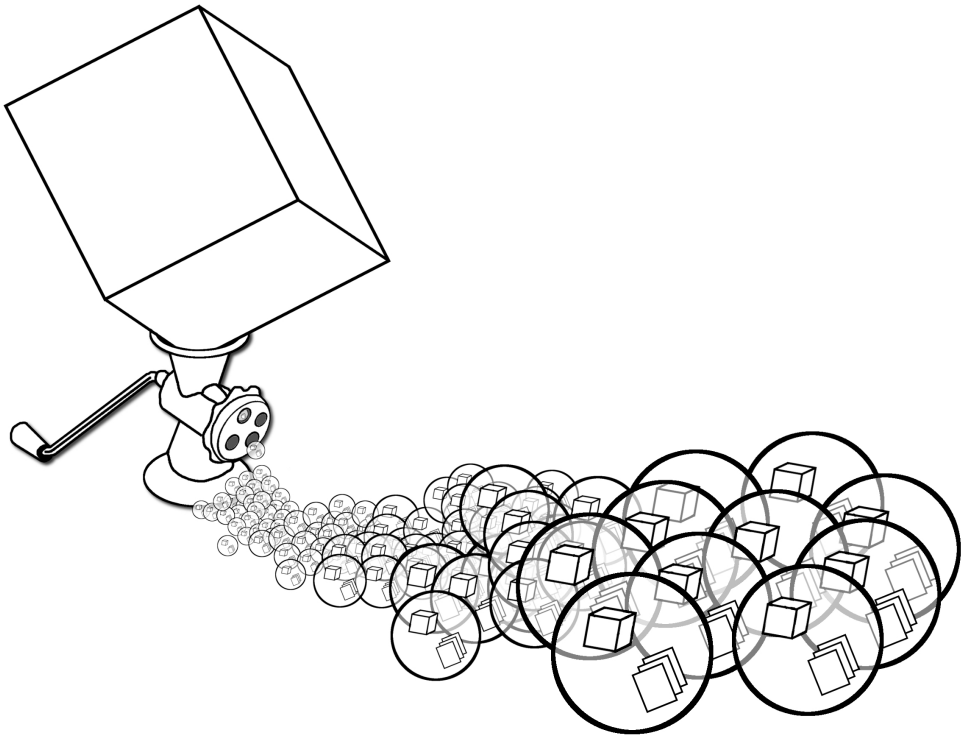
# Efficient and Parametrizable Edge-Collapses and Vertex-Splits



Master Thesis
13.10.2021

by David Schneeberger, 00-921-064

Supervisors:
Prof. Dr. Renato Pajarola
Lars Zawallich

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

University of Zurich[UZH]

VISUALIZATIONANDMULTIMEDIALAB

# Abstract

We present the implementation, and evaluation of a software library for efficient and parametrizable edge-collapses and vertex-splits. The library consists of a fast mesh data structure, decimation and refinement components and includes a set of modules that guide the decimation and refinement processes. The library is free of external dependencies and is modular in design. We investigate the choices we made during development, and give a detailed description of our implementation. We evaluate our implementation in terms of decimation quality and computational performance and compare it to other existing tools and libraries. Our evaluation demonstrates that both in terms of quality and performance our implementation can match or beat existing solutions.

# Contents

# 1 Introduction

The representation of 3-dimensional models in computers is widespread. They are used in many application areas, including the visualization and analysis of data, architectural and industrial design, and are common in film and game industries. With the increasing prevalence of 3-dimensional models, the tools and techniques that generate such objects continually evolve and, with them, the complexity of the models they produce. The storage and processing requirements to handle these objects can be significant. Despite advancements in computer hardware, highly detailed models often exceed the available capacity in many application areas. In order to provide a tradeoff between model size and quality, *mesh simplification* algorithms are used to reduce the complexity of objects.

**Mesh simplification** The simplification of polygonal meshes has been an active area of research for several decades [Cla76]. Research in this area has reached maturity, and several effective tools and algorithms are available. Mesh simplification describes a set of algorithms that reduce the number of elements in a polygonal mesh while approximating the original as well as possible. Figure 1.1 illustrates the simplification of an initial high detail mesh into a mesh that has less than 0.5% of the triangles but retains the overall shape of the original mesh.

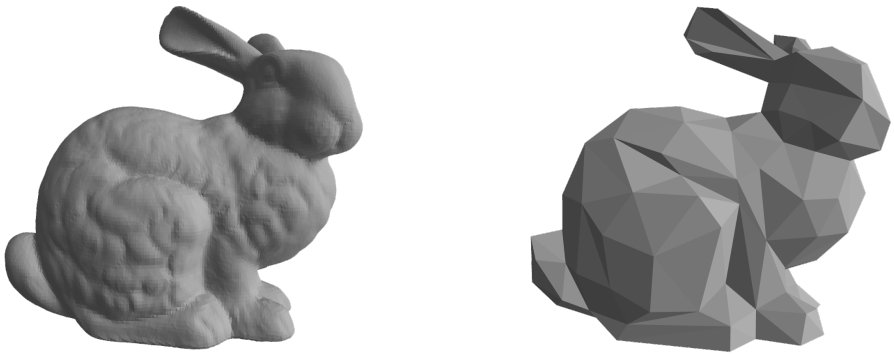


Figure 1.1: Simplification of a triangle mesh. The model on the left with 69 630 triangles is simplified to a mesh with 328 triangles on the right.

**Edge collapse** Several different approaches to mesh simplification exist. One such approach is to decimate a mesh by repeatedly collapsing one of its edges until a stopping criterion is satisfied. An *edge collapse* consists of removing an edge from the mesh and replacing its two endpoint vertices with a new vertex. In the process, the two triangles adjacent to the edge are eliminated from the mesh. The quality of the resulting mesh depends on two factors: First, in each iteration, a choice has to be made on which edge to collapse next. Second, the position of the new vertex has to be chosen.

**Vertex split** The edge collapse operator has an inverse operation called the *vertex split*. A vertex split involves removing a vertex from the mesh and inserting two vertices connected by an edge in its place. In doing so, two triangles are added to the mesh. Thus, the edge collapse operator simplifies the mesh, and the vertex split operator adds detail to it.

**Progressive Refinement** The incremental nature of the decimation algorithm and the ability to undo the effects of its operators has the nice property in that it allows the continuous refinement of a model by removing

or adding detail to it. The approach, also known as continuous level of detail, precomputes a list of refinement operations, which can then be applied in a separate run-time phase.

**Selective Refinement** A related approach is to refine a mesh by adding more detail to certain areas while removing detail in others. This approach has been widely studied in the context of view-dependent refinement, where the desired levels of detail are determined by the position and direction of a virtual camera.

## 1.1 Contributions

This thesis aims to design and implement efficient and parametrizable algorithms that utilize edge collapses and vertex split operations to simplify and refine triangle meshes.

As part of this thesis, we have developed the C++ library Pmesh, which we evaluate and compare to other existing libraries. The libraries central design principle is the modularity of its components, allowing for independent parametrization of each stage of the mesh decimation and refinement process. At the centre of the library is a fast mesh data structure, based on the directed edge structure [CKS98].

## 1.2 Overview

The remainder of this thesis is organized as follows:

- In Chapter 2 we discuss the background and related work
- Chapter 3 contains a detailed description of the algorithms we use and the choices we made for our implementation.
- In Chapter 4 we describe the implementation and interface of each of the components of the library
- In Chapter 5 we compare our algorithm to existing solutions both in terms of quality and performance
- Finally in Chapter 6 we summarize the conclusions of our work.

# 2 Related Work

## 2.1 Mesh Simplification

Mesh simplification algorithms aim to reduce the complexity of a given by transforming it into another mesh with fewer triangles. Quality criteria usually control the simplification procedure intending to preserve the shape or other properties of the mesh as much as possible. A wide range of algorithms for mesh simplification have been proposed and can be roughly categorized into two categories:

**Vertex clustering** This approach computes the bounding of a mesh and divides it into a number of cells. All vertices within one cell are replaced by a representative vertex [RB93]. Triangles that have become degenerate as a result are subsequently removed from the mesh. While the approach can be very fast, it may cause low-quality simplifications.

**Inremental decimation** These algorithms iteratively remove one vertex and two triangles from the mesh. The order in which the operations are applied is determined by a cost function.

## 2.2 Incremental Decimation

The iterative mesh decimation algorithms remove one vertex from the mesh at a time. At each step, the best candidate for removal is determined by a user-defined cost function. The basic removal is performed by applying a topological operator.

### 2.2.1 Topological Operators

The three main operators that perform a decimation step are as follows [Hor15]:

- The *vertex removal* operator deletes one vertex and retriangulates the resulting hole.
- The *edge collapse* operator joins two neighbouring vertices $\boldsymbol{p}$ and $\boldsymbol{p}$, by collapsing the edge between them and moving them to a new position $\boldsymbol{r}$ [Hop96].
- The *halfedge collapse* operator moves a vertex $\boldsymbol{p}$ to the position of one of its neighbours $\boldsymbol{q}$. The halfedge collapse operator can be considered a special case of the edge collapse operator where the new vertex position $\boldsymbol{r}$ coincides with $\boldsymbol{q}$.

Edge collapses, and halfedge collapses can only be performed when the topological constraints discussed in Section 3.2.1 are met. If those criteria are satisfied, all of the above operators preserve the mesh topology, i.e. no holes in the original mesh will be closed, and no connected components will be eliminated [BKP+10].
Each of the above operators has an inverse operation as shown in Figure 2.1. For the edge collapse operator, the vertex split, and for the halfedge collapse, the restricted vertex split. The inverse operations can be used to revert the changes to the mesh and thereby allow running a decimation scheme backwards by inserting successively more detail into the mesh [BKP+10].

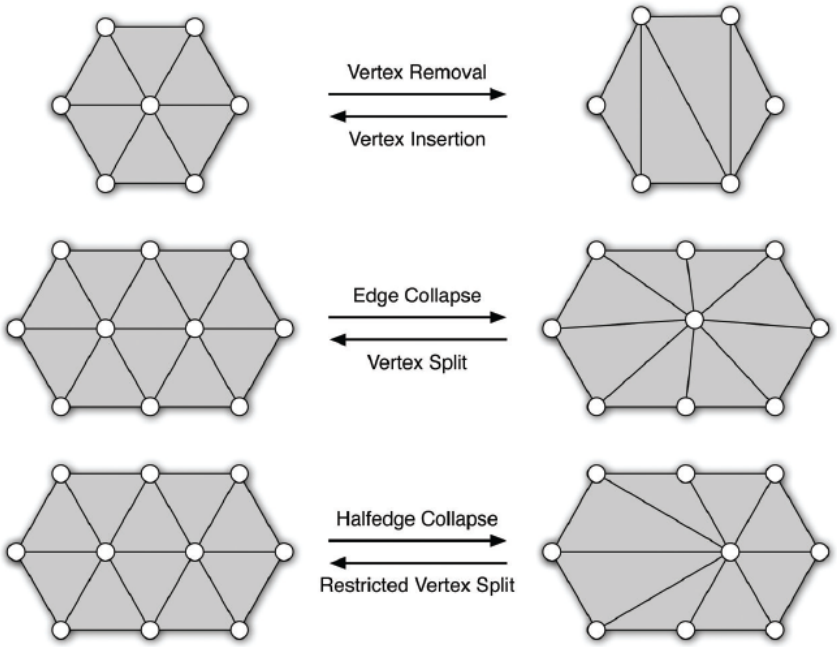



Figure 2.1: Topological operators and their inverses [BKP+10].

### 2.2.2 Cost function

In order to determine the best candidate vertex or edge for removal, a cost function is evaluated to assign a cost to each possible operation. The cost function is commonly defined to compute an approximation error of the simplified mesh. The approximation error can be defined as a distance measure that computes the distance of a triangle in the simplified mesh to its corresponding sub patch in the original mesh. It may also measure the visual differences between renderings of the original mesh and its simplified version. Apart from the approximation error, other so-called *fairness criteria* may also be used to guide the ordering of operations. For example, the criteria could include penalties if a triangle flips, i.e., the normal before and after the collapse changes by more than a certain amount.

A widely used distance measure in the context of mesh decimation is the *Quadric Error Metric* [GH97]. It works by assigning distance values to each vertex of the decimated mesh. The distance values are computed as the sum of squared distances from the planes formed by each triangle surrounding a vertex. The metric can be computed efficiently and only requires the storage of a symmetric $4 \times 4$ $\mathbf{Q}$ matrix per vertex. The Quadric Error Metric is discussed in detail in Section 3.4.1.

## 2.3 Progressive Meshes

The *progressive mesh* algorithm introduced in [Hop96] is based on the idea that a mesh is not stored in its original form but rather as a drastically simplified base mesh and a list vertex splits that can restore the detailed mesh. Figure 2.2 shows an example of the same mesh represented at different levels of detail. The leftmost image represents the coarse base mesh, consisting of only a few triangles. To the right, more triangles are added into the mesh through vertex splits operations until the fully detailed mesh is restored in the rightmost image.

In a preprocessing stage, the contruction of a progressive mesh starts with initial high-detail mesh $\hat{M} = M^n$ which is transformed into a drastically simplified mesh $M^0$ by applying a sequence of $n$ successive edge collapses:

$$(M = M^n) \xrightarrow{coollapse_{n-1}} \ldots \xrightarrow{collapse_1} M^1 \xrightarrow{collapse_0} M^0$$

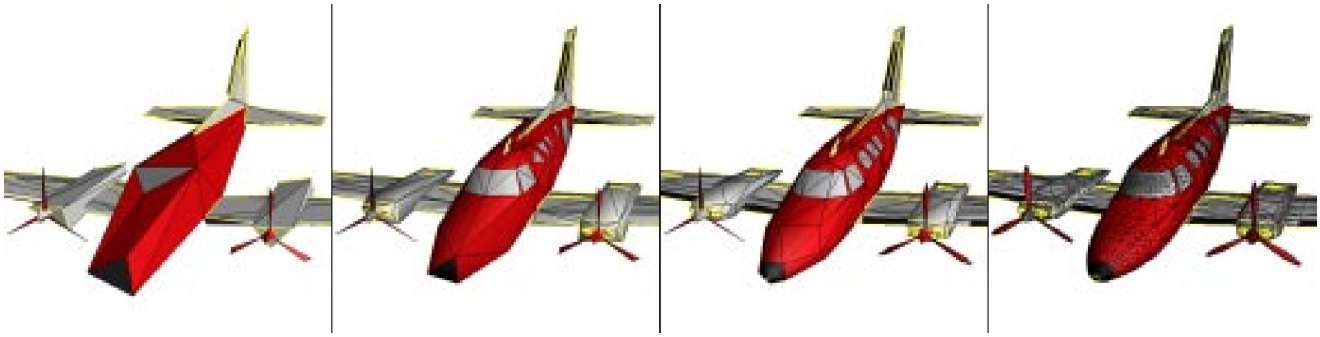


Figure 2.2: Progressive mesh at different levels of detail. [Hop96]

The detailed mesh can be successively restored by applying the transformations in reverse order as their inverse vertex split operations:

$$M^0 \xrightarrow{split_0} M^1 \xrightarrow{split_1} \ldots \xrightarrow{split_{n-1}} (M^n = M)$$

Hoppe [Hop96] refers to the notation $(M^0, split_0, \ldots, split_{n-1})$ as the *progressive mesh representation* of $M$. Each vertex split is encoded as $split(s, l, r, t, A)$. As illustrated in Figure 2.3a), the variables $s$ and $t$ refer to the endpoints of the collapsed edge, $l$ and $r$ refer to the cut vertices of the split, and $A$ denotes attribute information of the affected vertices, including the positions of the end vertices $\boldsymbol{v}_s$ and $\boldsymbol{v}_t$.

