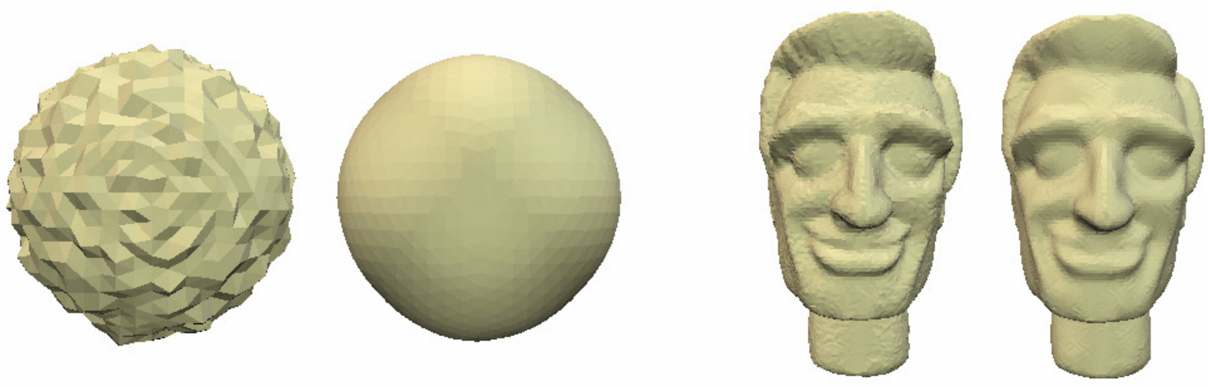
$$H = \sum_{e \in E} D(e)^2 . \tag{1.10}$$

$$X = \arg min_X H . \tag{1.11}$$

Solving the minimization problem leads, after sorting and summation, to the following per-vertex expression:

$$x_i = -\left(\sum_{e \in E_1(i)} c_{e,i} \alpha_e \right) \Big/ \left(\sum_{e \in E_1(i)} c_{e,i}^2 \right) . \tag{1.12}$$

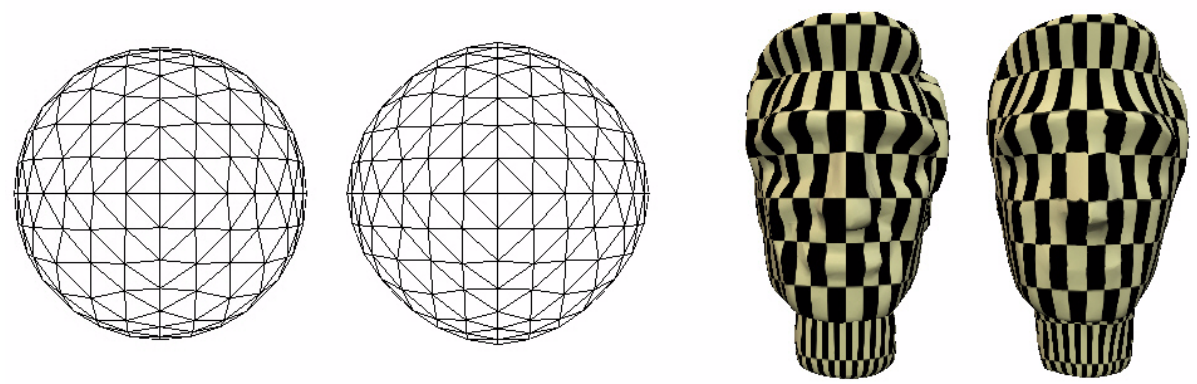
The coefficients $c_{e,i}$ are defined in terms of signed triangle areas and edge lengths. They also appear as weights in the terms α_e, which represent linear combinations of mesh vertices lying in the 1-ring neighborhood $N_1(i)$ of vertex x_i. A more detailed description of these coefficients can be found in [2], here they will not be discussed any further. Important property of Equ. (1.12) is that the fairing process using second order differences has a local characteristic again, i.e., only the 1-ring neighborhood of a vertex influences how it will be smoothed.



(a) **(b)** **(c)** **(d)**

Figure 1.10: (a): Noisy sphere (1’026 vertices), flat-shaded. (b): Smoothed version after 8 iterations of the second order difference operator. (c): Noisy head (17’358 vertices). (d): Smoothed version after 2 iterations of the second order difference operator. The integration step size was set to 1.0 in both cases.

Similar to the curvature flow operator, there is no need to concern about drifting vertices, since second order differences do not depend on the mesh parametrization either (see Figure 1.11). Experimenting with the second order difference operator showed that it is advisable to choose small integration step sizes or make use of an implicit integration scheme (see Section 1.2.8), whereby the stability can be clearly improved.



(a) **(b)**

Figure 1.11: As with the curvature flow operator, no drifting vertices can be found using the second order difference approach. (a): 5 smoothing iterations on the sphere model. (b): 10 smoothing iterations on the head model. Almost no texture distortion.

As a final remark, there should be stated that the second order difference operator is, from a computational point of view, the most expensive operator of those presented in this documentation, but at the same time the least aggressive one, when focusing on the smoothing degree.

1.2.8 Implicit fairing

So far the diffusion equation Equ. (1.3) from Section 1.2.3 was assumed to be solved by an explicit Euler integration scheme. For most purposes the usage of such a scheme is sufficient and serves well. However, as a consequence, the choice for the integration step size will be restricted, i.e., it has to be smaller than one, otherwise great stability problems will be encountered (see Figure 1.12).

(a) **(b)** **(c)** **(d)**

Figure 1.12: (a): Bunny model (35'947 vertices), flat-shaded. (b): Instability of explicit integration scheme for step size 20.0. For the single smoothing iteration the umbrella operator was used. (c): Implicit integration scheme. The same integration step size as in (b) leads to a nicely smoothed bunny mesh. (d): 20 smoothing iterations of the umbrella operator using the explicit integration scheme with step size 1.0 lead to similar smoothness as in (c). Despite the multiple iterations this approach is still an order of 15 times faster than the implicit approach.

To relax the restriction on the integration step size and to increase the overall stability of the diffusion process, one may consider to apply an implicit integration scheme. Thus, the integration step for Equ. (1.3) can be written as:

$$X^{n+1} = (I - \lambda\, dt\, L)^{-1} X^n . \tag{1.13}$$

Since we use linear approximations for the discretized Laplacian, we will end up with the following linear equation system:

$$(I - \lambda\, dt\, L) X^{n+1} = X^n . \tag{1.14}$$

Solving this system seems to be straightforward, but turns out to be a bit tricky. The point to consider is the problem size. Writing the discretized Laplacian in matrix form will lead to a n by n matrix, where n is the number of mesh vertices, and thus solving the system will be infeasible for large triangle meshes. However, since the matrix $A = (I - \lambda\, dt\, L)$ is sparse (each row contains approximately six non-zero, off-diagonal elements, which is the expected size of a vertex 1-ring neighborhood for triangular meshes), an iterative solving approach can be taken into account. The preconditioned bi-conjugate gradient method (refer to [14] for some pseudocode and more information) turns out to be suitable as it is based on simple matrix-vector multiplies which burn down to linear time computation because of the sparsity of matrix A.

Using the implicit integration scheme allows to deal with large integration step sizes and therefore achieves smoothing results which would require multiple runs in the explicit integration case. Nevertheless, despite the efficient linear solver, the implicit integration approach is still significantly slower than its explicit counterpart. Furthermore, it has to be said that visual differences in the results of the two integration schemes are hardly observable.

1.2.9 Volume preserving, anti-shrinking fairing

Using the diffusion process modeled by Equ. (1.3) as the basis for fairing, holds an inconvenient matter. Diffusion induces shrinkage, a property which is normally not desirable. Depending on the smoothing degree, the impact of shrinkage may be significant and countermeasures have to be taken into account (see Figure 1.13).

(a) **(b)** **(c)**

Figure 1.13: (a): Noisy sphere (1’026 vertices), Gouraud shading. (b): Smoothed version without volume preservation. The improved umbrella operator together with the implicit integration scheme was used. The integration step size was set to 1.0. (c): Smoothed version of (a) with activated anti-shrinking.

Taubin proposed to use a linear combination of the Laplacian L and the second Laplacian L^2 of the form $(\lambda + \mu)L - \lambda\,\mu L^2$ to minimize shrinkage. However, the parameters λ and μ have to be tuned by the user and differ from mesh to mesh.

A more general and much simpler method can be found in [1]. The idea behind their approach is to preserve the volume of an object throughout the smoothing process. Therefore, first of all, the interior volume of a triangle mesh must be computed. This can be done by summing the volumes of all the oriented pyramids centered at a point in space (the origin, for instance) and with the triangles of the mesh as bases. For a more formal definition let $x_{k,1}$, $x_{k,2}$ and $x_{k,3}$ denote the three vertices of the k-th triangle of a mesh. Then the expression for the volume looks as follows:

$$V = \frac{1}{6}\sum_k g_k \cdot N_k \tag{1.15}$$

where $g_k = (x_{k,1} + x_{k,2} + x_{k,3}) / 3$ and $N_k = (x_{k,2} - x_{k,1}) \times (x_{k,3} - x_{k,1})$. Since the sum goes over all mesh triangles, the time complexity of the computation will be linear in the number of triangles.

With the volume formula at hand, an automatic anti-shrinking fairing can be implemented quite easily. Starting with the initial mesh volume V_0, a smoothing integration step is applied, which leads to a new volume V_n. To cancel the shrinking effect, we want to scale the mesh back to its initial volume V_0. This can be achieved by simply scaling the mesh vertices with the factor

$\beta = (V_0 / V_n)^{1/3}$. Note that with this approach there is no need to tune any parameters, the anti-shrinking process adapts automatically to the mesh and to the smoothing. The nice effect of anti-shrinking extended fairing can be observed in Figure 1.13.

As a critical remark, volume preservation can not prevent degeneration of thin parts of the mesh.

1.2.10 Fairing – The user interface

The fairing specific operations are handled by the tool bar shown in Figure 1.14. The different action areas, marked by numbers, represent the interface to the discussed theoretical methods so far. Note that a special mesh coloring entry called “Smoothing Distance” to visualize the effect of fairing is available and can be found in the “Colored Vertices” rendering submenu.