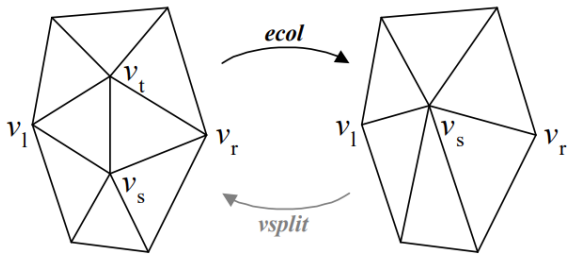



Figure 2.3: Edge collapse transformation [Hop96]

## 2.4 Selective Refinement

The algorithm that was described in 2.3 allows the transformation of an entire mesh into different levels of detail. For certain applications, it is, however, desirable to refine only certain areas of a mesh. One such widely used application is real-time rendering, where only the visible areas of a mesh require greater detail at a single point in time. Other areas that are not directly visible to the camera should be as coarse as possible in order to save processing resources.

One approach to add detail only in desired areas of a mesh would be to iterate through the list of vertex splits of a progressive mesh and split only those vertices where additional detail in the vertexes neighbourhood is desired [Hop96]. However, there exists a dependency among the different operations. A vertex split cannot be performed unless the corresponding vertex is currently part of the mesh. Skipping a vertex split prevents splitting the vertices it would generate and, in turn, all of their ancestors. This dependency among the vertices can represented through a binary forest of vertices, referred to as a *merge tree* [XESV97] or *vertex hierarchy* [Hop97].

In a vertex hierarchy illustrated in Figure 2.4, the root nodes correspond to the vertices in the coarsest mesh $M^0$, and the leaf nodes are the vertices of the original mesh $\hat{M}$. The two children of a vertex $v$ are the vertices $v'$ and $v''$ that are generated by splitting $v$. The arcs connecting the parent and child represent the dependencies that impose

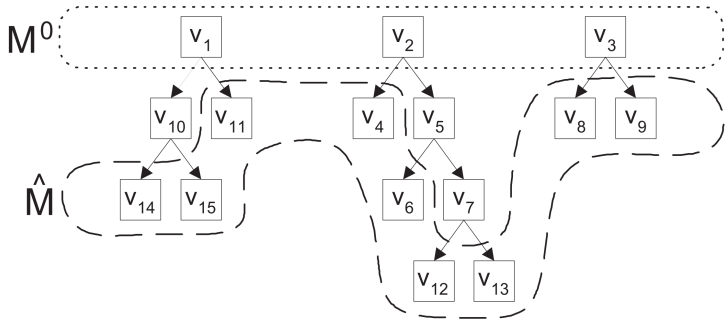



Figure 2.4: Vertex hierachy. [Hop97]

a partial ordering on the operations. Every possible cut across the forest represents a different simplification of the original mesh. The nodes below this front represent vertices that have been removed from the mesh as a result of an edge collapse. The nodes on or above the front represent the vertices that are part of the current mesh.

The vertex hierarchy is constructed in a preprocessing stage, in which the edge collapses used to simplify a mesh are recorded, and the parent-child relations are set. At run-time, the nodes that currently lie on the front are tested against specific criteria to determine if they should be collapsed or split. Each edge collapse or vertex split operation moves the front up or down in the hierarchy.

In [Paj01] an optimization to the vertex hierarchy referred to as a *halfedge collapse hierarchy* is presented. Recognizing that the leaf nodes of a vertex hierarchy do not contain any information required for an edge collapse or vertex split, the number of nodes in the hierarchy can be reduced by half. In the halfedge collapse hierarchy, each node represents a halfedge collapse instead of a vertex, as illustrated in Figure 2.5.

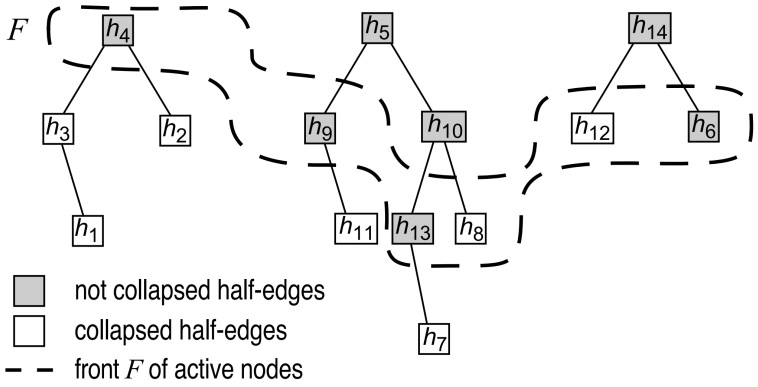



Figure 2.5: Half-edge collapse hierarchy. [PD04]

### 2.4.1 Neighbourhood Dependencies

The dependencies represented by the vertex hierarchy are not in general sufficient to guarantee that an operation will not cause nonmanifold connectivity in a mesh or cause triangles to fold over. Several approaches to address this problem have been proposed in the literature:

- In [XV96] the one-ring neighbours of each vertex generated as a result of an edge collapse in the preprocessing stage are explicitly stored. At run-time, an edge collapse or vertex split is only valid if the recorded neighbour vertices exist and are adjacent in the current simplification. If any vertex does not exist in the current mesh, additional edge collapse or vertex split transformations are necessary to activate the vertex.

- In [Hop97] the four faces adjacent to the two collapsed triangles are stored in the preprocessing stage and used as a condition at run-time.
- In [ESV99] vertex numbering is used to implicitly test for the conditions formulated in [XV96].
- In [KL01] the concept of *fundamental cut vertices* is introduced to allow the arbitrary reordering of vertex splits while still guaranteeing proper mesh connectivity. Edge collapses are restricted only by their topological constraints.
- In [Paj01] vertex splits require that the four faces adjacent to the two collapsed faces are active in the mesh. If they are not active, additional vertex splits are performed. The approach does not require any additional storage as the required information is already present in the employed mesh data structure. As in [KL01] edge collapse are only restricted by their topological constraints.
- In [HSH09] the storage cost to test for the conditions in [Hop97] is reduced by storing the maximum ID of the vertices that create the required triangle faces.

The less restricted approaches as formulated in [KL01] and [Paj01] allow for drastic differences in the level of detail between neighbouring areas of a mesh. This can cause undesirable fold-overs of some faces of the mesh, however, since the faces adjacent to a vertex may be different to what they were during the preprocessing stage.

## 2.5 Mesh Data structures

The efficiency of algorithms operating on a mesh depends in large part on the underlying data structure used to represent the mesh. A wide variety of mesh data structures have been proposed [SB11]. In order to evaluate different data structures, we have to take into account topological and algorithmic considerations. Following [BPK+08] we formulate the following criteria for evaluating the data structure used in our implementation:

- Time to answer adjacency queries, such as finding neighbouring vertices, faces, or edges.
- Time to perform vertex split and edge collapse operations
- Time to construct the data structure
- Memory efficiency and redundancy

In the following, we outline some of the common types of data structures used for the representation surface meshes and finally draw a comparison.

### 2.5.1 Face Based Data Structures

The *triangle list* data structure directly stores 3 vertex positions for each triangle in the mesh and represents the simplest way to store a surface mesh.

The *indexed face set* data structure avoids the redundancy of the triangle list by storing an array of vertices along with a separate array that contains sets of indices that encode the triangles in the mesh. The structure is widely used in many 3D file formats and rendering APIs such as OpenGL.

The face-based data structures do not explicitly store any information about neighbouring faces or vertices and, thus, require expensive searches whenever adjacency information is required.

### 2.5.2 Edge Based Data Structures

Edge-based data structures explicitly store connectivity information between the edges, faces and vertices of a mesh. They are therefore suited for efficient traversal of a mesh. Some well known examples include the *winged edge structure* [Bau72] and *quad-edge* [GS85] structures.

### 2.5.3 Halfedge Based Data Structures

Halfedge based data structures split each edge in a mesh into two oriented halfedges. They can thereby avoid case distinctions needed in edge-based structures when traversing a mesh. The halfedges are oriented in counter-clockwise order around each triangle and along mesh boundaries. The halfedge stores the references to encode the mesh connectivity information as illustrated in Figure 2.6;

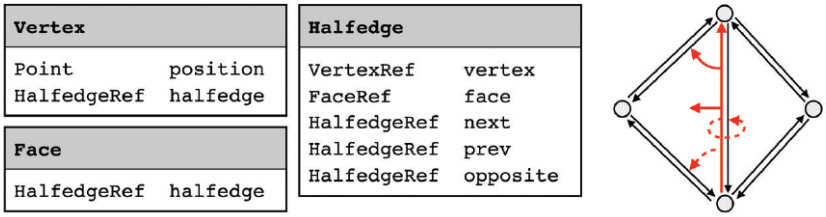



Figure 2.6: Connectivity information stored in a halfedge-based data structure. [BKP+10]

### 2.5.4 Directed Edge Structure

The directed edge data structure is a variant of the halfedge based data structures specifically designed for triangle meshes. It is particularly memory-efficient because it encodes some of the connectivity information implicitly by grouping the three halfedges belonging to a triangle contiguously in memory. We discuss the direct edge data structure in detail in Section3.1

### 2.5.5 Comparison

Table 2.1 contains an overview of the data structures and lists their memory requirements. The values for the memory consumption are taken from [BKP+10] and assume 12 bytes to store a vertex position and, on average, two faces per vertex. The directed edge structure has the lowest memory consumption out of the data structures that feature efficient adjacency queries. The data structure furthermore allows efficient updates to the mesh connectivity as part of an edge collapse or vertex split as shown in Section3.1. We have therefore chosen to base our implementation on the directed edge structure.

| Data Structure | Adjacency Queries | Memory Consumption |
|---|---|---|
| Triangle List | $O(V)$ | 72 bytes/vertex |
| Indexed Face Set | $O(V)$ | 36 bytes/vertex |
| Winged Edge | $O(1)$ | 120 bytes/vertex |
| Halfedge | $O(1)$ | 144 bytes/vertex |
| Directed Edge | $O(1)$ | 64 bytes/vertex |

Table 2.1: Comparison of mesh data structures.

# 3 Technical solution

## 3.1 Mesh Data Structure

### 3.1.1 Directed Edge Structure

We decided to base our mesh data structure on the directed edge structure [CKS98], because of its low memory requirements, its efficient adjacency queries and local modifications. We base our implementation in large part on the description in [Paj01].

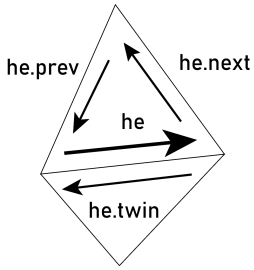



Figure 3.1: Directed edge data structure.

The directed edge data structure is based on the concept of halfedges. Each halfedge $he$ stores an index to its starting vertex $he.vtx$ and an index to its reverse twin halfedge $he.twin$. The halfedges for each triangle are stored consecutively in counterclockwise order as illustrated in Figure 3.2. This memory layout thereby implicitly encodes the connectivity between the three halfedges belonging to a triangle.

The data structure requires no additional information to store the mesh connectivity. The memory requirement is thus only 8 bytes per halfedge or 24 bytes per triangle. The vertex data is kept in a separate buffer and accessed using the vertex index stored for each halfedge. The authors of [CKS98] use an additional array to store a reference from each vertex to an outgoing halfedge. We have found no need for these additional references.

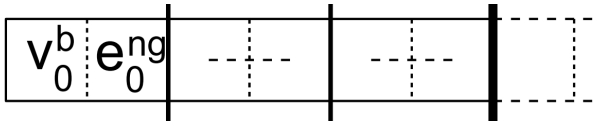



Figure 3.2: Memory layout of the directed edge structure. [CKS98]

Given that the halfedges of a triangle are stored consecutively, the indices of the three halfedges belonging to a face $f$ can be retrieved as follows

$$halfedge(f, i) = 3f + i, \quad i = 0, 1, 2 \tag{3.1}$$

The face of a halfedge with index $h$ and its index within that face are thus given by

$$face(h) = h/3, \quad \text{and} \quad faceindex(h) = h \; mod \; 3 \tag{3.2}$$

The previous halfedge $he.prev$ and next halfedges $he.next$ of halfedge $he$ with index $h$ can be determinded with the following expressions (in C-style pseudocode)

$$\begin{aligned} prev(h) &= \texttt{(h\%3 == 0) ? (h+2) : (h-1)} \\ next(h) &= \texttt{(h\%3 == 2) ? (h-2) : (h+1)} \end{aligned} \tag{3.3}$$

Rotating a halfedge $he$ one step around its starting vertex $he.vtx$ in counterclockwise (CCW) and clockwise (CW) order is done by

$$\begin{aligned} rotateCCW(he) &= he.prev.twin \\ rotateCW(he) &= he.twin.next \end{aligned} \tag{3.4}$$

Despite its low memory requirements, the data structure can conveniently answer neighbourhood queries using the functions presented above. As an example, enumerating the one-ring vertex neighbours around a halfedges starting vertex $he.vtx$ can be done by the code segment shown in Listing 3.1. The function sets the vertex indices $idx[0], ..., idx[n-1]$ of the $n$ neighbouring vertices.

```
1 n = 0
2 the = he
3 do
4   idx[n++] = the.next.vtx
5   the = the.prev.twin // rotate CCW
6 while the != he
7 return n
```

Listing 3.1: Enumerating one-ring neighbours

### 3.1.2 Edge Collapse

As illustrated in Figure 3.3 An edge collapse of halfedge $he$ involves connecting two halfedges $a$ and $b$, and $c$ and $d$ respectively. In order to connect the halfedges, their reverse twin halfedges are set to reference each other. Updating the mesh connectivity for the collapse of halfedge $he$ for triangles A and B is given by

$$\begin{aligned} he.prev.twin.twin &= he.next.twin \\ he.next.twin.twin &= he.prev.twin \end{aligned} \tag{3.5}$$

and for the triangles $B$ and $C$

$$\begin{aligned} he.twin.next.twin.twin &= he.twin.prev.twin \\ he.twin.prev.twin.twin &= he.twin.next.twin \end{aligned} \tag{3.6}$$

An edge collapse does not alter the data of the halfedges that are part of the removed triangles. They are simply disconnected from the rest of the mesh and should be ignored when rendering or converting the mesh to another format. In order to keep track of the removed triangles in the mesh, we use a separate array that stores a binary flag for each triangle to indicate whether it is collapsed.