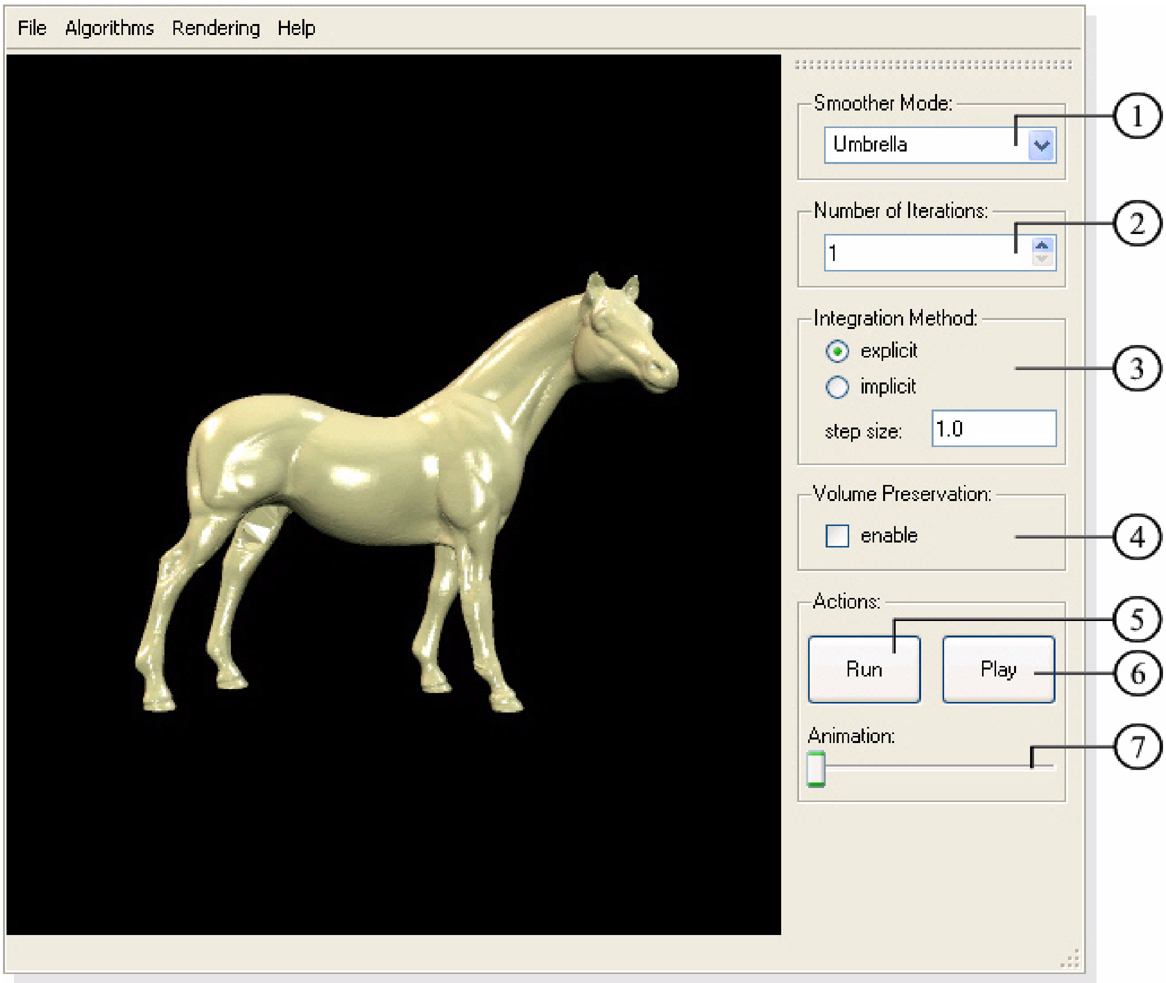


Figure 1.14: User interface for fairing.

1	Selection of the fairing operator. The available operators are *umbrella*, *improved umbrella*, *curvature flow* and *second order difference* (see sections 1.2.4, 1.2.5, 1.2.6 and 1.2.7).
2	Number of successive smoothing iterations to perform. Make use of this option to store all intermediate smoothing steps, which allows for more detailed animations.
3	Integration method for solving the diffusion equation. Implicit integration can handle integration step sizes greater than *1.0* without facing stability problems, however, it is significantly slower than the explicit integration scheme, which in return is bound to small integration step sizes (see sections 1.2.3 and 1.2.8). Note that the input field for the integration step size also accepts floating point numbers printed in the format [*mantissa*]*e*[*exponent*]. So *0.02* may be printed as *2e-2*, for example.
4	Enable/disable mesh volume preservation (see Section 1.2.9).
5	Start the smoothing process.
6	Show an animated transition between the mesh before and after the fairing step.
7	Slider to morph between the original and the smoothed mesh by hand.

Table 1.3: Description of the fairing tool bar.

1.2.11 Mesh frequency decomposition

As mentioned in Section 1.2.2, the notion of generalized frequencies can be used to describe the frequencies of a mesh. Generalized frequencies are defined as the eigenvectors of the Laplacian and consequently represent solutions to the following eigenproblem:

$$L\,y = \lambda\,y \tag{1.16}$$

where L denotes the discretized Laplacian of a mesh X. Let e_k be the generalized eigenvectors solving the eigenproblem given by Equ. (1.16), then the mesh X can be written as a linear combination of these eigenvectors:

$$X = \sum_{k=1}^{n} \alpha_k e_k \tag{1.17}$$

where n holds the number of mesh vertices. The coefficients α_k can be computed by projecting the mesh, or rather its vertices, onto the eigenvectors e_k. This is done by applying the inner product:

$$\alpha_k = \langle X, e_k \rangle\,. \tag{1.18}$$

One might notice that this is a generalized fourier transform.

The frequency decomposition of a mesh allows to apply methods from signal processing like lowpass filtering, for example. The idea to realize lowpass filtering is quite simple. Instead of reconstructing the mesh by using all its eigenvector terms, the ones with the largest corresponding eigenvalues are discarded. Say that the eigenvectors are sorted according to their eigenvalues in ascending order. Then lowpass filtering would be performed as follows:

$$X_{lowpass} = \sum_{k=1}^{l} \alpha_k e_k + \cancel{\sum_{k=l+1}^{n} \alpha_k e_k}\,. \tag{1.19}$$

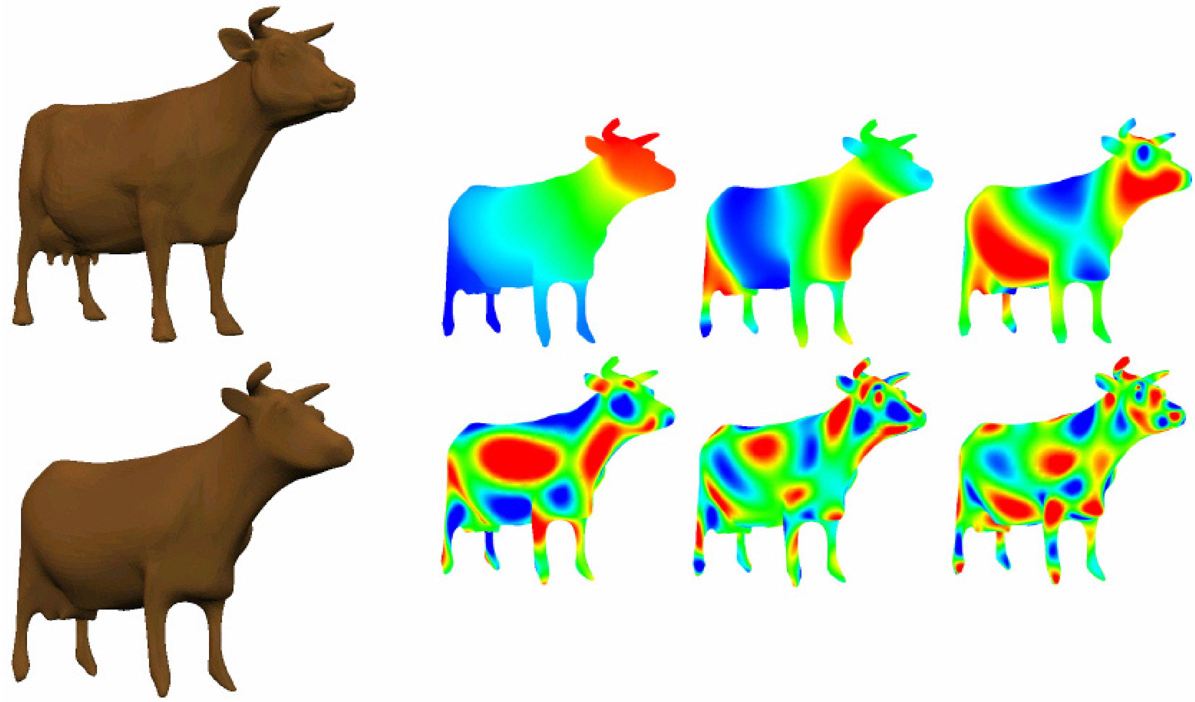
Since lowpass filtering removes high frequencies, the effect on the mesh will be a natural smoothing.

(a) (b) (c)

Figure 1.15: (a): Noisy sphere (1'026 vertices/frequencies). (b): Lowpass filtered mesh using the 50 lowest frequency terms. (c): Further filtering by taking only 3 of the 50 computed frequencies from (b) into account.

Given the method of mesh frequency decomposition, one might ask why to bother about the whole smoothing topic any longer, as it seems to be the most intuitive approach from a signal processing point of view. Well, the considerable problem of mesh frequency decomposition lies in its numerical instability. Doing an eigenvector/eigenvalue decomposition of the discretized Laplacian might be appropriate for small mesh sizes, however, with growing vertex number it becomes time consuming and instable (recall, the discretized Laplacian is represented by a $n \times n$ matrix, where n is the number of vertices). Furthermore, the discretized Laplacians listed in the previous sections are not symmetric in general, which increases the complexity of the problem. Still, there is one property which seems to make the task more manageable. As stated in Section 1.2.8 the discretized Laplacian matrix is generally extremely sparse. This allows to take some iterative approaches into account concerning the eigenvector/eigenvalue decomposition of the discretized Laplacian. The Implicitly Restarted Arnoldi Method (IRAM) proved to be suitable since it efficiently provides approximations of extremal eigenvalues and their corresponding eigenvectors. The basic idea behind IRAM is to orthogonally project the given eigenproblem onto a subspace, the so called Krylov Subspace, where the projected eigenproblem can be solved using a QR-algorithm, for instance. Then the computed eigenvectors are back-projected, forming approximations to the eigenvectors of the original problem (refer to [15] for details). The drawback of the method is that its focus lies on the ends of the spectrum of eigenvalues, i.e., only the smallest or largest eigenvalues, and hence the corresponding eigenvectors, are approximated. However, for lowpass filtering this behavior is quite acceptable as the goal is to concentrate on the low frequencies anyway.

The results shown in Figure 1.15 and 1.16 were achieved using the iterative Arnoldi method. The experienced computation time proves to be bearable as long as the number of eigenvectors to approximate is small and the mesh size consists of only a few thousand vertices. However, for larger meshes the performance drops down significantly. Still, the full eigenvector/eigenvalue decomposition is beaten by lengths.



(a) **(b)**

Figure 1.16: (a): Original cow mesh (11’610 vertices/frequencies) and lowpass filtered version using the 150 lowest frequency components (computation time: ~15min on Pentium 4 1.8GHz). (b): Coloring of 6 of the 150 computed frequencies.

Although the applied Arnoldi method induces a great deal of efficiency, it sometimes shows an undesirable behavior. Instead of computing the eigenvectors of the smallest eigenvalues, the ones of the largest eigenvalues are returned. Restarting the computation once or twice solves the problem, which leads to the supposition that the random starting vector used by the Arnoldi method might influence the converging behavior of the algorithm significantly.

1.2.12 Mesh frequency decomposition – The user interface

Applying the mesh frequency decomposition method is done via the tool bar shown in Figure 1.17. To visualize the computed mesh frequencies, an additional drawing option can be found in the “Colored Vertices” rendering submenu called “Frequency Mass”.

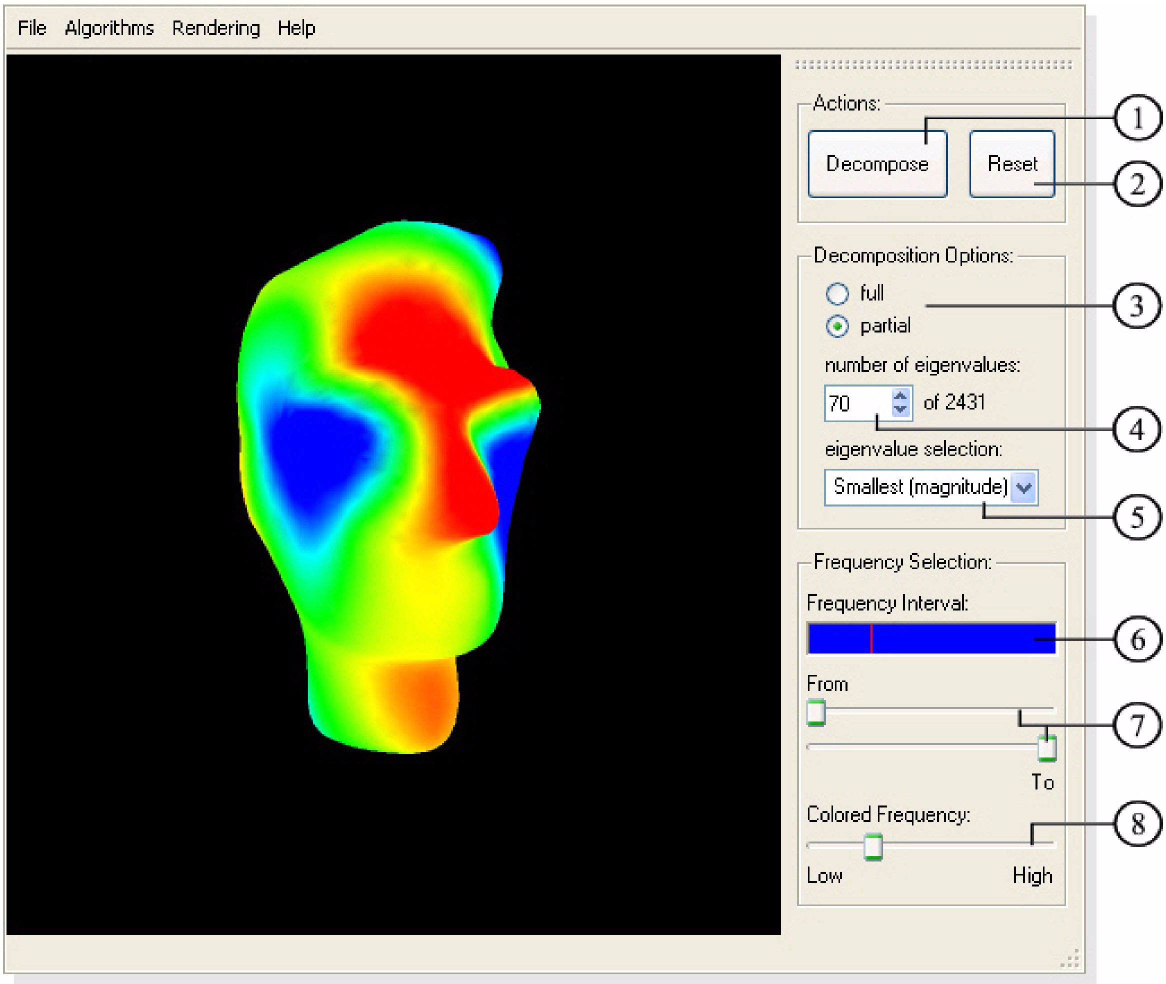


Figure 1.17: User interface for mesh frequency decomposition.

1	Decompose the mesh into its frequency components.
2	Restore the mesh model to the state before frequency decomposition took place.
3	Choose the method to compute the eigenvalues of the discretized Laplacian (see Section 1.2.11). Although full eigenvalue/eigenvector decomposition is available, it is not recommended for use, as the computational costs are high and the numerical behavior is instable. For large meshes the partial eigenvalue/eigenvector decomposition based on the Arnoldi method does a much better job.
4	Specify the number of eigenvalues/eigenvectors to approximate (only available when doing a partial decomposition).
5	Select which eigenvectors/eigenvalues to approximate (only for partial decomposition). Either the largest or the smallest eigenvalues of the spectrum can be computed. Note that as the discretized Laplacian matrix is asymmetric in general, there is the theoretical possibility to get complex eigenvalues. Therefore, one can state whether to search for smallest/largest eigenvalues according to their real part or their norm.
6	The frequency bar shows the range of computed and applied eigenvectors. A red marker indicates the selected frequency for the colored frequency visualization.
7	The sliders allow to further bound the range of used frequencies for the mesh reconstruction. Taking advantage of this option enables to show the effect of removing or adding additional frequencies on the mesh surface.
8	Selection of the frequency to visualize, when frequency coloring is activated in the rendering menu.

Table 1.4: Description of the frequency decomposition tool bar.

1.3 Subdivision

One of the oldest fields in 3D computer graphics can be found in the context of geometric modeling. Computer-aided design applications early revealed the power and usefulness of fast code processing units in design and modeling areas and inspired many follow-up products. Nowadays, people heavily depend on modeling tools, be it in 3D-model construction and processing for movies and computer entertainment or in the design and prototyping of real world products. However, the growing size and complexity of the objects to build soon cried for sophisticated methods to avoid drowning in the model's sea of vertices and faces. In this context Bézier Patches and NURBS had a big impact as they provide means to build and control complex surfaces by a comparatively small number of control points. Their major advantage lies in their offer of guaranteed smoothness properties concerning the limit surface. Still, for some applications these approaches hold a significant drawback, since they depend on a surface parametrization. It might be easy to derive such a parametrization in the case of constructing a virtual landscape, but turns out to be troublesome when dealing with an object of arbitrary form. Facing this inconvenience, the connection to the topic of subdivision can be made. It seems to be obvious that one would happily take advantage of the nice properties of NURBS, for instance, while working with triangle meshes. However, as mentioned above, finding a natural surface parametrization for arbitrary triangular meshes is not a simple task at all. Nonetheless, to efficiently create and process complex triangle meshes, a helping hand offering the comfort of control points might become essential. Additionally, one would like to make some statements about the smoothness to expect from the limit surface, in analogy to NURBS[2]. Both requests will be approached with subdivision.

2. Pushing the comparison between subdivision and splines, it might be interesting to mention that in fact, many subdivision schemes are based on generalizations of various spline types.

The idea behind subdivision seems quite simple. Given a coarse triangular mesh, the surface triangles are subdivided into subtriangles in a refinement step (for example by using the 1-to-4 refinement, as shown in Figure 1.18). Then the vertices are moved according to the so called subdivision rule which operates on local mesh characteristics only and aims at producing smooth surfaces. Finally, by repeating these two steps, an increasingly smooth surface is obtained. However, the real subdivision surface is reached only in the limit of the process. A prove of smoothness properties can be given for most subdivision surfaces, which justifies their application from a theoretical point of view as well.

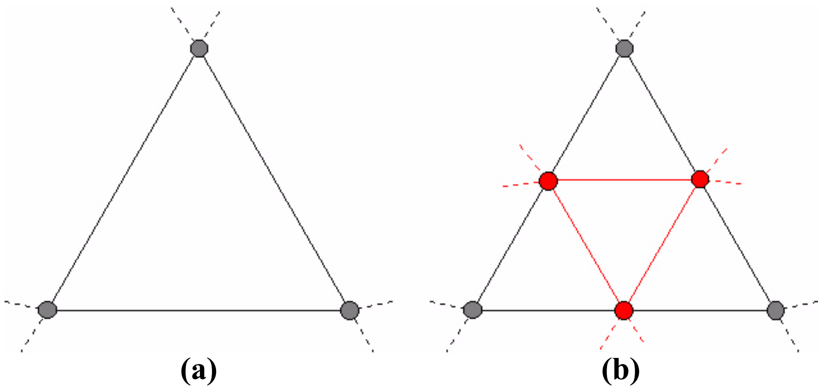


Figure 1.18: 1-to-4 refinement using dyadic splits. (a): Mesh face before refinement step. (b): Refined mesh face. The new vertices, drawn in red, are called ***odd***, the old vertices, in dark gray, ***even***.

In the following sections the focus will be on different subdivision rules, as they play the key role in the subdivision process, i.e., the subdivision rules define the properties of the limit surface.

1.3.1 Notation and definitions

The notation used in the subsequent sections is summarized here.

Regular and extraordinary vertices. The topology of mesh vertices plays an important role in subdivision, therefore two types of vertices are distinguished. In the context of triangular meshes, *regular* vertices are those with valency 6 (for interior vertices) and valency 4 (for vertices on the mesh boundary) respectively. All other vertices are considered as *extraordinary*. Note that under the refinement step illustrated in Figure 1.18, every newly created interior vertex will be regular.

Odd and even vertices. As depicted in Figure 1.18, the vertices of the coarser mesh are also vertices of the refined mesh. For any subdivision level, all new vertices that are created at that level are called *odd* vertices. The vertices inherited from the previous level are called *even*.