Collapsing an edge, additionally involves assigned the new target vertex index for each of the halfedges incident to the new vertex. This is done by rotating around the vertex, starting from one of the incident halfedges from triangles $A$, $B$, $C$ and $D$ in Figure 3.3 ($he.prev.twin$, $he.next.twin.next$, $he.twin.next.twin.next$, or $he.twin.prev.twin$). The update can be done as shown in the code segment in Listing 3.2

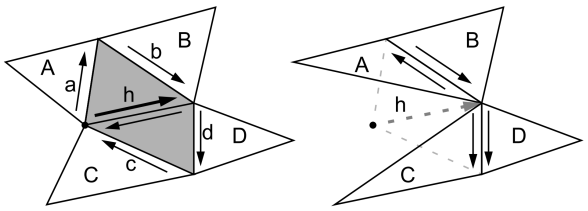



Figure 3.3: Updating connectivity for edge collapse and vertex split. [Paj01]

```
start = he.prev.twin
the = start
do
  the.vtx = newVtx
  the = the.prev.twin // rotate CCW
while the != start
```

Listing 3.2: Updating the vertex after an edge collapse

### 3.1.3 Vertex Split

A valuable property of the directed edge structure is that the deactivated triangles of the collapsed halfedge and its reverse twin halfedge are not altered. To perform a vertex split of a collapsed halfedge, we can therefore use the halfedge entries of the two deactivated triangles to perform the necessary updates. The connectivity of the triangles A and B in Figure 3.3 can be restored by

$$\begin{aligned} he.prev.twin.twin &= he.prev \\ he.next.twin.twin &= he.next \end{aligned} \tag{3.7}$$

and for the triangles $B$ and $C$

$$\begin{aligned} he.twin.next.twin.twin &= he.twin.next \\ he.twin.prev.twin.twin &= he.twin.prev \end{aligned} \tag{3.8}$$

As is the case for the edge collapse operation, a vertex split involves updating the vertex indices after the connectivity has been reassigned. The updates involve rotating around the start-vertex of the restored halfedge $he$ and setting each halfedges vertex index to $he.vtx$. Additionally, the end-vertex $he.next.vtx$ is restored by rotating around $he.next$. The update procedure for the start-vertex is shown in Listing 3.3.

```
the = he.prev.twin
while the != he
  the.vtx = he.vtx
  the = the.prev.twin // rotate CCW
end
```

Listing 3.3: Reassigning the starting vertex after a vertex split

The two vertices $vl = he.twin.prev.vtx$ and $vr = he.prev.vtx$ (shown in Figure 3.3 are referred to as the *cut vertices* for the vertex split of $he$ [KL01]. In the case of selective refinement, in which the vertex splits are not carried out in the reverse order of the edge collapses, the vertices $he.twin.prev.vtx$ and $he.prev.vtx$ have to be assigned the cut vertices in the current mesh, as the vertex indices of the surrounding halfedges may have been altered by other operations while the edge was collapsed. The update can be done as follows

$$\begin{aligned} he.prev.vtx &= he.next.twin.vtx \\ he.twin.prev.vtx &= he.twin.next.twin.vtx \end{aligned} \tag{3.9}$$

### 3.1.4 Boundaries

We handle mesh boundaries y defining a specific value to be considered invalid (in our case, the constant `UINT32_MAX`). In order to test if a given halfedge $he$ lies on the mesh boundary, it is sufficient to test if its twin halfedge $he.twin$ is set to an invalid value. In order to accommodate for the possibility of invalid values, some adjustments to the aforementioned functions have to be made.
Rotation around a vertex $v$ starting from a given halfedge $he$ has to consider that an invalid halfedge may be encountered and then stop the rotation. If the rotation is incomplete, the procedure is subsequently carried out in the opposite direction. The modified function including the two rotations to reassign the start vertex $he.vtx$ after a vertex split is shown in Listing 3.4. We adjust the function to assign a new vertex after an edge collapse in the same manner.

```
1  the = he.prev.twin
2  while isValid(the) && the != he
3    the.vtx = he.vtx
4    the = the.prev.twin // rotate CCW
5  end
6
7  if the == he
8    return  // no boundary encountered
9
10 if !isValid(he.twin)
11   return // no rotation in CW direction possible
12
13 the = the.twin.next
14 while the != he
15   the.vtx = he.vtx // rotate CW
16   if !isValid(the.twin)
17     return
18   the = the.twin.next
19 end
```

Listing 3.4: Setting a vertex with boundary test

We modify the edge collapse operation of halfedge $he$ test that the halfedges for which the reverse twin is reassigned are valid. If $he$ is itself a boundary edge, the reverse side cannot be updated. The modified procedure for updating the connectivity is shown below in Listing 3.5.

The inverse vertex split operation is modified analogously by testing the validity of twin halfedges before assignments are made.

### 3.1.5 Nonmanifold meshes

The directed edge structure is limited in regards to the representation of nonmanifold meshes.

**Nonmanifold vertices** The surface around *nonmanifold vertices* consists of multiple otherwise manifold components that touch each other at exactly one vertex. Nonmanifold vertices can be represented without modifications to the structure; however, they do not form a connection between otherwise unconnected triangles. While the halfedge connectivity remains consistent for all edges touching the vertex, the single point connection between the individual manifold components may be removed. When edge collapses do not involve repositioning

```
1 if isValid(he.prev.twin)
2   he.prev.twin.twin = he.next.twin
3 if isValid(he.next.twin)
4   he.next.twin.twin = he.prev.twin
5
6 if isValid(he.twin)
7   if isValid(he.twin.next.twin)
8     he.twin.next.twin.twin = he.twin.prev.twin
9   if isValid(he.twin.prev.twin)
10    he.twin.prev.twin.twin = he.twin.next.twin
```

Listing 3.5: Edge collapse with boundary conditions

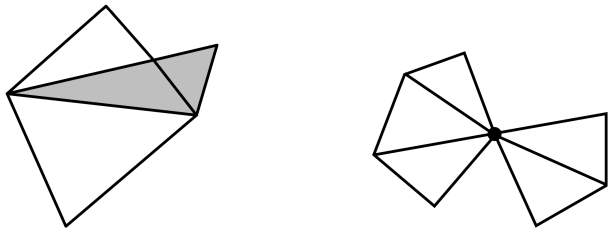



Figure 3.4: (a) Nonmanifold edge. (b) Nonmanifold vertex. [LRC+03]

of a vertex (i.e. halfedge collapse, see [Paj01], [PD04]), nonmanifold vertices present no further issue in regards to the data structure. If the vertex is repositioned as a result of an edge collapse, the new position may be chosen incorrectly in regards to the other components touching the vertex since the new position can only be chosen by examining the triangles connected by edges.

We have chosen to handle the issue of nonmanifold vertices by using "virtual" vertex indices for triangles that share a nonmanifold vertex $v$. Each connected group of triangles that share $v$ and that is unconnected to any triangle of another group is assigned a virtual index $\hat{v}$. As a result, the mesh no longer has any nonmanifold vertices, and the individual groups have no connection in common. We maintain the list of virtual vertices in a separate array that we can use to look up the original vertex $v$, which we can then use as an index into the vertex buffer. When we need to modify the positions of the virtual vertices, we can convert them to "real" vertices by increasing the size of the vertex buffer and copying their vertex data into the buffer.

**Nonmanifold edges** The directed edge structure provides no way to handle *nonmanifold edges*, which are edges shared by more than two triangles. The connectivity cannot be represented in the structure, and thus, the edges cannot be correctly traversed. In [CKS98] an approach to address this problem is presented that uses the sign-bit of halfedge indices to indicate if an edge is nonmanifold. Negative indices represent nonmanifold edges. By removing the sign-bit, the index points to a separate array that stores a list of connected halfedges. However, the approach seems impractical when performing local mesh modifications and would likely require reordering and reindexing of the array. We have thus chosen a more straightforward approach in our implementation by converting nonmanifold edges to unconnected manifold edges during the construction of the structure, as explained in Section 3.1.6. The conversion of these edges generates nonmanifold vertices, which can be dealt with as explained above.

### 3.1.6 Construction

We build the directed edge structure with an index buffer containing indices to vertices and one or more buffers containing the data for each vertex (such as position and vertex normal) as input. During construction, we apply

a set of checks and transformations to ensure that each triangle has unique vertices and produce a mesh with manifold connectivity. As a result, the produced mesh may not be identical to the input mesh. The mesh may contain multiple unconnected surfaces, and we do not attempt to connect those surfaces or split the mesh into multiple meshes.

In order to find and connect each halfedges reverse twin halfedge, we sort a temporary array containing an entry for each halfedge of the struct shown in Listing 3.6. The sorting comparator is chosen in such a way that neighbouring halfedges will be adjacent in the sorted array. If the input data contains nonmanifold edges, those will be adjacent before any potential twin halfedges. This allows us to easily detect nonmanifold edges and handle them appropriately. After sorting, we iterate over the array and assign the twin connectivity in the array of halfedges contained in the data structure.

```
1 struct SortEntry {
2   enum State { None, Reverse, Done };
3   uint32_t halfedge;
4   uint32_t vtxLow;
5   uint32_t vtxHigh;
6   State    state;
7 };
```

Listing 3.6: Helper struct find connected halfedges.

The fields `vtxLow` and `vtxHigh` in Listing 3.6 are assigned the minimum and maximum values of the index of each of the halfedges endpoints. The field `state` is used to specify the direction of the halfedge and is used to track if a halfedge has already been assigned.

In order to detect nonmanifold vertices, we maintain a separate array of integers that tracks the number of halfedges that have a particular vertex as their starting vertex. In the first iteration, over all of the halfedges, the counter for each halfedges starting vertex is incremented. After connectivity has been assigned, we iterate over all halfedges once more and check if their one-rings match the counter for the vertex. We maintain a flag to avoid redundantly checking the same vertex more than once. If the counter and the size of the one-ring do not match, the vertex is nonmanifold. We can then generate a new vertex ID and assign it to all connected halfedges by rotating around the vertex.

### 3.1.7 Edges

A drawback of the directed edge structure is the lack of an explicit representation of edges. An edge is shared by two connected halfedges or a single boundary halfedge. The structure provides no way to enumerate or access edges in the mesh directly. For our purposes, we have found that a direct reference to edges is beneficial during the mesh decimation phase to determine the cost and order of the edge collapses. We have thus augmented the data structure to include directly accessible edge indices. By maintaining edge indices, we can avoid redundant cost updates.

We have added edge references to the data structure by maintaining two additional arrays that allow the lookup of an edge index given a halfedge index and vice versa. The edge-to-halfedge table only includes a reference to one of the two halfedges of each edge. The other is accessible by looking up its reverse twin halfedge. The two arrays add an additional requirement of 4 bytes per halfedge plus 4 bytes per edge. If the edge references are only used during the preprocessing phase, the two buffers could be discarded thereafter and would not require any memory during the run-time phase.

We update the edge references after each modification of the mesh. Following an edge collapse shown in Figure 3.5a, the edge of halfedge $b$ is assigned to $a$ and the edge $d$ is assigned $c$. The original edges of $a$ and $c$ are deactivated. The edge information for the two deactivated triangles remains unmodified in the process.

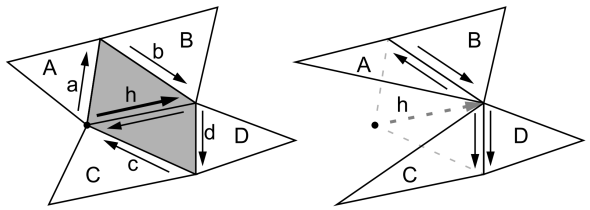



Figure 3.5: Edge updates after an edge collapse. [Paj01]

After a vertex split, we apply the inverse operation by reassigning the original edges to $a$ and $c$ as shown Figure 3.5 b. The reassignment of an edge to a halfedge includes updating both lookup tables (edge-to-halfedge, halfedge-to-edge) to ensure that both tables have valid entries.

## 3.2 Constraints

An edge collapse operation may potentially introduce inconsistencies or degeneracies into the mesh. In order to avoid such undesirable transformations, several constraints have to be met for an operation to be considered valid. We can differentiate between *topological constraints*, which preserve the overall topology of the mesh, and *soft constraints* that improve the overall quality of the generated mesh.

### 3.2.1 Topological Constraints

#### Edge Collapse Constraints

To preserve the genus and overall manifold connectivity of the mesh, Hoppe et al. formulate three constraints for edge collapse transformations [HDD$^+$93]. An *edge collapse* of edge $(v_1, v_2)$ is topologically valid if and only if the following criteria hold:

1. *Boundary constraint*: If $v_1$ and $v_2$ are boundary vertices, then the edge $(v_1, v_2)$ has to be a boundary edge.
2. *Intersection constraint*: The intersection of the one-ring neighbourhoods of $v_1$ and $v_2$ consist of the vertices opposite the edge $(v_1, v_2)$ only. In other words, the intersection contains exactly two vertices in the case of a non-boundary edge and exactly one vertex in the case of a boundary edge.
3. *Empty constraint*: The surface has more than 4 vertices if neither $v_1$ nor $v_2$ are boundary vertices, or more than 3 vertices if either $v_1$ or $v_2$ are boundary vertices.