1.3.2 Loop subdivision scheme

The Loop subdivision scheme for triangular meshes was proposed by Charles Loop [9]. It pursues an approximating approach, i.e., even vertices will not keep their position throughout the subdivision process, neither will they be part of the limit surface. The Loop subdivision scheme is based on the three-directional box spline, consequently one can expect a C^2-continuous limit surface. However, this only holds for regular meshes. At extraordinary points the smoothness reduces to C^1-continuity or even C^0-continuity.

As already mentioned, the subdivision rule holds the positioning scheme for the odd and even vertices after the refinement step. The new coordinates are computed using weight masks of local range, i.e., linear combinations of neighboring vertex positions. Note that concentrating on the near neighborhood makes subdivision extremely efficient.

The Loop subdivision rule distinguishes between several cases. There are different subdivision masks for odd and even vertices as well as for vertices on the boundary. The detailed masks, illustrating the vertices and their corresponding weights to use for computing the new positions, are given in Figure 1.19.

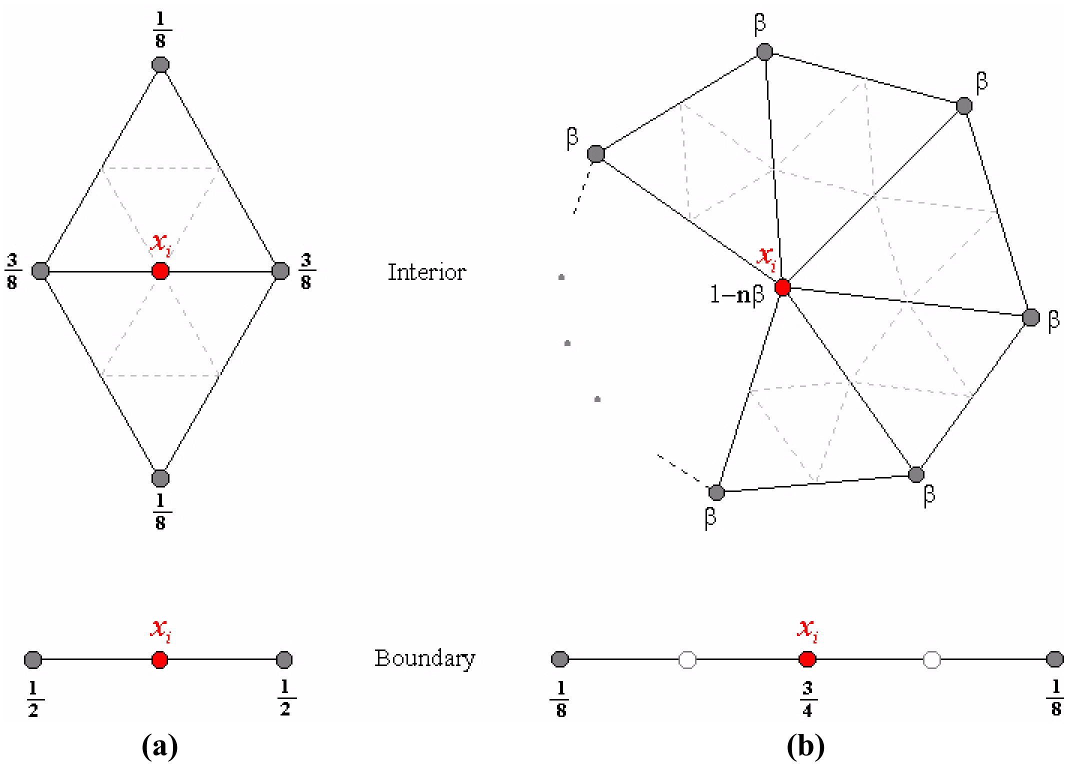


Figure 1.19: Subdivision masks for Loop scheme. The vertex x_i, marked in red, is the vertex for whom the new position has to be computed. The dark gray vertices stand for even vertices, the white ones for odd vertices. The dotted lines represent the new topology after the refinement step. (a): Mask for odd vertices. (b): Mask for even vertices. In both cases the boundary vertices are handled separately.

$$\beta = \frac{1}{n}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\left(\frac{2\pi}{n}\right)\right)^2\right),\ n \text{ is the valency of the vertex } x_i.$$

The definition of the weight β in Figure 1.19 reveals that it is not fixed, but rather depends on the valency of the vertex at hand. Therefore, one might consider to precompute and store the weight for a certain number of valencies to increase the subdivision performance.

The results obtained by applying the Loop subdivision scheme are shown in Figure 1.20. The visual smoothness of the surface after a few iterations can be easily observed. Simultaneously an undesirable effect of the Loop scheme gets apparent, the volume of the model shrinks during the subdivision process. If volume preservation is an issue, one might use the anti-shrinking approach presented in the section on fairing (Section 1.2.9).

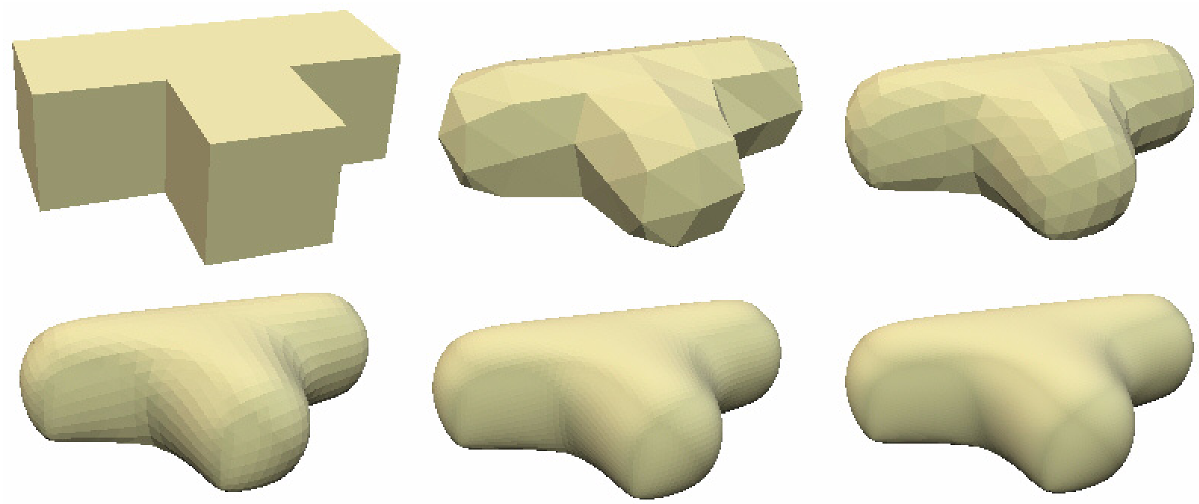


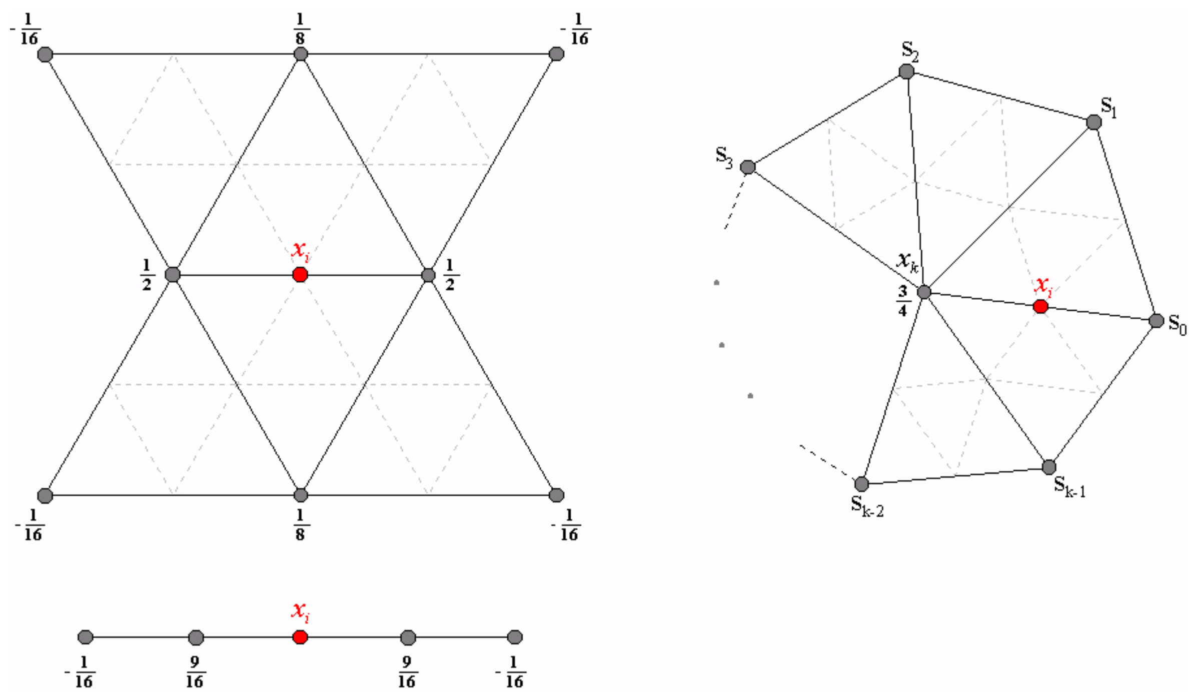
Figure 1.20: Successive subdivision iterations using the Loop subdivision scheme. Starting with 20 vertices, the model complexity rises to 74, 290, 1’154, 4’610 and finally 18’434 vertices.

1.3.3 Modified Butterfly subdivision scheme

In the previously presented subdivision scheme the positions of the even vertices altered in every subdivision step. As a consequence, the influence of the control points on the resulting surface remained limited. The Modified Butterfly scheme, dealt with in this section, follows an interpolating approach, i.e., the even vertices keep their position throughout the subdivision process and hence will be part of the limit surface.

The original Butterfly scheme was first proposed by Dyn, Gregory and Levin in [7]. It was supposed to reach C^1-continuity on the complete surface, however, this property only held for regular[3] meshes. To ensure C^1-continuity on arbitrary meshes, the extraordinary vertices have to receive special treatment as shown in [10].

The subdivision mask for the Modified Butterfly scheme is presented in Figure 1.21. Note that, compared to the Loop subdivision scheme, the local support is larger, i.e., vertices beyond the 1-ring neighborhood are included.