**Boundary constraint** The collapse of two boundary vertices through a non-boundary edge leads to a non-manifold vertex, as shown in Figure 3.6. For a halfedge $he$ with vertices $(v_1, v_2)$ to be collapsed, we check the *boundary constraint* efficiently by examining the boundary flags of the two vertices and testing if the halfedges reverse twin $he.twin$ is valid.

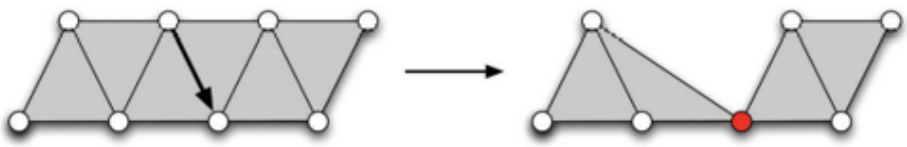


Figure 3.6: Nonmanifold vertex generated as result of an edge collapse. [BKP$^+$10]

**Intersection constraint** Figure 3.7 shows the result of an edge collapse where the *intersection constraint* is not met. The neighbourhoods of $v_1$ and $v_2$ share more than two vertices, leading to a nonmanifold edge created due to the collapse.

In order to test for the intersection constraint, we can choose one of two approaches:

1. We rotate around $v_1$ and $v_2$ and add the one-ring vertices of both into an array. We subsequently sort the array and check if the array contains duplicate entries.
2. Another approach requires that we have to keep an array of binary flags, which we can manipulate for each vertex in the mesh. We rotate around $v_1$ and set the flag for each one-ring vertex to true. We then rotate around $v_2$ and check if any of its vertices have been set to true. If a duplicate vertex is not one of the opposite vertices of the edge $(v_1, v_2)$ the constraint is violated. We then need to perform a third rotation around $v_1$ and reset the flags.

In our implementation, we tested both approaches and found no significant difference in speed between the two.

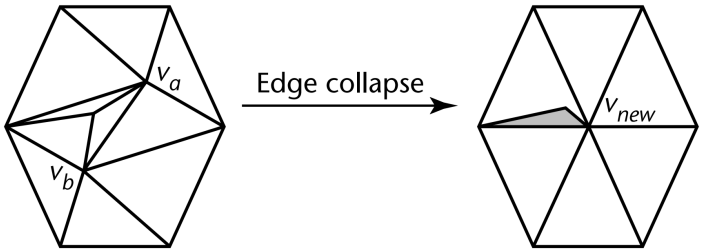



Figure 3.7: Nonmanifold edge generated as result of an edge collapse. [LRC+03]

**Empty constraint** The third constraint, which we refer to as the *emtpy constraint* prevents single unconnected triangles and triangles folded onto each other from being eliminated by an edge collapse. We can test for the constraint as shown Listing 3.7.

```
1 if (he.twin && he.prev.vertex == he.twin.prev.vertex) return false;
2 if (he.twin && he.twin.next.boundary && he.twin.prev.boundary) return false;
3 if (he.next.boundary && he.prev.boundary) return false;
```

Listing 3.7: Testing for the empty constraint.

### Vertex Split Constraints

A *vertex split* transformation is always legal, as it can never change the topological type of a mesh [HDD+93]. However, Kim et al.[KL01] note in the context of selective refinement, that when the two cut vertices $vl$ and $vr$ (illustrated in Figure 3.4) are the same, a degenerate case may occur. If we enforce the vertex split, in this case, the resulting mesh will not be simply connected.

## 3.2.2 Soft Constraints

An edge collapse can cause undesirable effects, even when the collapse is topologically valid. To address such problems, we have implemented additional checks to evaluate a proposed edge collapse. We can either add a penalty factor or disallow the transformation entirely when failing one of these checks. If undesirable edge collapses are only penalized, they may be still be performed if all other transformations have significant penalties. It does ensure, however, that that the algorithm can still progress even when all contractions are considered "bad" [Gar99].

**Triangle flips** One such unwanted side effect are *mesh foldovers* [XESV97] or *triangle flips*, in which the mesh gets a folded crease. In Figure 3.8 the collapse of the edge $(v_a, v_b)$ to the vertex $v_{\text{new}}$ causes a foldover due to the new triangle $(v_d, v_c, v_{\text{new}})$. A popular way to detect this is to measure the change in normals of the corresponding triangles before and after the edge collapse. A mesh foldover happens when the angle between the normals is greater than a user-defined threshold.

We have chosen to also implement the approach described by Garland [Gar99] to detect foldovers. For every triangle around $v_a$, excluding the two triangles shared with $v_b$, there is an edge opposite $v_b$. By placing a plane through the edge, perpendicular to the triangle, the position of $v_{\text{new}}$) must lie on the same side of the plane as the vertex $v_a$. The same applies to the triangles surrounding $v_b$. We have implemented this check by comparing the two dot-products of $(\boldsymbol{v}_a - \boldsymbol{v}_b)$ and $(\boldsymbol{v}_{\text{new}} - \boldsymbol{v}_b)$ with the normal of the plane. If the second product is smaller than the first product multiplied by some factor, the triangle is considered to be flipped.

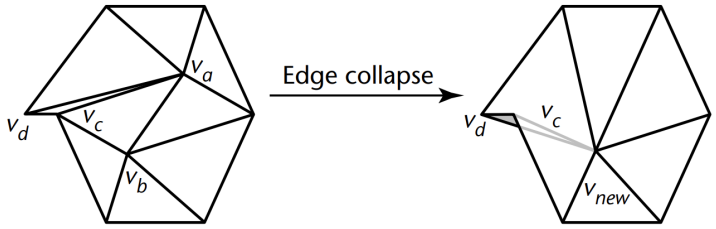



Figure 3.8: An edge collpase which causes a mesh foldover. [LRC+03]

**Sliver Triangles** Another potentially unwanted side effect of edge collapses is the generation of *sliver triangles*. These are skinny triangles with an interior angle close to 0. Slivers are not usually a problem in rendering. However, they can pose problems for some algorithms, such as finite element analysis, and can lead to instabilities in numerical simulations [BPK+08]. We can quantify the compactness of a triangle with area $a$ and lengths of the three sides $l_0$, $l_1$, and $l_2$ with the following formula as proposed by Guéziec [Gué96]:

$$\gamma = \frac{4\sqrt{3}a}{l_0^2 + l_1^2 + l_2^2} \tag{3.10}$$

A compactness of 1 corresponds to an equilateral triangle and 0 to a triangle whose vertices are colinear. We can penalize or discard edge collapses whose compactness fall below some threshold.

## 3.3 Decimation Algorithm

The decimation algorithm performs repeated edge collapses until no further collapses are possible or until a stopping criterion is reached. Each halfedge is initially assigned a cost that is incurred by its collapse. The cost of an edge collapse operation quantifies the change it would cause to the mesh. Hence it is also referred to as the error associated with the operation. Starting with an initial high detail mesh, the operation with the least cost (that is valid) is applied to the current mesh. Collapsing an edge may cause the cost of neighbouring halfedges to change. The costs of the affected neighbours thus have to be reevaluated after each step. In order to maintain the order of the operations after cost updates, the candidate halfedges are kept in a priority queue.

Before applying an edge collapse, we test the topological and other constraints of the collapse, as discussed in Section 3.2. If the edge collapse is not considered to be valid, we assign a maximum cost value to it and update its position in the queue. The halfedge is not be removed from the queue since the violation of constraints may be temporary. If the halfedges neighbourhood changes as a result of another edge collapse, its cost will get reevaluated. When it reaches the top of the queue again, its validity will be tested anew.

When the top element in the queue has the maximum cost value associated with it, the end of the procedure is reached since all other remaining elements will also have maximum cost. This, in turn, means that all remaining candidates are invalid and cannot be collapsed. Algorithm 1 shows the pseudocode of the decimation procedure.

**Algorithm 1:** Decimation algorithm

```
Input: original mesh M = M̂, stopping criterion stop
Output: coarse mesh M^0
P ← empty priority queue
foreach halfedge he in M do
    cost ← compute collapse cost of he
    Insert(he, cost) in P
while P not empty and stop not reached do
    he ← top element of P
    if he is removed then
        Remove (he) from P
    else if he has maximum cost then
        return;
    else if collapse if he is not valid then
        cost ← maximum cost
        Update(he, cost) in P
    else
        Apply edge collapse of he on M
        Remove he from P
        foreach halfedge he_i affected by the collapse do
            cost ← compute collapse cost of he_i
            Update(he_i, cost) in P
```

**Edge entries** As discussed in Section 3.1.7, we augmented the directed edge data structure to include direct edge references. This allows us to use edge indices instead of halfedges as entries in the queue. We can thereby reduce the size of the queue as well as the number of updates required. In this setup, two opposite halfedges share a single cost entry. In the case of a halfedge collapse, however, the cost of collapsing one halfedge is not the same as the cost of collapsing its reverse twin halfedge. We can solve this problem by evaluating the edge collapse cost in such a way that the lower of the two costs returned. Additionally, we store which of the two halfedges has the lower cost.

This distinction is only necessary for certain types of edge collapses. More specifically, it depends on how the new vertex is placed after an edge collapse. If a vertex is placed at the midpoint of an edge, for example, the collapse cost will be the same for both halfedges. We designed the Cost and Vertex Placement modules as separate entities that are passed to the decimating module as C++ template parameters. This means that we can evaluate at compile if two separate cost evaluations are needed. Even in the case of the halfedge collapse, which requires two cost evaluations, the use of edge references is advantageous because fewer updates to the queue are needed.

**Heap** We implemented the priority queue as a binary heap data structure. For each edge, we maintain a handle to its corresponding heap position in the heap. For every update to the heap, the handle is updated to point to the correct location. We can thereby update the cost of an edge without having to first remove it.

### 3.3.1 Lazy Cost Evaluation

A variant of the decimation procedure is presented in [Coh99]. In order to reduce computation time, the algorithm performs *lazy evaluation* of edge costs. After an edge collapse, the costs of the affected edges are not recomputed. Instead, the edges are marked as *dirty* by setting a flag. When a dirty edge appears at the top of the queue, its cost is recalculated, the edge is marked as clean and reinserted into the queue. The approach ensures that if the cost of an edge has increased as a result of an edge collapse, it will not be applied before its cost has been recomputed. On the other hand, if the collapse cost has decreased as a result of an edge collapse, it will go unnoticed.

We adapted this approach to our implementation by making a slight adjustment: Some edges in the queue can at any have a maximum cost value assigned, which means that they could not legally be collapsed, the last time they were checked. When such an edge is affected by an edge collapse, we always update its cost. Since we are using edges instead of halfedges for the queuing, we also have to update the edges of the removed triangles, i.e., the edges `he.next.twin.edge` (=`he.prev.twin.edge`) and `he.twin.prev.twin.edge` (=`he.twin.next.twin.edge`).

### 3.3.2 Independent Sets

The *independent sets* approach [DFMP97] [XESV97] is motivated by the desire to build flat vertex hierarchies for view-dependent refinement. The algorithm performs a maximal set of independent operations, .i.e., operations whose neighbourhoods do not overlap. Hierarchies are thereby built one level at a time. The independence criterion can formulated as follows: [PR00]

- For each edge $e_1 = (v_1, v_2)$ that will be collapsed and any edge $e_2 = (w_1, w_2)$ forming a quadrilateral $(v_1, v_2, w_1, w_2)$ with $e_1$, $e_1$ and $e_2$ cannot be collapsed in the same batch

Since neighbouring halfedges cannot be collapsed in the same batch, there is no need to maintain a dynamic queue. The cost values can be maintained in an array and recomputed once in each batch. The array is sorted at the beginning of each batch.

We have implemented the approach as follows: After an edge collapse, we flag the four vertices belonging to the two removed triangles as locked. When iterating over the sorted array, we test if any of the four triangle vertices of the candidate halfedge are locked and if so, we move on to the entry.

## 3.4 Cost and Error Metrics

The cost and error metrics are what determine the ordering of edge collapse operations in the decimation algorithm and thereby determine the quality of the resulting simplification. We implemented the cost functions used in the decimation algorithm as independent modules. A cost module is passed to the decimation algorithm as a templated instance, which enables function inlining and certain compile-time optimizations, as discussed in Section 3.3

The most basic cost function is the *edge length* cost, which simply returns the length of the candidate edge as a cost value.

In the following, we discuss our implementation of Quadric Error Metric.

### 3.4.1 Quadric Error Metric

Every vertex can be seen as an intersection of the planes of its adjacent triangles. Garland [Gar99] defines a quadric $Q$ as a triple $Q = (\mathbf{A}; \boldsymbol{b}; c)$. $\mathbf{A}$ is a 3x3 matrix, b is a 3-vector, and $c$ is a scalar. The quadric error $Q(\boldsymbol{v})$ for every point in space $v$ is given by the second order equation

$$Q(\boldsymbol{v}) = \boldsymbol{v}^T \mathbf{A} \boldsymbol{v} + 2\boldsymbol{b}^T + c \tag{3.11}$$

In order to compute the sum of squared distances to a set of planes, only one Quadric is needed. It can be computed as the sum of the quadrics of each plane.

We have implemented the metric by making use of the fact that the matrix $\mathbf{A}$ is symmetric. We thus only need to store six floating-point variables for $\mathbf{A}$ or ten variables for the whole Quadric $Q$. We store three separate arrays for the triangles, edges, vertices in the mesh, which we initialize to zero. During initialization, we proceed as follows:

1. We calculate and store the quadric for each triangle in the mesh. Given a triangles normal $\boldsymbol{n}$, and its centroid $\boldsymbol{v}_c$ the quadric is initialized as follows:
   a) $\mathbf{A} = \boldsymbol{n}\boldsymbol{n}^T$
   b) $\boldsymbol{b} = -\mathbf{A}\boldsymbol{v}_c$
   c) $c = -\boldsymbol{v}_c\boldsymbol{b}$
2. The quadric of every triangle is added to the quadric of each of its three verices.
3. For each edge, the quadrics of its two vertices are added. Additionally, the quadrics for the two triangles adjacent to the edge are subtracted since they are included in both vertex quadrics.

To evaluate the error or cost of an edge collapse of a half-edge $he$ to a new vertex $v_{\text{new}}$, we lookup its edge quadric $Q$ and calculate the error $Q(v_{\text{new}})$ given by the equation 3.11.

After an edge collapse to a new vertex $v_{\text{new}}$, we update the quadrics of the affected triangles, vertices, and edges as follows:

1. Reset the quadric for the target vertex $v_{\text{new}}$ to zero.
2. The quadrics of each triangle incident to the target vertex are recalculated as outlined above. The resulting Quadric is added to the quadrics of each of the triangles the vertices while subtracting the previous Quadric of the triangle.
3. The quadrics of the two deactivated triangles are subtracted from the vertex quadrics of their vertices.
4. Finally, the quadrics of all affected edges are recalculated as above.

To achieve invariance of the specific triangulation of a given surface, Garland suggests weighting the triangle quadrics by each triangles area [Gar99]. We achieve this by scaling the triangle quadrics by the area of the triangles when adding them to the vertex quadrics.

In order to preserve the boundaries of a mesh, Garland and Heckbert place perpendicular planes running through the boundary edges of the mesh [GH97]. These constraint planes are then converted into quadrics, weighted by a significant penalty factor and added into the initial vertex quadrics for the edge's endpoints. Since the number of boundary edges is typically much smaller than the number of non-boundary edges, we have implemented the boundary quadrics using a hash table with the edge index as keys. We sum the boundary quadrics by rotating around boundary vertices and add the boundary quadrics for each boundary edge.

## 3.5 Vertex Placement

An edge collapse can be performed with different vertex placement strategies. In order to enable parameterization, we implemented the vertex placement strategy as separate modules. The vertex placement modules are passed as template parameters to a cost module. During decimation, the cost evaluation can be based on the position of the vertex that would get generated as a result of the collapse.

We handle the generation of new vertices differently, depending on whether the operations need to be invertible. If the decimation algorithm is used directly, then new vertices can be stored in place of the end-vertex of the collapsed halfedge. When the refiner components are used, operations need to be invertible, and the collapsed vertices need to remain accessible. For this purpose, we allocate an additional vertex buffer inside the mesh data structure to store the generated vertices. We handle the special case of the halfedge collapse differently. Since the edges collapse to one of their endpoints, a separate vertex buffer is not needed.

**Halfedge collapse** A simple but effective strategy is *endpoint placement* or *halfedge collapse*, where the newly placed vertex coincides with one of the endpoints of an edge or the end-vertex of a halfedge. It has the nice property that it is invertible without requiring extra storage.

**Midpoint placement** Another straightforward strategy creates the vertex at the midpoint of the edge. Midpoint placement, however, leads to a loss in volume in the model, and it has been shown that endpoint placement strategies can better preserve the features of the original model [LT99].

**Quadric optimal placement** *Quadric optimal placement* computes the target position of vertex so as to minimize its quadric error. We can obtain the target vertex by solving.

$$\boldsymbol{v} = -\mathbf{A}^{-1}\boldsymbol{b} \tag{3.12}$$

If the matrix is not invertible, we test the endpoints of the edge and return the position that has the smallest error. The optimal placement module requires the Quadric cost module.

## 3.6 Progressive Refinement

By recording the index of the halfedge that gets collapsed at each step in a decimation process, we obtain an array of all the operations performed. We can use this list to transform the mesh from one level of detail to another. We iterate over the array and successively apply each operation until we reach a target. The array can be traversed in either direction, and, depending on the direction, the operations are applied as edge collapses or vertex splits.

As detailed in Section 3.1.3, we store all the connectivity information required to perform a vertex split as part of the directed edge structure. There is, therefore, no additional storage needed to enable the transformation.

As pointed out in Section 3.5 however, depending on the configured vertex placement strategy, a separate vertex may need to be allocated.

## 3.7 Selective Refinement

As discussed in Section 2.4 a linear list of operations is not sufficient to enable selective refinement of a mesh because there exist dependencies between the individual operations. We thus need a way to model these dependencies. We have based our approach on the halfedge collapse hierarchy for view-dependent refinement described in [Paj01] and [PD04].:

In order to create a more general implementation that could be used for other applications, we separated the view-dependent parameters and tests from the core functionality. We implemented a separate view-dependent refiner that utilizes the general selective refiner.

### 3.7.1 Halfedge Collapse Hierarchy

The halfedge collapse hierarchy is a forest of binary trees where each node can be seen to represent a halfedge. Each node in the hierarchy contains a link to its parent, its left and right children and an index to a halfedge. A node can be seen as being collapsed or expanded depending on the state of its halfedge.

A node $t$ in the hierarchy can be collapsed if:

- none of its descendants have to be collapsed first
- the collapse is topologically correct

A node $t$ in the hierarchy can be split if:

- all of its ancestors have been split

- all four halfedges h.prev.twin, h.next.twin, h.twin.next.twin and h.twin.prev.twin are currently active in the mesh

A $front$ through the hierarchy defines a particular level-of-detail mesh. The nodes on the front are called $active$ nodes. A node $t$ is defined to be *active* if and only if the following two properties hold:

1. $t$ is not collapsed, and its child nodes are either collapsed or do not exist
2. $t$ is collapsed, its parent is not collapsed, and its sibling exists and is not collapsed

### 3.7.2 Construction

We construct the hierarchy by decimating a mesh using the approach described in Section 3.3.

In order to obtain a direct relationship between the vertices and the nodes in the hierarchy, we number the vertices that get generated in the decimation process in increasing order. If the original model has $n$ vertices with indices $0, 1, \ldots, n-1$, the first vertex generated will have index $n$. Each new vertex will have an index of one more than the previous vertex. We store the nodes in the hierarchy in a contiguous array.

Using this numbering, we can access a node given a vertex index as follows:

```
function vertexToNode(int vertex) {
  return vertex >= numVertices ? node[vertex - numVertices] : NULL;
}
```

After each edge collapse in the decimation process, we add a new node to the end of the array and set the child links as follows:

```
node.left = vertexToNode(halfedge.vertex)
if (node.left) node.left.parent = node

node.right = vertexToNode(halfedge.endVertex)
if (node.right) node.right.parent = node
```

### 3.7.3 Front

We implemented the front as a doubly-linked list. Each front node contains links to the previous and next node, as well as a link to a hierarchy node. The hierarchy will at any time have no more active nodes than there are leaf nodes in the hierarchy. We can thus preallocate an array of said length and initially set each node in the array to reference the next. In essence, we maintain two lists, the front nodes that are currently in use and the nodes that aren't. When a front node gets activated or deactivated, we remove it from one list and add it to the other, setting the previous and next links.

After parent-child links in the hierarchy have been established, we add all root nodes of the hierarchy to the front list.

### 3.7.4 Refinement

The refinement procedure iterates over the front and tests for each node if it should be split or collapsed. Vertex splits are performed as forced splits to ensure that the conditions for vertex splits described in Section 3.7.1 are met. If the required faces are not active in the mesh, the corresponding nodes are recursively force-split until the required configuration is achieved.

The edge collapse operations have to adhere to the topology constraints described in Section 3.2.1. We found that during view-dependent simplification, groups of nodes could fail to collapse due to topological constraints not being met. Figure 3.9 illustrates such a case. The horse model shows that groups of triangles are at a much higher detail level than the surrounding triangles. We were able to solve the issue by repeatedly performing additional vertex splits on those nodes that caused the intersection constraint to fail.

Another approach we took was to completely ignore the *intersection* constraint. This required that we handled the special cases in the mesh data structure itself in order to avoid generating holes in the mesh. We found that as long as the vertex splits and edge collapses were the exact inverses of each other, the refinement would continue working without problems. The resulting mesh may at times have nonmanifold edges but visually it does not exhibit any unexpected differences. We found, however, that this only works in the confines of the hierarchy during the run-time phase. When ignoring the topology constraints in the decimation phase, the hierarchy cannot correctly be built.

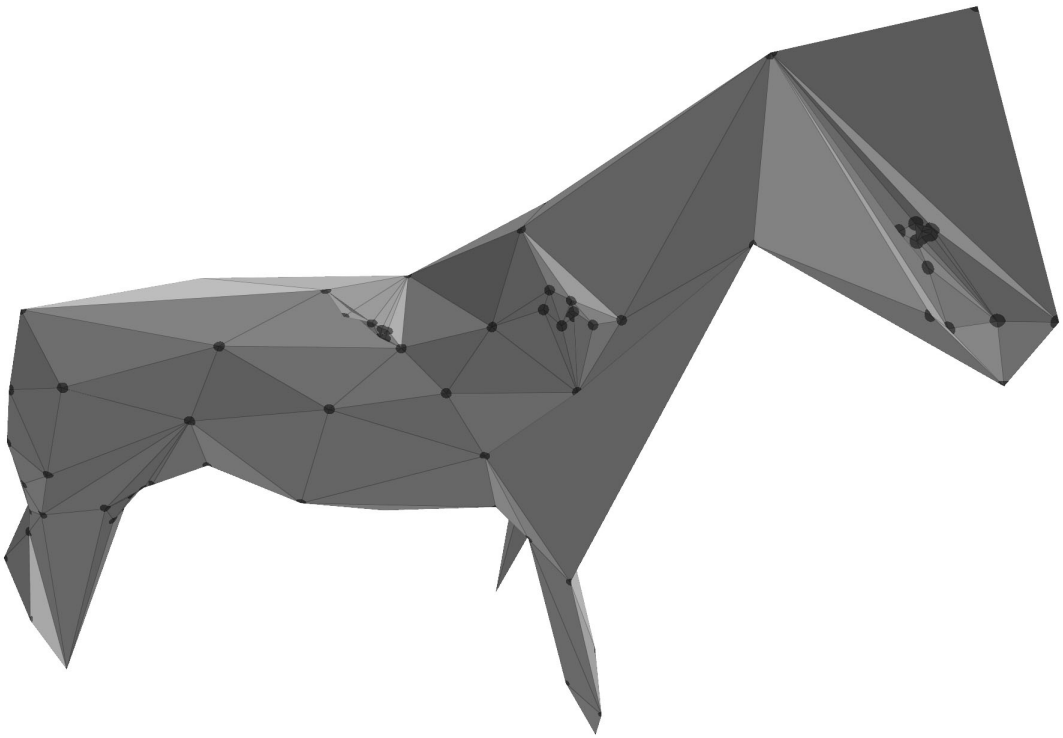


Figure 3.9: Regions of nodes failing to collapse .

### 3.7.5 Tree Balancing

We implemented two different approaches that aim to create more balanced hierarchies by modifying the cost term during mesh decimation.

In [Hop97] the cost metric of an edge collapse is modified by adding a small penalty factor. The penalty for the edge collapse $ecol(v_t, v_u)$ is $c(n_{v_t} + n_{v_u})$. Where $n_v$ is the number of descendants of $v$ and $c$ is a user-specified parameter. In [Gra02] a large penalty factor is described to reduce the number of hierarchies.

# 4 Implementation

In the following, we outline the most important aspects of the implementation of our library Pmesh and introduce the different components of the library. The library is implemented in C++ and requires a compiler with C++ 17 support. We have tested the library on Ubuntu Linux 21 with GCC 10.3 and Clang 12.0 compilers. On Windows, we used the Microsoft Visual Studio 2019 compiler, and on Mac, we tested with XCode 13 using the Apple version of the Clang compiler. The library is implemented without any external dependencies apart from the C++ Standard Library and is therefore easy to build and integrate. A CMake file for integration into other builds is provided. The library does not contain any file I/O functionality and, therefore, has to be paired with another solution to read and write meshes from 3D model files.

We use C++ class templates to enable parametrization of the functionality provided by the library. Custom modules can be used in conjunction with other components by implementing a predefined functional interface for certain module types. We decided on using templates instead of virtual interfaces mainly because we wanted to avoid the additional overhead associated with virtual functions and allow for certain types of compiler optimizations, such as function inlining. Templates are used only where needed to achieve low-level composability or to enable certain optimizations. However, the use of templates should not get in the way of using the library when no customized modules are required. Even though the library uses templates, the compilation times seem reasonable. Since much of the functionality is implemented inside of header files, it may be helpful to limit the inclusion of headers to a single C++ source file when using the library.

The library consists of the following main components:

- The halfedge-based `Mesh` class represents the mesh on which the other components operate.
- The `Decimator` module iteratively decimates a mesh by applying a series of edge collapses.
- A `Cost` module assigns a cost value to each possible edge collapse.
- The `Constraints` module checks for the validity of an edge collapse operation and can optionally penalize an operation by modifying its value.
- The `VertexPlacement` module defines where a new vertex will be placed as a result of an edge collapse.
- The *Linear Refiner* module records the edge collapses of a Decimator and successively allows to transform a mesh between various levels of detail at run-time.
- The `SelectiveRefiner` module builds an edge collapse hierarchy of recorded edge collapses and implements functionality to refine a mesh at run-time selectively.
- The `ViewRefiner` module uses the functionality provided by the Selective Refiner to provide view-dependent refinement of mesh.

The `Mesh`, `Decimator`, `LinearRefiner` `SelectiveRefiner` and `ViewRefiner` can output statistics of their operations to a separate structure, which can optionally be passed as a pointer when initializing the structures. Each of the Refiner modules can generate their output using the same functions as explained for the Mesh module in Section 4.1.1.

To aid in the development and testing of the library we also developed a viewer application and command-line tool, which are described in Sections 4.2 and 4.3.

## 4.1 Library Components

### 4.1.1 Mesh Class

The `Mesh` class stores the data of a mesh and provides functionality to query and manipulate the mesh structure. It is based on the directed edge data structure and forms the common basis for the other components.

#### Mesh Input And Output

In order to initialize the Mesh structure, the user is expected to provide an index buffer in the form of an indexed triangle list and one or multiple buffers containing the vertex data (such as position and vertex normal). The input, therefore, closely resembles the buffers used in exchange with 3D graphics APIs such as OpenGL or DirectX. The data types used in the buffers and the layout of the vertex data have to be described with additional helper structures. We intentionally chose not to explicitly store the vertex positions and normals in dedicated arrays to allow more flexibility and avoid unnecessary duplication and copying of data. By default, the Mesh class does not copy the vertex data and only references it. By specifying a flag during initialization or by calling the function `makeCopyOfVertexInputData` after initialization, the Mesh can be asked to copy the data and maintain ownership of the copy. Listing 4.1 shows a C++ code sample of how the Mesh structure can be initialized.

```
pm::VertexAccess vtxInput;
pm::IndexData    idxInput;

// describe the layout of the vertex data
vtxInput.m_layout.m_position.m_type   = pm::VertexAttributeType::Float3;
vtxInput.m_layout.m_position.m_offset = posOffsetInBytes;
vtxInput.m_layout.m_position.m_stream = 0;
vtxInput.m_layout.m_normal.m_type     = pm::VertexAttributeType::Float3;
vtxInput.m_layout.m_normal.m_offset   = normalOffsetInBytes;
vtxInput.m_layout.m_normal.m_stream   = 0;

// set the pointer, size and stride of the vertex data
vtxInput.m_data.m_numVertices        = numVertices;
vtxInput.m_data.m_numStreams         = 1;
vtxInput.m_data.m_stream[0].m_data   = (uint8_t*)vertexDataPointer;
vtxInput.m_data.m_stream[0].m_stride = vertexStrideInBytes;

// describe the index data
idxInput.m_data            = indexDataPointer;
idxInput.m_numIndices      = numIndices;
idxInput.m_indexBufferType = IndexBufferType::Uint16;

// initialize the mesh
pm::Mesh mesh;
mesh.initialize(idxInput, vtxInput);
```

Listing 4.1: Initialization of the Mesh structure.