(a) (b)

Figure 1.21: Subdivision masks for Modified Butterfly scheme. The vertex x_i, marked in red, is the vertex for whom the new position has to be computed. The dark gray vertices stand for even vertices. The dotted lines represent the new topology after the refinement step. (a): Mask for regular vertices (interior and boundary case). (b): Mask for vertices adjacent to an extraordinary vertex.

$$s_i = \frac{1}{n}\left(\frac{1}{4} + \cos\left(\frac{2i\pi}{n}\right) + \frac{1}{2}\cos\left(\frac{4i\pi}{n}\right)\right), \text{ for } n > 5;$$

$$s_0 = \frac{3}{8}, s_{1,3} = 0, s_2 = -\frac{1}{8}, \text{ for } n = 4; \; s_0 = \frac{5}{12}, s_{1,2} = -\frac{1}{12}, \text{ for } n = 3.$$

n is the valency of the extraordinary vertex x_k.

3. A regular mesh consists of regular vertices alone (see Section 1.3.1).

In analogy to the Loop scheme, it is advisable to precompute the mask weights s_i, defined in Figure 1.21, and thus speed up the subdivision process. Note that the presented set of subdivision masks is not complete. In Figure 1.21(a) the upper mask can face certain conditions under which some of the vertices weighted by (-1/16) are absent. An easy way to solve the problem is by reflecting the opposite vertices and use them instead. However, to ensure C^1-continuity extra masks have to be provided in these cases (see [11]).

The effect of the Modified Butterfly subdivision scheme is shown in Figure 1.22. The interpolating behavior can be nicely observed. However, interpolation comes at the cost of smoothness, which gets apparent when comparing with the result of the Loop scheme. To catch the importance of the modification applied to the original Butterfly scheme, Figure 1.23 shows the effect of not giving special treatment to extraordinary vertices.

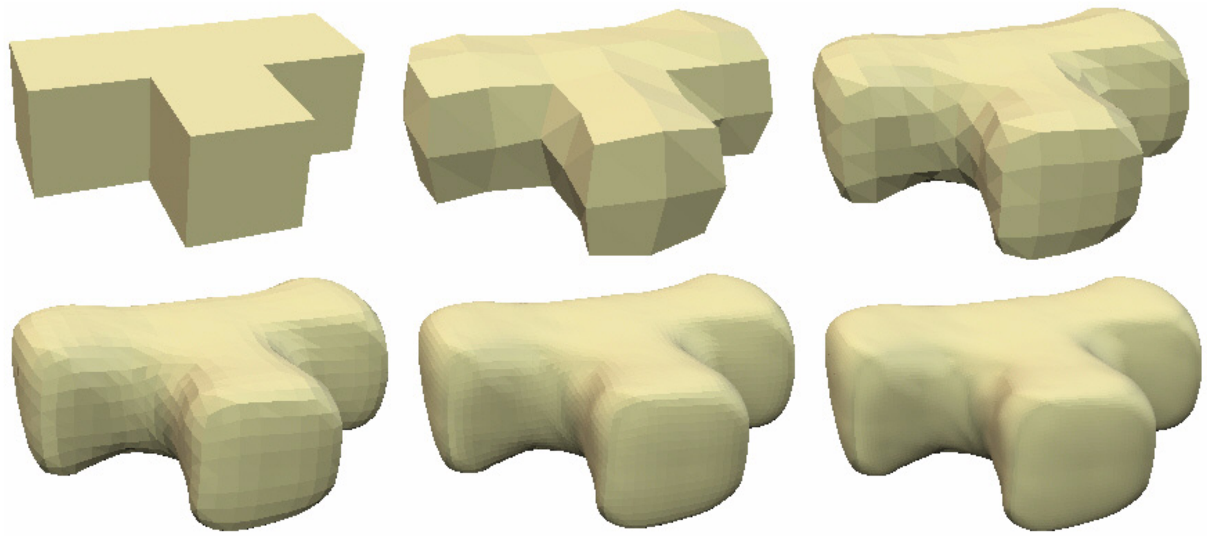


Figure 1.22: Successive subdivision iterations using the Modified Butterfly subdivision scheme. Starting with 20 vertices, the model complexity rises to 74, 290, 1’154, 4’610 and finally 18’434 vertices.

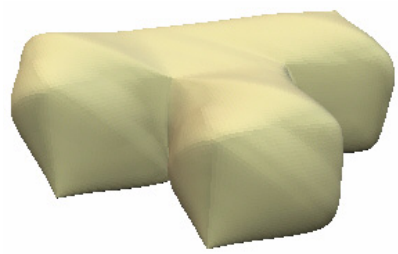


Figure 1.23: T-Shape model after 5 iterations of the original Butterfly subdivision scheme. The problem areas around extraordinary vertices, where no C^1-continuity is reached, seem apparent.

1.3.4 Sqrt3-Subdivision scheme

Another interesting subdivision scheme was presented by Kobbelt in [8]. It is called $\sqrt{3}$-subdivision and belongs, like the Loop scheme, to the class of approximating schemes. Contrary to the discussed subdivision schemes so far, the refinement step is not done by using a 1-to-4 split strategy[4]. Kobbelt proposed to use a different face dissection which increases the number of faces by a factor of 3 in every refinement step. This is done by inserting a new vertex for every face of the given mesh, followed by an edge flipping operation as shown in Figure 1.24. Note that applying the refinement step twice leads to a 1-to-9 refinement of the original mesh. As this corresponds to a so called tri-adic split (two new vertices are introduced for every original edge) the scheme was named $\sqrt{3}$-subdivision.

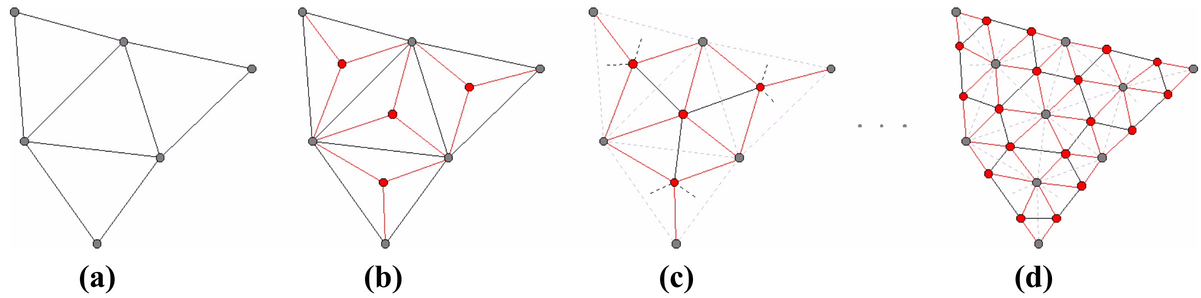


Figure 1.24: Refinement step for the $\sqrt{3}$-subdivision scheme. Odd vertices are drawn in red, even vertices in dark gray. The light dotted lines mark the flipped edges. (a): Mesh faces before refinement step. (b): For every triangle face a new vertex is inserted. (c): The edges connecting even vertices are flipped. (d): After two refinement steps we have a 1-to-9 refinement of the original mesh faces.

As for all subdivision schemes, the subdivision rule must not be missing, thus it is illustrated in Figure 1.25. Note that boundary vertices are updated in every second subdivision iteration only. The given rule was designed such that the $\sqrt{3}$-subdivision scheme reaches C^2-continuity on the limit surface except for the regions around extraordinary vertices, where one can expect C^1-continuity.

The alternative refinement rule, which provides a “slower” mesh refinement in contrast to classical subdivision schemes, makes the $\sqrt{3}$-subdivision scheme also well suited for more sophisticated subdivision methods like locally adaptive refinement. For more information on that refer to [8].

4. Every triangle face of the mesh is split into 4 subtriangles during a refinement step.

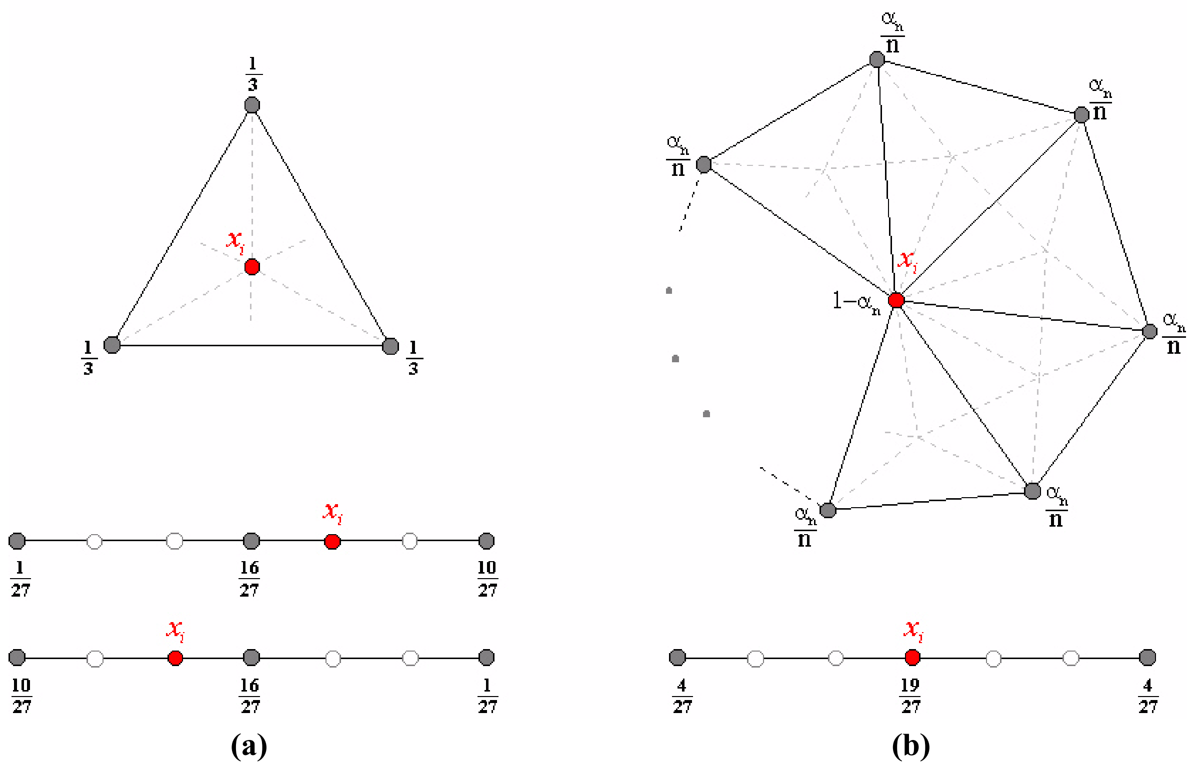


Figure 1.25: Subdivision masks for $\sqrt{3}$-subdivision scheme. The vertex x_i, marked in red, is the vertex for whom the new position has to be computed. The dark gray vertices stand for even vertices, the white ones for odd vertices. The dotted lines represent the new topology after the refinement step. (a): Mask for odd vertices (interior and boundary case). (b): Mask for even vertices (interior and boundary case).

$$\alpha_n = \frac{1}{9}\left(4 - 2\cos\left(\frac{2\pi}{n}\right)\right)$$, n is the valency of the vertex x_i.