At any point, after any number of edge collapse or vertex split operations, have been performed, the Mesh can generate output buffers with the same layout as specified during initialization. The user is expected to provide preallocated buffers of sufficient size. The required buffer sizes can be queried beforehand.

#### HalfEdgeIterator

A central element to our implementation of the Mesh class is the concept of iterators. The `HalfEdgeIterator` consists of a pointer to a Mesh and an index to a halfedge of that mesh. It provides most of the functionality needed to navigate around the mesh and access the data stored in vertices (e.g. enumerating one-ring vertices, accessing

neighbouring halfedges or calculating the normal of the triangle a halfedge belongs to). The iterator is defined so that multiple calls can be chained together, each returning a HalfEdgeIterator. If a halfedge does not exist because of a mesh boundary, an invalid iterator is returned. For example, rotating one step around the starting vertex in either direction can be done as follows:

```
HalfEdgeIterator he1 = prev().twin();  // CCW
HalfEdgeIterator he2 = twin().nextIf(); // CW
```

In addition to the HalfEdgeIterator, an EdgeIterator is also provided. The functionality of the EdgeIterator is, in essence, limited to returning a HalfEdgeIterator. Its primary use is to provide a single index to a pair of halfedges or one boundary halfedge.

#### Mesh Modification

Both structural iterator types only maintain a *const* reference to the mesh. Thus, to manipulate the mesh in any way, the functions provided in the Mesh class have to be used. The functions `edgeCollapse` and `vertexSplit` take a HalfEdgeIterator as parameter. The edge collapse function optionally takes a new vertex index as input. Otherwise, an edge will be collapsed to the end-vertex of the halfedge. Both functions do not perform any checks on the topological validity of the operation, and it is the responsibility of the user to perform the required checks. The function `checkCollapseTopologicalConstraints` may be used to test if an edge collapse is legal before executing it. The edge collapse function is designed not to fail if topological constraints are not met and can be undone in any case by applying its inverse vertex split operation. The vertex split function requires that the halfedges `prev.twin`, `next.twin`, `twin.next.twin`, `twin.prev.twin` are active, if they exist in the mesh. The functions `setTwin` and `setRemoved` allow modification of the mesh structure at a lower level. The data of a vertex can be modified using a `VertexIterator`. The vertex providers allow setting the position or normal of a vertex or access the raw data.

### 4.1.2 Decimator

The *Decimator* module performs iterative edge collapses on a given Mesh instance and can optionally be passed `DecimatorInterface` class as a template argument, which can record or react to the applied edge collapses. The order in which edge collapses are performed is determined by a cost module referenced by the template parameter `TCost`. A constraints module, referenced by the template parameter `TConstraints` determines if an edge collapse is legal and can optionally penalize certain edge collapses by modifying their assigned cost. Both modules, along with a Mesh instance, are passed to the Decimators constructor. The `decimate` function executes the decimation process, taking an optional `DecimationOptions` struct as a parameter. The options for the decimation process are as follows:

- `lazyEvaluateCost` specifies that the cost values in the priority queue are updated lazily, thereby improving the running time of the algorithm at the expense of losing some accuracy.
- `newVertexReplacesEndVertexData` specifies that the data of newly collapsed vertices is written in place of existing ones. If this option is not activated, a separate buffer for the generated vertices will be allocated. The mesh has ownership over the buffer.
- `newVertexGetsNewIndex` controls whether generated vertices are numbered using a new sequential index or if they are assigned the index of the end-vertex of the collapsed halfedge.
- `independentSets` specifies if the decimation process is performed using batches of largest independent sets of edge collapses, the algorithm uses array sorting instead of a priority queue based approach.
- In case the independent sets option is specified, the `batchSizeRatio` defines how many edges can be collapsed in one batch.

- The `minTriangles` option is given as a stopping criterion for the decimation process.

By default, the independent sets algorithm uses the C++ Standard Library `std::execution::par_unseq` policy to sort the array of cost values. This functionality was not available on the Mac during our testing and is therefore disabled on Apple platforms. The optional precompiler switch `PM_ENABLE_EXEC_PAR` can be used to override the default behaviour.

An instance of a class that wishes to react to the decimation process is passed to a Decimators constructor, denoted by the template parameter `TInterface`. The class has to implement the following function, which is called each time an edge collapse has occurred.

```
1     template <EdgeCollapseKind kCollapseKind>
2     void decimatorCollapse(HalfEdgeIterator he, uint32\_t mergeIndex, double cost);
```

The parameter `he` specifies the collapsed halfedge, `mergeIndex` is the index of the edge collapse and `cost` represents the assigned cost value. The template `kCollapseKind` specifies if the edge collapse is a full edge collapse or a halfedge collapse and will be the same for each successive call of the function.

### 4.1.3 Vertex Placement Module

The *Vertex Placement* and *Cost modules* are closely linked. The Vertex Placement module defines where a new vertex will be placed after an edge collapse. By default, it is provided as a template parameter to a Cost module. The Decimator only interfaces with the Cost module and contains no reference to the Vertex Placement module. An implementation of a custom Cost Module may therefore choose not to use a separate Vertex Placement module and implement all functionality inside the Cost class. In order to work with the provided Cost modules, a Vertex Placement class has to contain the function and `constexpr` statements shown in Listing 4.2.

```
1 struct VertexPlacementInterface
2 {
3   static constexpr EdgeCollapseKind kCollapseKind = EdgeCollapseKind::FullEdge;
4   static constexpr bool kDependsOnEdgeDirection = false;
5
6   template <typename TFloat>
7   Vec3T<TFloat> getCollapsePosition(HalfEdgeIterator he);
8 };
```

Listing 4.2: Interface of the Vertex Placement module.

The statement `kDependsOnEdgeDirection` indicates if the placement of vertex is different depending on the direction of the collapsed halfedge. It is used to optimize the common case, where the position is the same in either direction. `kCollapseKind` is similarly used for compile-time optimization in case of a halfedge collapse. The parameter should never be set to *true* except in the already provided `VertexPlacementEnd` module.

The library contains the following Vertex Placement modules:

- `VertexPlacementEnd` represents the special case of a halfedge collapse, which places the new vertex to coincide with the end-vertex of the collapsed halfedge. In contrast to other vertex placement strategies, no new vertex has to be created. Consequently, the additional storage requirement for new vertices can be avoided in the Refiner modules.

- `VetexPlacementMidpoint` places the new vertex at the midpoint of an edges vertice.

- `VertexPlacementLerp` linearly interpolates between the two edge endpoints, according to a user-specified parameter.

- `VertexPlacementQuadricOptimal` places the vertex at the position where the Quadric error is minimized and can only be used in conjunction with a Quadrics cost module. The module does not implement any functionality. Instead, the Quadrics cost module computes and returns the optimal position if this module is passed as a template parameter.

### 4.1.4 Cost Module

*Cost modules* are used by a Decimator to assign a cost to each possible edge collapse. An implementation of a Cost class has to provide the interface shown in Listing 4.3.

```
1 struct CostInterface
2 {
3   static constexpr EdgeCollapseKind kCollapseKind = VertexPlacementT::kCollapseKind;
4   static constexpr bool kDependsOnEdgeDirection = VertexPlacementT::
        kDependsOnEdgeDirection;
5
6   template <typename TFloat = float>
7   Vec3T<TFloat> getCollapsePosition(HalfEdgeIterator he);
8
9   double getCollapseCost(HalfEdgeIterator he);
10  void   edgeHasCollapsed(HalfEdgeIterator he);
11  void   initialize(Mesh& mesh);
12  void   release();
13 };
```

Listing 4.3: Interface of the Cost module.

The `kCollapseKind` and `kDependsOnEdgeDirection` statements as well as the `getCollapsePosition` function are typically provided by a Vertex Placement module, can however be defined in the Cost module itself.

The `getCollapseCost` function provides the main functionality of this module and returns a cost value given a halfedge. The other three functions can be empty but may be needed if the module maintains separate data structures. `edgeHasCollapsed` is called whenever an edge collapse has occurred. The function `initialize` is called before the decimation begins, and `release` is intended to free any allocated resources after the decimation process has ended.

The cost modules `CostShortestEdge` and `CostQuadrics` are provided as part of the library. The shortest edge cost module simply returns the edge length as a cost value. The `CostQuadrics` module implements the Quadric Error Metric. It can be configured with the following options:

- `recalculateVertexQuadrics` specifies if the vertex Quadrics will be fully recalculated after each edge collapse. This is more expensive but provides some additional accuracy.
- When `areaWeighted` is set, each triangle is weighted by its area.
- `constrainBoundaries` specifies if the Quadric of a perpendicular plane is added to each boundary edge in order to preserve mesh boundaries better.
- `boundaryWeight` specifies the weighting of the boundary planes.

### 4.1.5 Constraints Module

The Constraints module is used by the Decimator module and has two main purposes:

1. Determine if an edge collapse is legal. This typically involves testing the topological constraints of an edge collapse. It may also be used to formulate additional constraints where an edge collapse should be disallowed.

2. Penalize certain contractions by modifying their cost value.

An implementation of a Constraints module has to provide the two functions shown in Listing 4.4.

```
1 struct ConstraintsInterface
2 {
3   template <typename TCost>
4   double applyPenalties(HalfEdgeIterator he, double collapseCost, TCost& costs);
5
6   template <typename TCost>
7   AllowCollapse allowCollapse(Mesh& mesh, HalfEdgeIterator he, TCost& costs, std::vector
        <uint32_t>& scratchBuffer);
8 };
```

Listing 4.4: Interface of the Constraints module.

The `applyPenalties` function takes a cost value as input and returns a modified cost value. This is the actual value that the Decimator will use to determine the order in which it performs edge collapses. The `allowCollapse` function returns an enumeration type taking one of the three possible values:

1. `Yes`: The edge collapse is legal and can be applied in the current mesh.

2. `No`: The edge collapse is not legal in the current configuration but may become legal when the neighbourhood of the edge changes.

3. `Never`: The edge collapse is never allowed. The Decimator will remove this edge from the queue, and it will not be retested.

The provided default implementation of the Constraints module includes a number of different tests to be performed, and which can be configured by modifying the options field of type `ConstraintsOptions`. The penalties applied by the different constraints can be specified by two parameters that `scale` the cost value of an edge collapse or add a constant `bias`. The following tests are part of the default Constraints module:

- The *topological constraints* include the three topological constraints associated with an edge collapse. Options to disable each constraint are provided, but doing so is not supported in conjunction with the `SelectiveRefiner` module and will lead to undefined behaviour.

- Testing for *triangle flips* can be configured to either compare the normal deviation or to use the perpendicular plane approach.

- The test for *sliver triangles* can be configured with a threshold parameter

- The contraction *boundary vertices* and *boundary edges* can be penalized separately.

We have found that in order to implement additional constraints, it can be helpful to wrap the default Constraints module inside a new struct with the required signatures and calling the functions `applyPenalties` and `allowCollapse` of the default implementation from inside the new module. This approach is used in the `SelectiveRefiner` to add cost penalties that help balance the trees that form the edge collapse hierarchy.

### 4.1.6 Linear Refiner

The `LinearRefiner` records a set of edge collapses of a `Decimator` and then allows runtime-transformation of the mesh by applying the operations as vertex split or edge collapses to increase or decrease the level of detail of the mesh. Initialization of the LinearRefiner is done similarly to the Decimator module. The Decimator is instantiated from within the `initialize` function of the module and does not have to be provided separately. In addition to the `DecimationOptions` an option to update the vertex normals at runtime can be set. After initialization the desired level of detail can specified by using the functions `refineToMaxTriangleCount` and `refineToTriangleRatio`.

### 4.1.7 Selective Refiner

The `SelectiveRefiner` enables selective refinement of a mesh at runtime by building an edge collapse hierarchy and providing the functionality to perform edge collapses and vertex splits given the constraints of the hierarchy. It is intended to be used either as a base class or in conjunction with another class that specifies the criteria on how the mesh should be refined.

The edge collapse hierarchy is implemented in the separate `CollapseHierarchy` data structure. It is composed of two the two data structures `Tree` and `Front`. The Tree structure stores and maintains the forest of binary trees of edge collapses. A `TreeIterator` is used to access and traverse the individual tree nodes in the structure. The Front structure maintains the front of active nodes in the hierarchy. The `FrontIterator` structure is used to access the list nodes of the front and to retrieve the `TreeIterator` it references.

The SelectiveRefiner class has one template argument `TInterface` which implements two functions as shown Listing 4.5.

```
1 class SelectiveRefinerInterface
2 {
3   bool shouldRefine(CollapseHierarchy::TreeIterator node);
4
5   template <EdgeCollapseKind kCollapseKind>
6   void nodeAdded(CollapseHierarchy::TreeIterator node, HalfEdgeIterator
        collapsedHalfEdge);
7 };
```

Listing 4.5: Interface for the Selective Refiner.

The function `nodeAdded` is called during initialization of the structure, whenever a new node is added to the hierarchy. It gives the implementing class the possibility to build and prepare its own data. The function `shouldRefine` is repeatedly called during selective refinement and determines if a node should be collapsed or split. Selective refinement is initiated by calling the function `refine` which traverses the front of active nodes and refines the mesh according to the result of `shouldRefine`.

The implementing class is not restricted to refine the mesh in the context of the `refine` function and may choose a different approach to refine the mesh. A subset of the functions which the implementing class may call are the following:

- `forceSplitNode` splits a node by performing other vertex splits necessary. The function returns false if a node fails to split. The node passed to the function has to be splittable given the current state of the hierarchy. To test if a node can currently be split, the member function `canSplit` of `TreeIterator` may be used.
- The function `forceSplitAncestorsRecursive` force-splits a node by first splitting the nodes ancestors.