Results of the $\sqrt{3}$-subdivision scheme are given in Figure 1.26. The visual similarity to the Loop scheme can be observed in every second iteration step. The slower refinement process, compared to the other subdivision schemes, can be noticed as well.

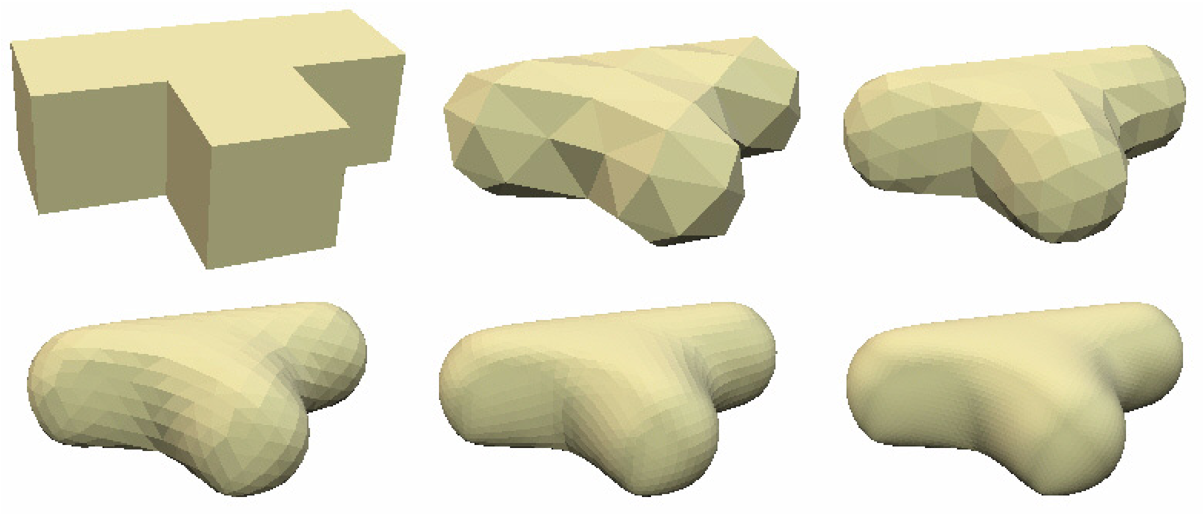


Figure 1.26: Successive subdivision iterations using the $\sqrt{3}$-subdivision scheme. Starting with 20 vertices, the model complexity rises to 56, 164, 488, 1'460 and finally 4'376 vertices.

1.3.5 Subdivision – The user interface

The subdivision methods from the previous sections are implemented in the triangle mesh demo program and can be tested using the tool bar depicted in Figure 1.27. There is an additional drawing mode to color the mesh refinement process. It is accessible through the "Colored Vertices" rendering submenu and is named "Subdivision Vertices".

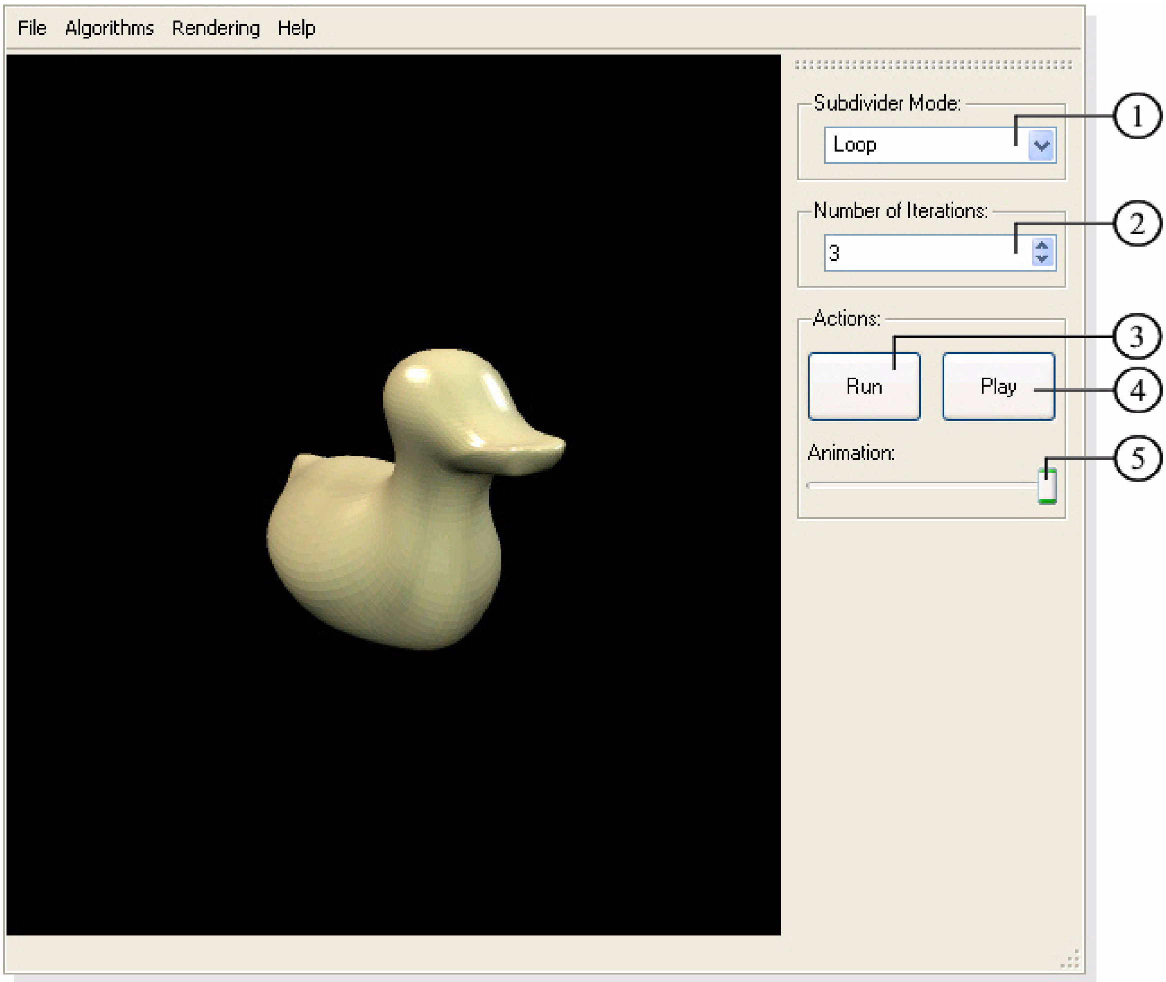


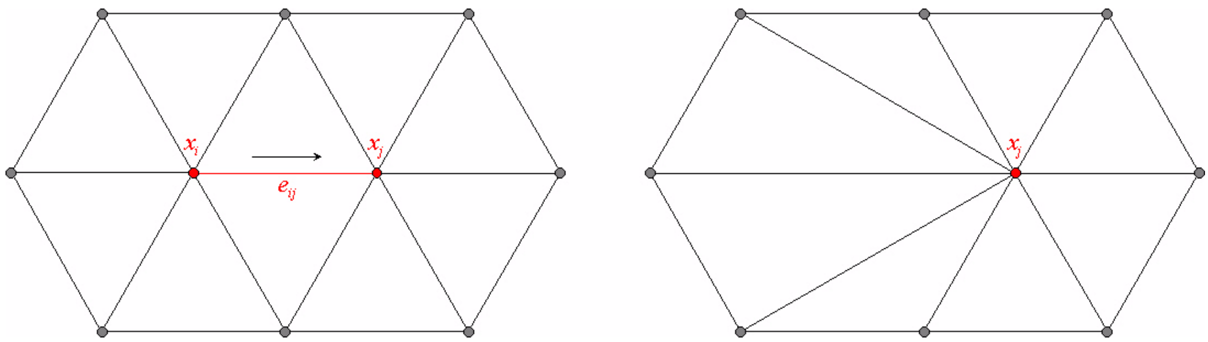
Figure 1.27: User interface for subdivision.

1	Choose the subdivision scheme among *Loop*, *Sqrt3*, *Butterfly* and *Modified Butterfly* subdivision (see sections 1.3.2, 1.3.3 and 1.3.4).
2	Specify the number of successive subdivision iterations. Useful to show the development of the subdivision process, since all intermediate steps are stored.
3	Subdivide the triangle mesh.
4	Show an animated transition between the mesh before and after the performed subdivision.
5	Slider to morph between the original and the subdivided mesh by hand.

Table 1.5: Description of the subdivision tool bar.

1.4 Mesh decimation

The hunger for ever increasing mesh complexity and hence more and more realistic 3D-models can be observed for years and is reflected in the impressive performance gain of computer graphics hardware so far. However, for real-time or web applications one might face the problem to reduce the complexity of a given mesh model to ensure acceptable processing rates. Doing the task by hand seems impractical, thus methods to decimate a mesh, i.e., decrease the number of faces, have been developed. The main goal, besides pure decimation, is to preserve the original shape as good as possible, which means keeping the introduced error low. Many different strategies to reduce the mesh complexity have been proposed. Some of them try to remove appropriate vertices, edges or faces and retriangulate the resulting hole in the mesh using fewer triangle faces. Others merge vertices of a mesh model using spatial binning. A very popular idea considers collapsing edges according to assigned weights (see Figure 1.28). The nice properties of this approach lie in the rather simple setting, the efficient data structure to represent the decimation procedure, the reversibility of the process using so called vertex splits and the “modest” impact of a single edge collapse on the whole mesh. Furthermore, a decimation scheme based on edge collapses can be quite easily extended to support features like geomorphs[5] between different decimation degrees of a model, mesh compression or selective mesh refinement. The mesh decimation process using edge collapses leads to the notion of a *progressive mesh*, which builds the model of the decimation method implemented in the triangle mesh demo program as well.



(a) **(b)**

Figure 1.28: (a): Mesh before edge collapse. (b): Mesh after edge collapse.

As mentioned above, the edge collapse method assigns weights to every edge. These weights are subsequently used to build a priority queue, indicating which of the edges should be collapsed first. Intuitively it seems clear that the weights somehow have to indicate the error which is introduced, when collapsing the corresponding edge. The function which computes the edge weights will be referred to as cost function. Needless to say that the result of the decimation process will heavily depend on how this cost function is chosen. Many different cost functions were proposed so far, all with their own place on the line marking the trade-off between quality and efficiency. The following sections will present a couple of cost functions and illustrate their effects on the decimation process.

5. A smooth visual transition between two meshes.

1.4.1 Quadric error metric

The quadric error metric uses a heuristic to characterize the geometric error induced by an edge collapse. A symmetric 4×4 matrix Q_i is assigned to every vertex v_i and defines the induced error $\Delta(v_i)$ as:

$$\Delta(v_i) = v_i^T Q_i v_i . \tag{1.20}$$

Please note that the mesh vertices v_i are defined in homogeneous coordinates, i.e., the vertex $x_i = (x_{ix}, x_{iy}, x_{iz})^T$ becomes $v_i = (x_{ix}, x_{iy}, x_{iz}, 1)^T$. The matrix Q_i is built such that the error $\Delta(v_i)$ represents the sum of squared distances to some planes p:

$$\Delta(v_i) = \sum_{p \in planes(v_i)} (p^T v_i)^2 \tag{1.21}$$

$$Q_i = \sum_{p \in planes(v_i)} p^T p . \tag{1.22}$$

The planes $p = (a, b, c, d)^T$, defined by the equation $ax + by + cz + d = 0$, are initially chosen to coincide with the triangle faces adjacent to the vertex v_i. Consequently the starting error estimate will be 0.

The cost of collapsing an edge e_{ij} connecting vertex v_i with v_j is now given by:

$$\Delta(e_{ij}) = \bar{v}^T (Q_i + Q_j) \bar{v} \tag{1.23}$$

where $\bar{v}$ denotes the vertex the edge is collapsed to.

The quadric error metric provides a quite good error measure for edge collapses. Furthermore, it can be computed very fast, which is an important property, especially when dealing with large triangle meshes. Visual proof of the quality using the error quadrics approach is given in Figure 1.29, where even at a high edge collapse degree, the overall shape of the original mesh is nicely preserved.

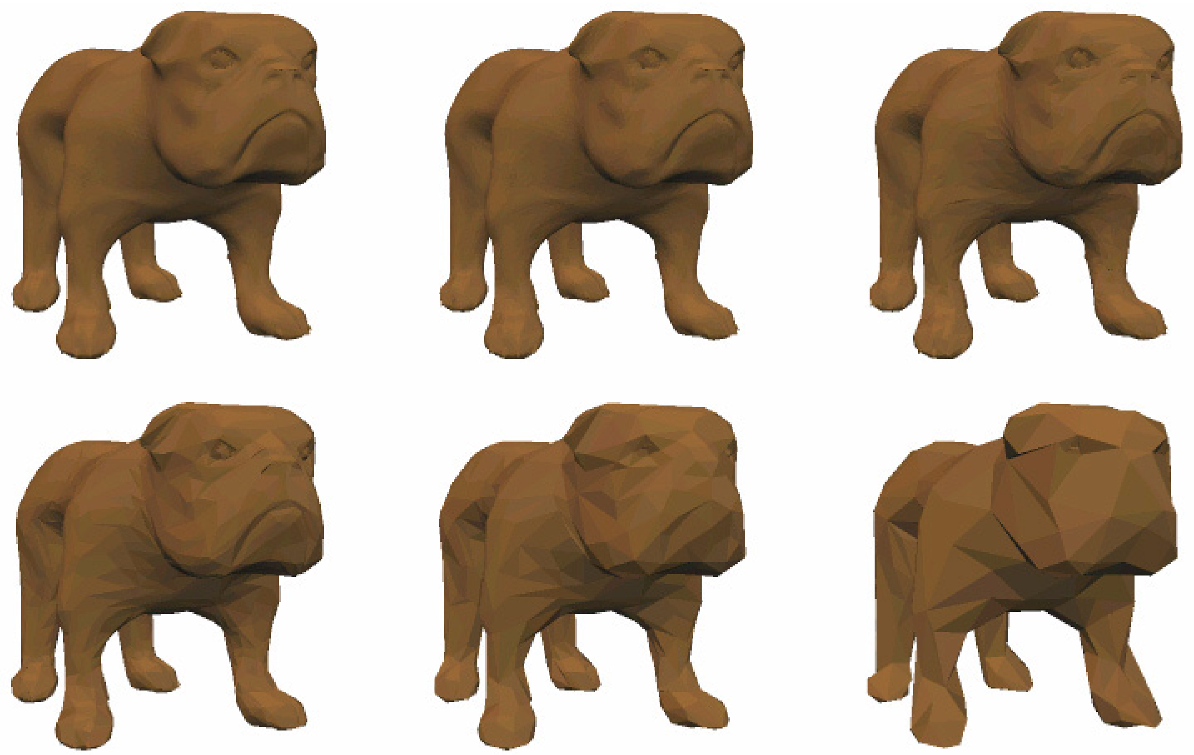


Figure 1.29: Decimation of triangle mesh using quadric error metric. The initial model, consisting of 28’264 vertices, is decimated by successive iterations, leading to a complexity reduction down to 11’306, 4’523, 1’810, 725 and finally 291 vertices. In every step 60% of all edges were collapsed.

1.4.2 Roundness

One might weight an edge collapse according to the shape of the triangles affected by the collapse. A desirable property for those triangles might be that they should not be degenerate. The triangle roundness criteria introduces such a measure, i.e., it quantifies the deviation of a triangle from an ideal equilateral triangle. This is done by dividing the radius of the circumference by the length of the shortest edge of the triangle (see Figure 1.30). Let R denote the above mentioned measure, then we get the normalized roundness of a triangle by building the ratio between the smallest possible R value and the one obtained from the triangle under consideration. Minimal $R = \sqrt{1/3}$ is achieved in the case of an equilateral triangle.

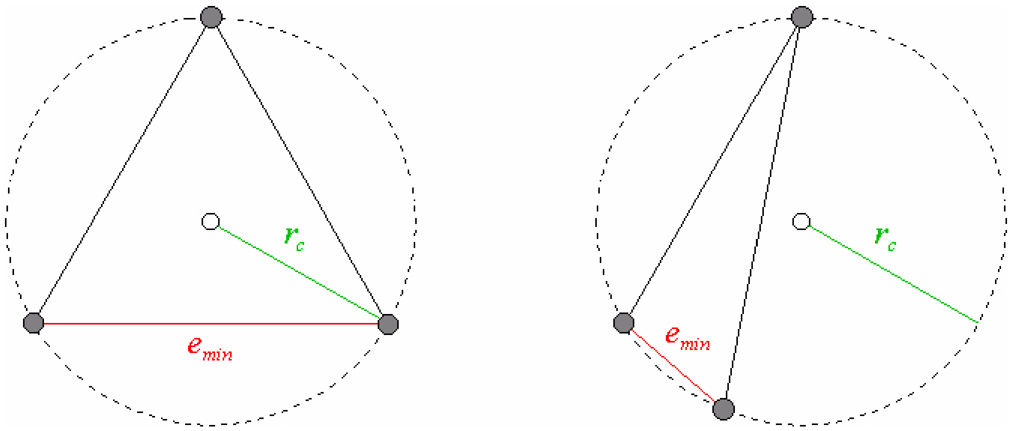


Figure 1.30: Roundness criteria $R = |r_c|/|e_{min}|$

To weight an edge collapse, the situation after the collapse has to be considered and the normalized roundness of all affected triangles must be computed and compared. The smallest roundness value is assigned to the edge collapse as weighting term, then decimation is done by performing the edge collapses with largest weights first.

The quality of roundness based mesh decimation is visualized in Figure 1.31. Despite a lower decimation degree, normalized roundness performs poorly compared to the quadric error metric approach. However, as to computational speed, the roundness criteria is undefeated.

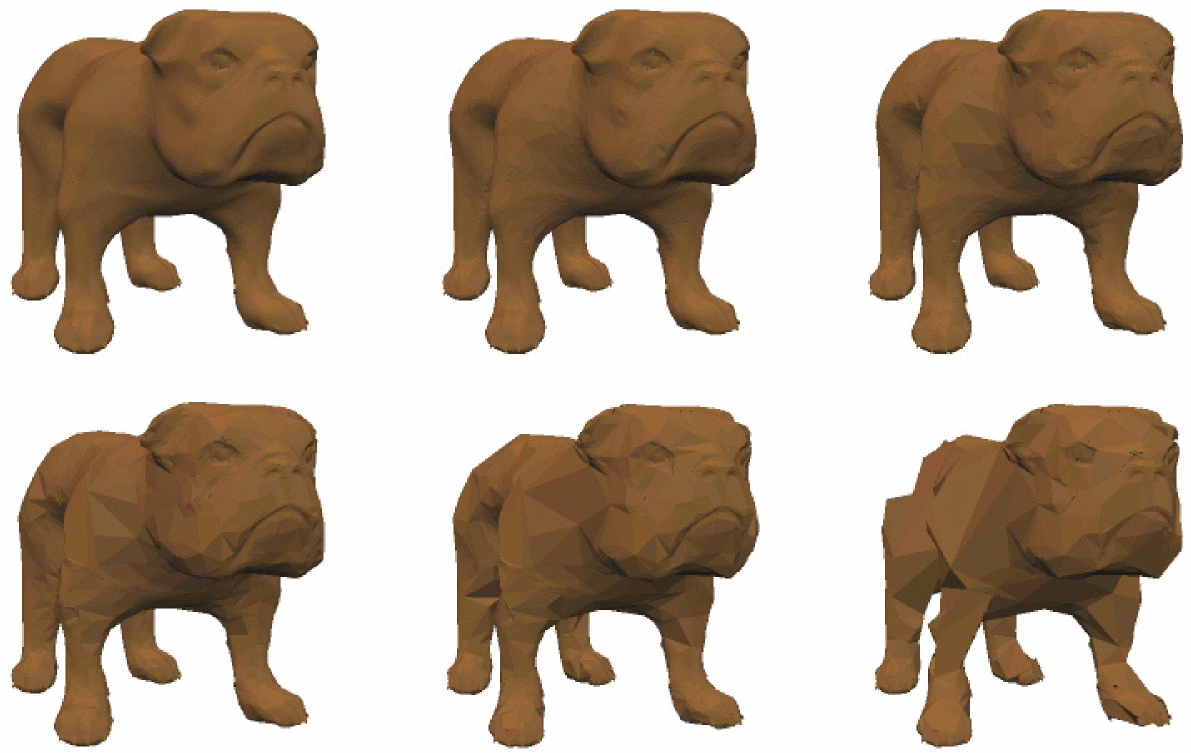


Figure 1.31: Decimation of triangle mesh using normalized roundness criteria. The initial model, consisting of 28’264 vertices, is decimated by successive iterations, leading to a complexity reduction down to 19’785, 13’850, 9’696, 6’788 and finally 4’752 vertices. In every step 30% of all edges were collapsed.