- `collapseNode` performs an edge collapse of a node. The passed node is expected to lie on the front of active nodes. All topological and other constraints have to be checked before calling this function.
- The function `isValidCollapseRuntime` tests the topological and neighbourhood constraints and returns true if the constraints allow the node to be collapsed.

In each case, the active node front is updated as part of the vertex split and edge collapse functions and does not have to be explicitly maintained by the caller.

A number of different options to the SelectiveRefiner can be specified by passing a `SelectiveRefiner Options` struct at initialization:

- `neighbourDependencies` specifies the neighbourhood constraints for edge collapse and vertex split operations.
- `treeBalancing` specifies the type of tree balancing performed by modifying edge collapse costs in the decimation phase.
- `reorderTriangles` specifies if the triangles in the mesh will be reordered. By setting this flag to true, the structure will be able to generate the output buffers by visiting the nodes in the hierarchy instead of iterating over each triangle in the mesh.
- `allowCollapseHelperSplits` specifies that additional vertex splits may be performed to enable the collapse of nodes that could otherwise not be collapsed because of topological constraints.
- `ignoreIntersectionConstraint` specifies that the intersection constraint for edge collapses will not be enforced during selective refinement. Enabling this option can lead to nonmanifold edges being generated in the output, but in turn, it ensures that all edge collapses can be carried out without the need for additional vertex splits.

### 4.1.8 View Refiner

The `ViewRefiner` inherits from the `SelectiveRefiner` to provide view-dependent refinement of a mesh. The function `refineToView` takes a `Camera` struct as input, which defines the view parameters. The Camera struct implements a convenenience function `fromModelViewProjection` to allow setting the parameters from with the commonly used view and projection matrices.

The ViewRefiner defines the following options in the ViewRefinerOptions struct, which is passed at initialization but may be updated at runtime:

- `exactViewParameterInitialization` specifies how the view-dependent error metric coefficients for each node will be generated. This parameter only has an effect during initialization.
- The remaining parameters specify which view tests will be performed during refinement and can be dynamically changed at runtime:
  - frustumTest
  - backfaceSimplification
  - preserveSilhouettes
  - shading
  - screenProjection
  - angularDeviationTolerance
  - projectedAreaErrorTolerance
  - silhouetteProjectedAreaErrorTolerance

## 4.2 Viewer Application

As an aid in the development and testing of the Pmesh library, we have developed a standalone cross-platform viewer application, shown in Figure 4.1. The program was developed in parallel to the library. The application allowed us to set most of the parameters of the library via a graphical user interface and helped us see the effects in real-time. We also used the application to run various performance tests, including those conducted with external libraries.

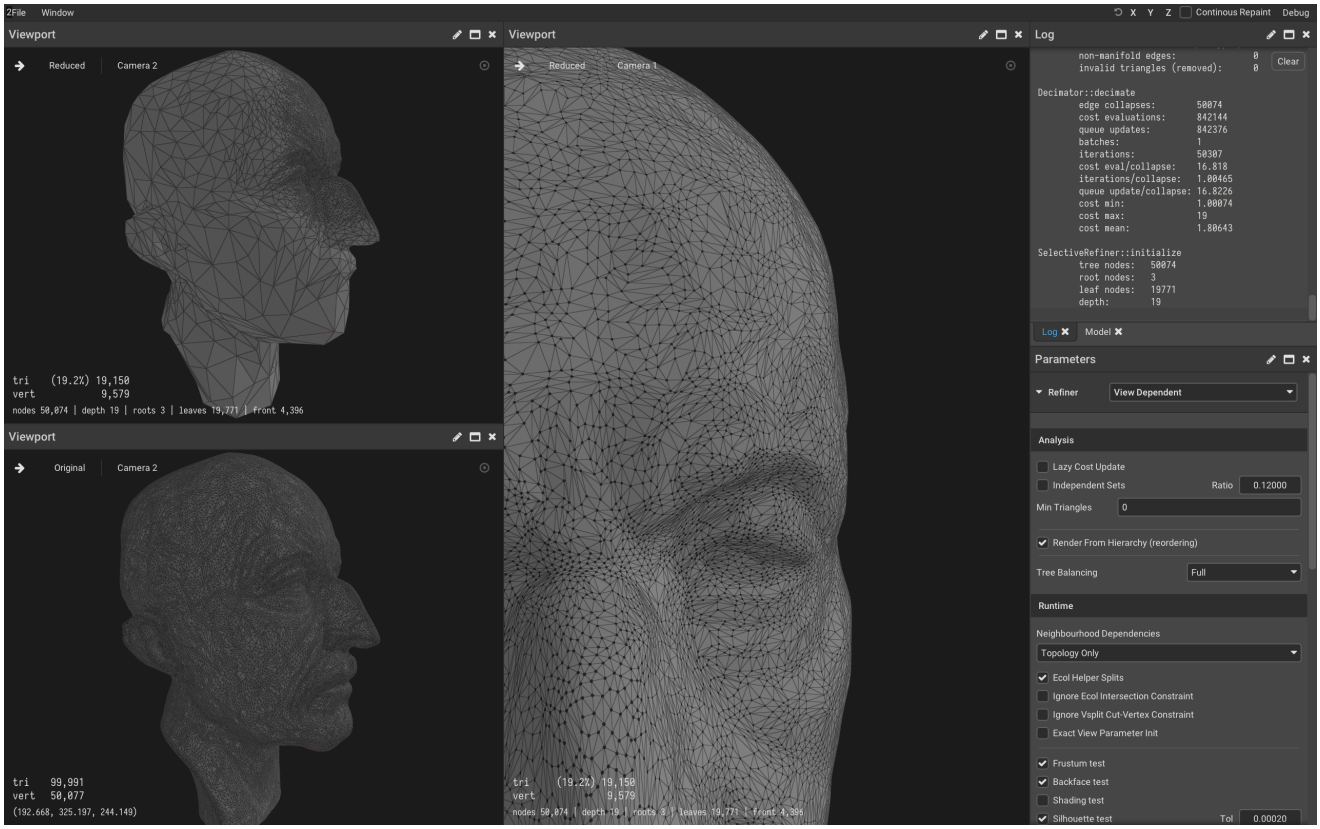



Figure 4.1: Screenshot of the viewer application.

## 4.3 Command Line Tool

In order to help compare the quality of our decimation algorithm to that of external applications, we also developed a simple command-line tool that takes a 3D model file as input and produces a decimated version of the original model as output. The tool has a number of different parameters but only supports a subset of the options provided in the library. The tool uses the Libigl library to read and write the 3D files.

# 5 Experimental results

## 5.1 Quantative Results

### 5.1.1 Decimation Accuracy

In order to evaluate the quality achieved by our decimation algorithm, we first compare our implementation to the results achieved with other programs. We then evaluate the different effects queuing and tree balancing approaches have on the quality of the decimated meshes.

#### Comparison With Other Programs

We evaluated the quality of our mesh decimation algorithm by comparing the decimated meshes to those generated by four popular programs in the domain, namely OpenFlipper, MeshLab, Blender and Qslim. All programs use a variant of the quadric error metric for the decimation. We used the open-source program MeshLab to compare the decimated models to the original model for the evaluation. The errors of the decimated meshes were computed using the one-sided Hausdorff distance metric. Using the quadric cost module, we configured our library using two different vertex placement strategies: halfedge collapse and Quadric optimal placement. We ran the tests with six common 3D test models at decimation rates 50%, 25%, 5%, 2% and 1%. In order to generate the decimated models and save them to a 3D mesh file format, we developed a simple command-line tool, as shown Section in 4.3.

Figures 5.1 and 5.2 show the mean errors at various simplification rates for the Fandisk and Bunny models, respectively. A full listing of the tested models with mean and max errors are shown in Table 5.1 and Table 5.2.

We can observe that for the tested models, the mean error rates for our quadric optimal implementation is comparable and, in many cases, surpasses the other implementations. In the halfedge collapse case, the errors are noticeably higher but still in the range of the other implementations. The maximum errors are in both cases also in the range of the other algorithms.

In the case of the Lucy model, we can observe that the quadric optimal placement strategy actually has higher maximum errors than the halfedge collapse implementation, while at the same having the lowest mean error of the tested programs. We suspect that the issue may be related to floating-point precision issues when computing the matrix inversion. It is worth noting that a low error rate in the employed metric does not necessarily translate to quality on a perceptual basis. We achieved lower error rates in general when not penalizing triangle inversion, for example. However, the data lets us conclude that our mesh decimation algorithm and implementation of the quadric error metric is correct and comparable to other well-established implementations.

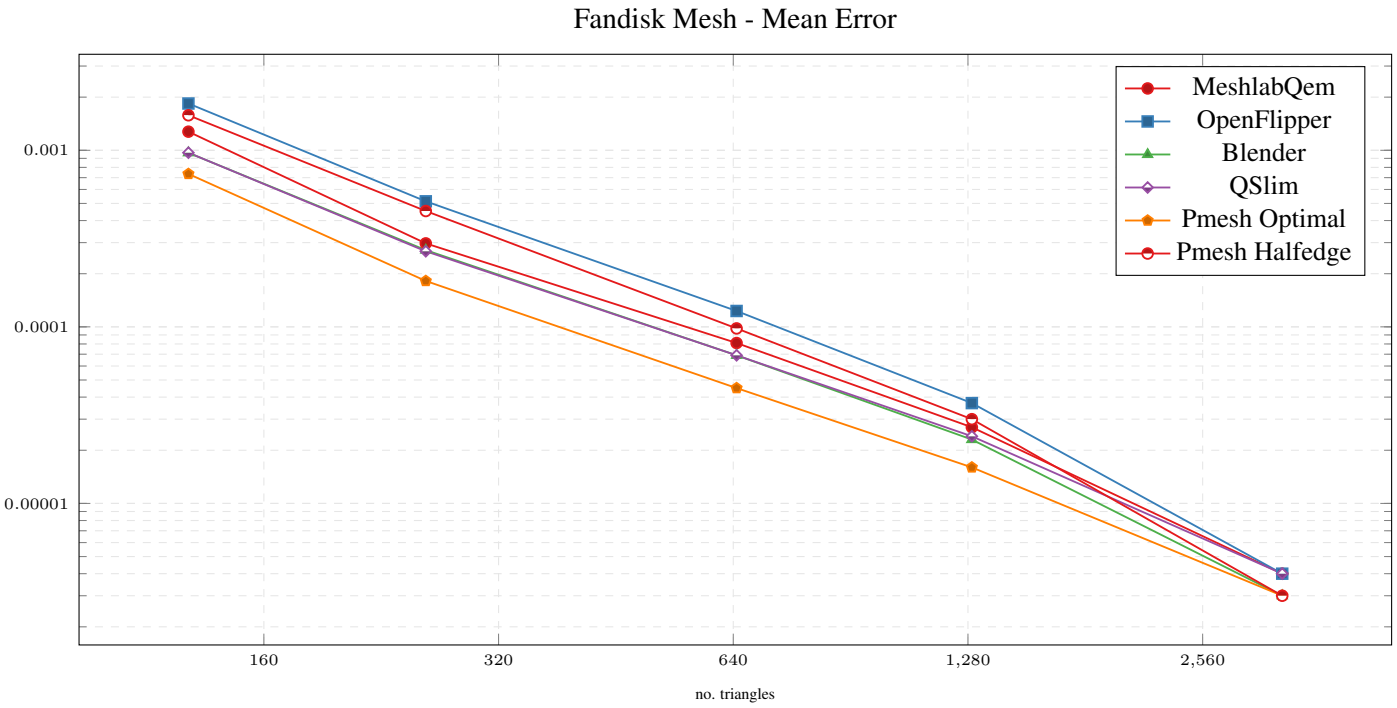



Figure 5.1: Plotting number of triangles versus mean errors of the Fandisk model using different programs for decimation.

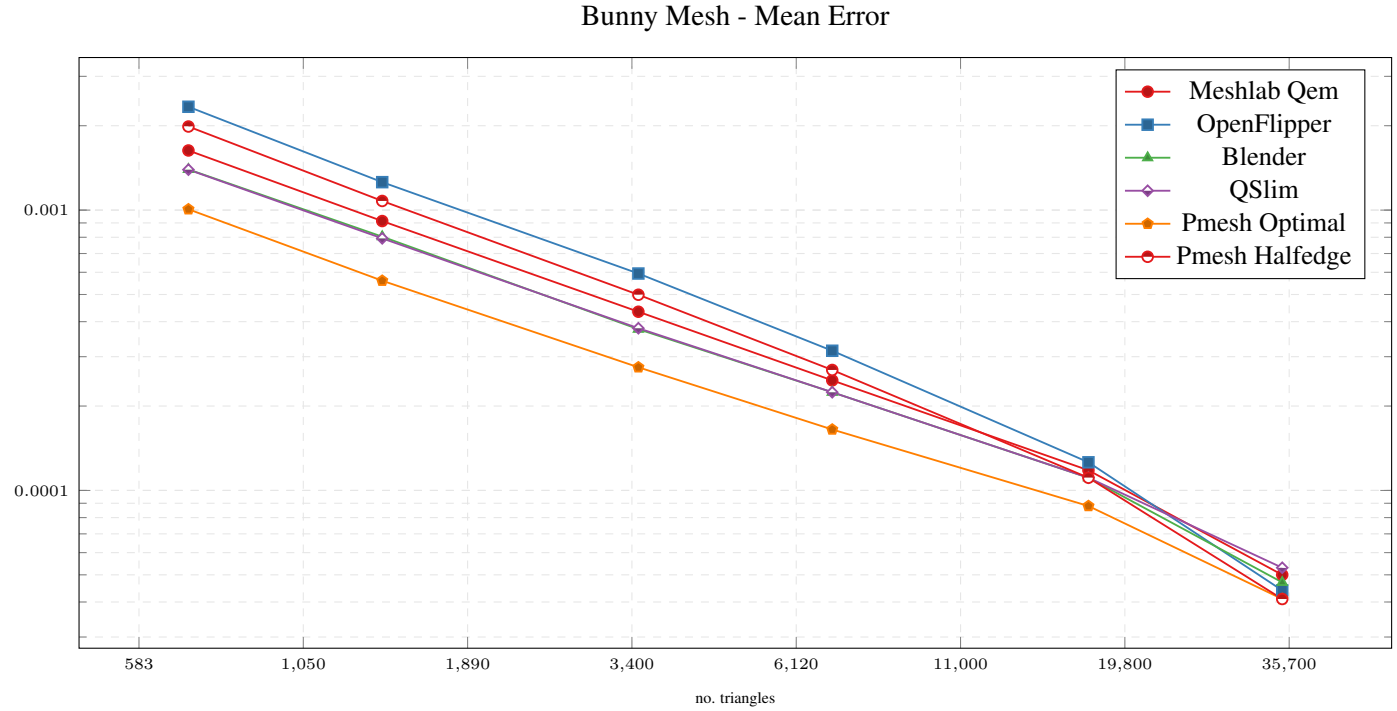



Figure 5.2: Plotting number of triangles versus mean errors of the Bunny model using different programs for decimation.