1.4.3 Binary constraints

In addition to the use of cost functions to prioritize edge collapses, binary constraints might be taken into consideration. As opposed to weights on edge collapses, binary constraints act much more restrictive, i.e., they define whether an edge collapse is legal or not. If an edge collapse turns out to be illegal, it will not be performed. Thus binary constraints may help to avoid unwanted constellations on the mesh surface, mostly at the expense of decimation degree. A common approach for binary constraints is the combination of a cost function with a threshold. Hence, if the cost of an edge collapse exceeds a certain threshold, it is marked as illegal and therefore omitted. The triangle mesh demo program offers two binary constraints of this kind as well as a third one, which follows a different approach. These three binary constraints are discussed below.

Roundness

As one might infer from the name, this binary constraint is based on the roundness criteria from the previous section and is modified only by including a threshold to classify an edge collapse as legal or illegal.

Normal Flipping

To decide whether a certain edge collapse shall be permitted, the normals of the affected faces before and after the collapse are compared. If the angle between two such face normals exceeds a predefined value, the edge collapse will not be performed. In the triangle mesh demo program the threshold angle was set to 90 degrees.

Independent Set

A very simple approach, which does without any kind of cost function, is given by the independent set binary constraint. After every collapse, the 1-ring neighborhood of the remaining vertex is locked, i.e., the corresponding vertices are marked to be unfit for participating in edge collapses. If an edge has a locked vertex, it will not be considered for a collapse anymore. Of course, this constraint is extremely restrictive and will not allow a high degree of mesh decimation.

The above listed binary constraints might seem useful at first glance, however, their influence on the quality of the decimation result is quite limited in practice. While the independent set constraint is to restrictive, one has to deal with finding appropriate thresholds for roundness or normal flipping.

1.4.4 Mesh decimation – The user interface

Figure 1.32 shows the user interface for the mesh decimation methods. Actions can be taken using the corresponding tool bar of the mesh demo program application window. Consider that there is an additional rendering option to visualize the collapsing edges of the decimation process. It can be found in the "Draw Vectors" rendering submenu under the entry "Collapsing Edges".

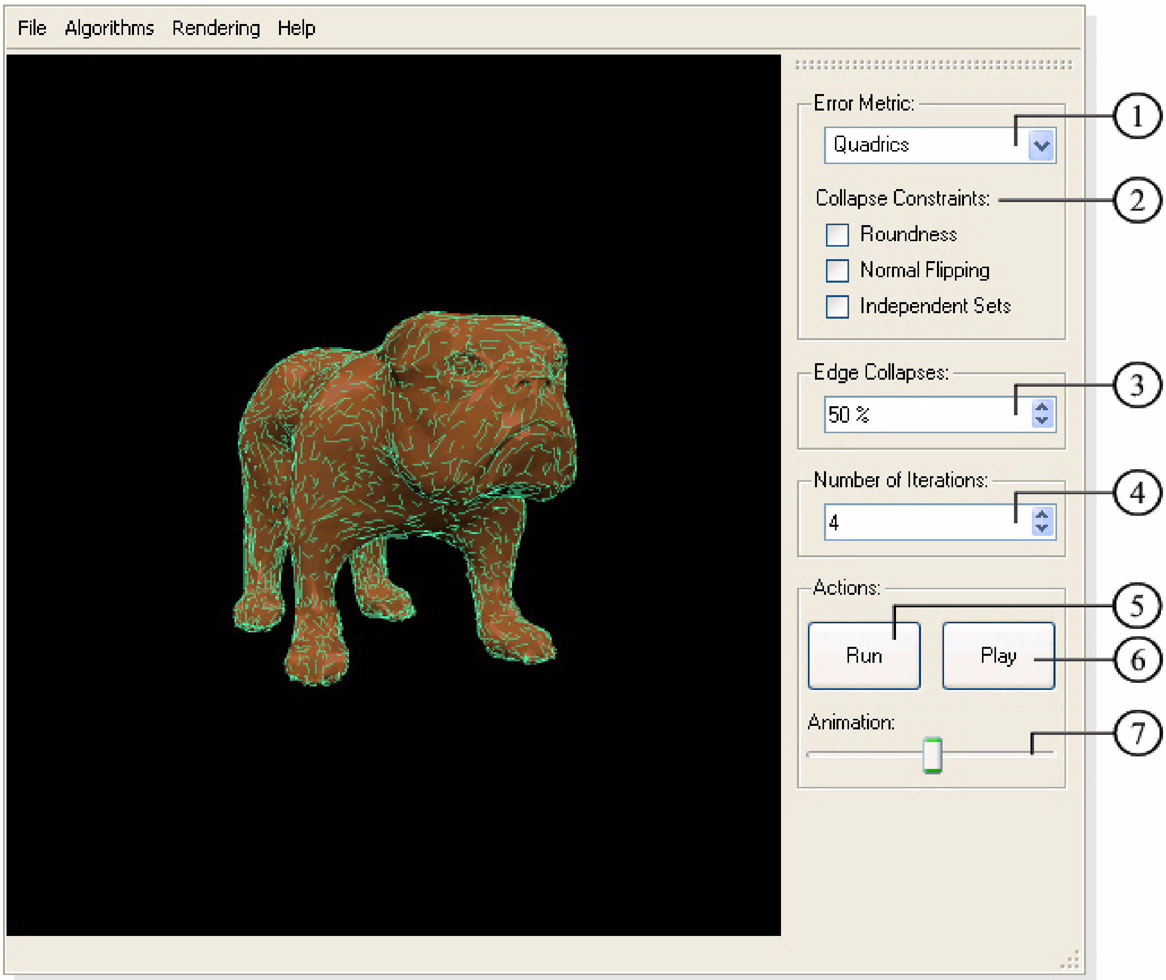


Figure 1.32: User interface for mesh decimation. Visualization of collapsing edges on the mesh model is activated.

①	Selection of the edge collapse cost function. The available options are the *quadric error metric* ("Quadrics") and the *roundness criteria* ("Roundness") (see sections 1.4.1 and 1.4.2).
②	Enable/disable the corresponding binary constraints on edge collapses (see Section 1.4.3).
③	Set the percentage of edges to collapse in every iteration.
④	Specify the number of eigenvalues/eigenvectors to approximate (only available when doing a partial decomposition).
⑤	Decimate the triangle mesh.
⑥	Show an animated transition between the mesh before and after the decimation step.
⑦	Slider to morph between the original and the decimated mesh by hand.

Table 1.6: Description of the mesh decimation tool bar.

1.5 Technical details

The triangle mesh demo program was written in C++ and developed with Microsoft Visual Studio .Net 2003. For the user interface design and implementation Trolltech's Qt library was used, the 3D visualization is based on OpenGL. The data structure to deal with triangular meshes was provided by OpenMesh, a very powerful and flexible tool, which already offered some of the subdivision algorithms and a couple of mesh decimation modules too. OpenMesh was developed by the Computer Graphics Group at the RWTH Aachen. Last but not least, the GNU Scientific Library, the Linear Algebra Package and the Arnoldi Package served as tools for algebraic computations.

The demo program runs under Windows XP, however, since the core libraries are operating system independent and available for other platforms, it should not be too hard building the demo program under Linux, for example.

A

References

[1] M. Desbrun, M. Meyer, P. Schröder, and A. Barr. "Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow." *SIGGRAPH 99*, p.317-324, 1999.

[2] I. Guskov, W. Sweldens, and P. Schröder. "Multiresolution Signal Processing for Meshes." *SIGGRAPH 99*, p.325-334, 1999.

[3] G. Taubin. "A Signal Processing Approach to Fair Surface Design." *SIGGRAPH 95*, p.351–358, 1995.

[4] I. Guskov. "Multivariate Subdivision Schemes and Divided Differences." Tech. rep., Department of Mathematics, Princeton University, 1998.

[5] M. H. Gross and A. Hubeli. "Eigenmeshes." Technical Report No. 338, Computer Science Department, ETH Zürich, 2000.

[6] M. Alexa. "Wiener Filtering of Meshes." Proceedings of the Shape Modeling International 2002, p.51, 2002

[7] N. Dyn, D. Levin, and J. A. Gregory. "A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control." *ACM Trans. Gr. 9*, 2 (April 1990), p.160–169.

[8] L. Kobbelt. "$\sqrt{3}$-Subdivision." *Computer Graphics* Proceedings, Annual Conference Series, 2000.

[9] C. Loop. "Smooth Subdivision Surfaces Based on Triangles." Master's thesis, University of Utah, Department of Mathematics, 1987.

[10] D. Zorin, P. Schröder, and W. Sweldens. "Interpolating Subdivision for Meshes with Arbitrary Topology." *Computer Graphics Proceedings (SIGGRAPH 96)*, p.189–192, 1996.

[11] D. Zorin, P. Schröder, T. DeRose, L. Kobbelt, A. Levin, and W. Sweldens. "Subdivision for Modeling and Animation." Course Notes (*SIGGRAPH 2000*).

[12] H. Hoppe. "Progressive Meshes." *SIGGRAPH 96*, p.99–108, 1996.

[13] M. Garland and P. S. Heckbert. "Surface Simplification Using Quadric Error Metrics." *SIGGRAPH 97*, p.209–216, 1997.

[14] E. W. Weisstein et al. “Biconjugate Gradient Method.” MathWorld, Wolfram Web Resource. http://mathworld.wolfram.com/BiconjugateGradientMethod.html

[15] D.C. Sorensen. “Implicitly restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations.”, In D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds. Parallel Numerical Algorithms, pages 119-166, Dordrecht, 1997, Kluwer.

[16] R. B. Lehoucq, D. C. Sorensen, C. Yang. “ARPACK Users’ Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.” SIAM: Philadelphia, PA, 1998.

[17] Arnoldi Package, http://www.caam.rice.edu/software/ARPACK/

[18] C-version of the Linear Algebra Package, http://www.netlib.org/clapack/