| Model | Ratio | Triangles | Blender | QSlim | Meshlab Qem | OpenFlipper | Pmesh Halfedge | Pmesh Optimal |
|---|---|---|---|---|---|---|---|---|
| Bunny | 1% | 696 | 1.40e-03 | 1.39e-03 | 1.63e-03 | 2.34e-03 | 1.99e-03 | 1.00e-03 |
| | 2% | 1392 | 8.04e-04 | 7.93e-04 | 9.14e-04 | 1.26e-03 | 1.08e-03 | 5.56e-04 |
| | 5% | 3480 | 3.76e-04 | 3.79e-04 | 4.34e-04 | 5.94e-04 | 4.99e-04 | 2.73e-04 |
| | 10% | 6962 | 2.24e-04 | 2.24e-04 | 2.47e-04 | 3.15e-04 | 2.69e-04 | 1.66e-04 |
| | 25% | 17406 | 1.11e-04 | 1.11e-04 | 1.18e-04 | 1.26e-04 | 1.11e-04 | 8.90e-05 |
| | 50% | 34814 | 4.70e-05 | 5.30e-05 | 5.00e-05 | 4.40e-05 | 4.10e-05 | 4.40e-05 |
| Dragon | 1% | 2499 | 9.39e-04 | 9.26e-04 | 1.07e-03 | 1.20e-03 | 1.02e-03 | 6.58e-04 |
| | 2% | 4998 | 5.47e-04 | 5.21e-04 | 5.93e-04 | 7.20e-04 | 6.05e-04 | 3.74e-04 |
| | 5% | 12498 | 2.66e-04 | 2.53e-04 | 2.81e-04 | 3.46e-04 | 2.89e-04 | 1.83e-04 |
| | 10% | 24998 | 1.58e-04 | 1.50e-04 | 1.67e-04 | 1.99e-04 | 1.66e-04 | 1.14e-04 |
| | 25% | 62498 | 8.60e-05 | 8.20e-05 | 9.00e-05 | 9.40e-05 | 7.90e-05 | 6.50e-05 |
| | 50% | 124998 | 4.40e-05 | 4.30e-05 | 4.80e-05 | 4.20e-05 | 3.50e-05 | 3.40e-05 |
| Fandisk | 1% | 128 | 9.68e-04 | 9.69e-04 | 1.28e-03 | 1.84e-03 | 1.58e-03 | 7.35e-04 |
| | 2% | 258 | 2.72e-04 | 2.68e-04 | 2.97e-04 | 5.14e-04 | 4.53e-04 | 1.82e-04 |
| | 5% | 646 | 6.90e-05 | 6.90e-05 | 8.10e-05 | 1.23e-04 | 9.80e-05 | 4.50e-05 |
| | 10% | 1294 | 2.30e-05 | 2.40e-05 | 2.70e-05 | 3.70e-05 | 3.00e-05 | 1.60e-05 |
| | 25% | 3236 | 3.00e-06 | 4.00e-06 | 4.00e-06 | 4.00e-06 | 3.00e-06 | 3.00e-06 |
| | 50% | 6472 | 0.00e+00 | 0.00e+00 | 0.00e+00 | 0.00e+00 | 0.00e+00 | 0.00e+00 |
| Happy | 1% | 986 | 2.88e-03 | 3.02e-03 | 3.40e-03 | 3.42e-03 | 3.06e-03 | 3.22e-03 |
| | 2% | 1972 | 1.36e-03 | 1.95e-03 | 2.76e-03 | 1.86e-03 | 1.66e-03 | 1.95e-03 |
| | 5% | 4930 | 5.68e-04 | 9.40e-04 | 1.57e-03 | 8.46e-04 | 7.48e-04 | 9.35e-04 |
| | 10% | 9860 | 3.16e-04 | 5.20e-04 | 8.79e-04 | 4.50e-04 | 4.05e-04 | 5.12e-04 |
| | 25% | 24649 | 1.52e-04 | 2.31e-04 | 3.72e-04 | 1.90e-04 | 1.72e-04 | 2.25e-04 |
| | 50% | 49299 | 7.10e-05 | 1.00e-04 | 1.64e-04 | 7.70e-05 | 7.10e-05 | 9.70e-05 |
| Lucy | 1% | 998 | 1.62e-03 | 1.64e-03 | 2.00e-03 | 2.13e-03 | 1.87e-03 | 1.28e-03 |
| | 2% | 1998 | 8.91e-04 | 8.98e-04 | 1.10e-03 | 1.23e-03 | 1.12e-03 | 7.15e-04 |
| | 5% | 4998 | 4.28e-04 | 4.10e-04 | 4.97e-04 | 5.99e-04 | 5.30e-04 | 3.16e-04 |
| | 10% | 9996 | 2.45e-04 | 2.38e-04 | 2.81e-04 | 3.41e-04 | 2.98e-04 | 1.78e-04 |
| | 25% | 24992 | 1.17e-04 | 1.14e-04 | 1.32e-04 | 1.43e-04 | 1.26e-04 | 8.60e-05 |
| | 50% | 49984 | 5.10e-05 | 5.20e-05 | 6.00e-05 | 5.30e-05 | 4.80e-05 | 4.10e-05 |
| Ogre | 1% | 1239 | 1.09e-03 | 1.74e-03 | 3.14e-03 | 1.80e-03 | 2.35e-03 | 2.03e-03 |
| | 2% | 2479 | 5.89e-04 | 9.53e-04 | 2.04e-03 | 1.13e-03 | 1.22e-03 | 7.81e-04 |
| | 5% | 6199 | 2.63e-04 | 4.22e-04 | 1.20e-03 | 6.05e-04 | 6.67e-04 | 2.97e-04 |
| | 10% | 12400 | 1.46e-04 | 2.16e-04 | 5.65e-04 | 3.39e-04 | 3.19e-04 | 1.49e-04 |
| | 25% | 31001 | 6.00e-05 | 7.40e-05 | 1.37e-04 | 1.09e-04 | 1.01e-04 | 5.40e-05 |
| | 50% | 62004 | 2.00e-05 | 2.20e-05 | 3.20e-05 | 2.80e-05 | 2.60e-05 | 1.70e-05 |

Table 5.1: Mean errors in comparison of various mesh simplification algorithms

| Model | Ratio | Triangles | Blender | QSlim | Meshlab Qem | OpenFlipper | Pmesh Halfedge | Pmesh Optimal |
|---|---|---|---|---|---|---|---|---|
| Bunny | 1% | 696 | 1.06e-02 | 8.56e-03 | 1.01e-02 | 1.64e-02 | 1.05e-02 | 7.82e-03 |
| | 2% | 1392 | 7.68e-03 | 5.80e-03 | 7.51e-03 | 9.44e-03 | 6.63e-03 | 4.58e-03 |
| | 5% | 3480 | 4.53e-03 | 3.05e-03 | 3.52e-03 | 4.02e-03 | 3.72e-03 | 2.99e-03 |
| | 10% | 6962 | 1.95e-03 | 1.88e-03 | 1.90e-03 | 2.84e-03 | 2.34e-03 | 1.50e-03 |
| | 25% | 17406 | 8.01e-04 | 8.52e-04 | 1.11e-03 | 1.62e-03 | 8.98e-04 | 8.66e-04 |
| | 50% | 34814 | 3.30e-04 | 3.50e-04 | 4.88e-04 | 6.53e-04 | 4.57e-04 | 7.19e-04 |
| Dragon | 1% | 2499 | 3.54e-02 | 1.48e-02 | 1.09e-01 | 1.65e-02 | 1.18e-02 | 1.01e-02 |
| | 2% | 4998 | 1.53e-02 | 4.55e-03 | 6.14e-02 | 1.65e-02 | 6.87e-03 | 5.44e-03 |
| | 5% | 12498 | 1.48e-02 | 3.00e-03 | 1.47e-02 | 1.65e-02 | 3.72e-03 | 3.52e-03 |
| | 10% | 24998 | 2.38e-03 | 2.97e-03 | 1.48e-02 | 1.65e-02 | 2.41e-03 | 2.91e-03 |
| | 25% | 62498 | 2.53e-03 | 2.58e-03 | 1.48e-02 | 1.65e-02 | 2.44e-03 | 2.55e-03 |
| | 50% | 124998 | 2.47e-03 | 2.57e-03 | 1.48e-02 | 1.65e-02 | 2.49e-03 | 2.56e-03 |
| Fandisk | 1% | 128 | 1.20e-02 | 1.10e-02 | 1.60e-02 | 2.17e-02 | 1.79e-02 | 9.85e-03 |
| | 2% | 258 | 2.96e-03 | 3.73e-03 | 6.38e-03 | 5.02e-03 | 4.17e-03 | 2.49e-03 |
| | 5% | 646 | 1.05e-03 | 1.11e-03 | 1.86e-03 | 1.36e-03 | 1.34e-03 | 6.15e-04 |
| | 10% | 1294 | 3.59e-04 | 3.50e-04 | 4.63e-04 | 4.93e-04 | 4.95e-04 | 2.18e-04 |
| | 25% | 3236 | 4.90e-05 | 5.00e-05 | 1.40e-04 | 7.40e-05 | 8.30e-05 | 5.60e-05 |
| | 50% | 6472 | 4.00e-06 | 7.00e-06 | 6.00e-06 | 5.00e-06 | 4.00e-06 | 4.00e-06 |
| Happy | 1% | 986 | 1.46e-02 | 1.69e-02 | 2.04e-02 | 1.80e-02 | 2.91e-02 | 1.80e-02 |
| | 2% | 1972 | 8.40e-03 | 1.27e-02 | 1.58e-02 | 1.30e-02 | 1.42e-02 | 1.70e-02 |
| | 5% | 4930 | 7.23e-03 | 9.03e-03 | 1.04e-02 | 7.32e-03 | 8.75e-03 | 8.82e-03 |
| | 10% | 9860 | 7.63e-03 | 7.88e-03 | 1.01e-02 | 8.18e-03 | 8.75e-03 | 9.25e-03 |
| | 25% | 24649 | 8.11e-03 | 7.92e-03 | 8.63e-03 | 8.18e-03 | 8.19e-03 | 7.97e-03 |
| | 50% | 49299 | 7.87e-03 | 7.85e-03 | 7.30e-03 | 7.87e-03 | 7.87e-03 | 7.85e-03 |
| Lucy | 1% | 998 | 1.13e-02 | 1.75e-02 | 2.42e-02 | 1.36e-02 | 1.86e-02 | 2.46e-02 |
| | 2% | 1998 | 6.23e-03 | 7.60e-03 | 1.10e-02 | 9.57e-03 | 2.17e-02 | 2.16e-02 |
| | 5% | 4998 | 3.31e-03 | 3.63e-03 | 6.80e-03 | 4.51e-03 | 7.32e-03 | 8.26e-03 |
| | 10% | 9996 | 1.96e-03 | 2.44e-03 | 3.60e-03 | 2.59e-03 | 3.90e-03 | 1.99e-03 |
| | 25% | 24992 | 1.09e-03 | 1.21e-03 | 1.90e-03 | 2.09e-03 | 1.46e-03 | 1.31e-03 |
| | 50% | 49984 | 4.50e-04 | 8.66e-04 | 1.58e-03 | 1.02e-03 | 1.33e-03 | 1.60e-03 |
| Ogre | 1% | 1239 | 1.05e-02 | 1.16e-02 | 1.60e-02 | 1.20e-02 | 2.02e-02 | 1.22e-02 |
| | 2% | 2479 | 5.20e-03 | 5.68e-03 | 1.39e-02 | 7.98e-03 | 7.10e-03 | 9.07e-03 |
| | 5% | 6199 | 2.36e-03 | 3.77e-03 | 7.92e-03 | 3.96e-03 | 5.46e-03 | 4.77e-03 |
| | 10% | 12400 | 1.16e-03 | 2.08e-03 | 5.26e-03 | 2.51e-03 | 2.58e-03 | 1.49e-03 |
| | 25% | 31001 | 7.78e-04 | 6.52e-04 | 1.71e-03 | 1.10e-03 | 8.42e-04 | 9.12e-04 |
| | 50% | 62004 | 3.21e-04 | 3.22e-04 | 5.83e-04 | 5.46e-04 | 4.22e-04 | 3.14e-04 |

Table 5.2: Max errors in comparison of various mesh simplification algorithms

### Quality Evaluation of Queuing and Tree Balancing Approaches

In order to evaluate the effect different queuing and tree balancing approaches have on the quality of the generated meshes, we performed the same test as discussed in Section 5.1.1 and compared it to the default implementation. The test included the following different settings:

- **Default:** The default settings corresponds to the default queuing algorithm in the Decimator module
- **Lazy:** Uses lazy cost evaluation
- **Balance Small** Decimation is performed by building a collapse hierarchy using the SelectiveRefiner component and biasing the cost.
- **Balance Full** Decimation is performed by building a collapse hierarchy using the SelectiveRefiner component and modifying the cost with the full tree balancing term.
- **Independent 12%:** Decimation is performed using the independent sets algorithm with a batch size ratio of 12%.

Figures 5.3 and 5.4 show the mean errors at various simplification rates for the Fandisk and Bunny models, respectively. A full listing of the tested models with mean and max errors are shown in Table 5.3 and Table 5.4.

We notice that the *Balance Small* algorithm shows noticeably higher errors than the default algorithm. However, at low triangle counts, the error approaches that error of the default implementation. *Balance Full* has a significantly higher error at all reduction ratios. The errors for the *Lazy* queuing approach are very close to the default implementation in all the tested and across all of the tested models. The errors for the *Independent* sets algorithm are comparable to *Balance Full* and are significant in comparison to the default implementation.

The findings with respect to the *Lazy* queuing algorithm and *Independent* sets algorithm seems to confirm the results of similar tests described in [LRC+03].

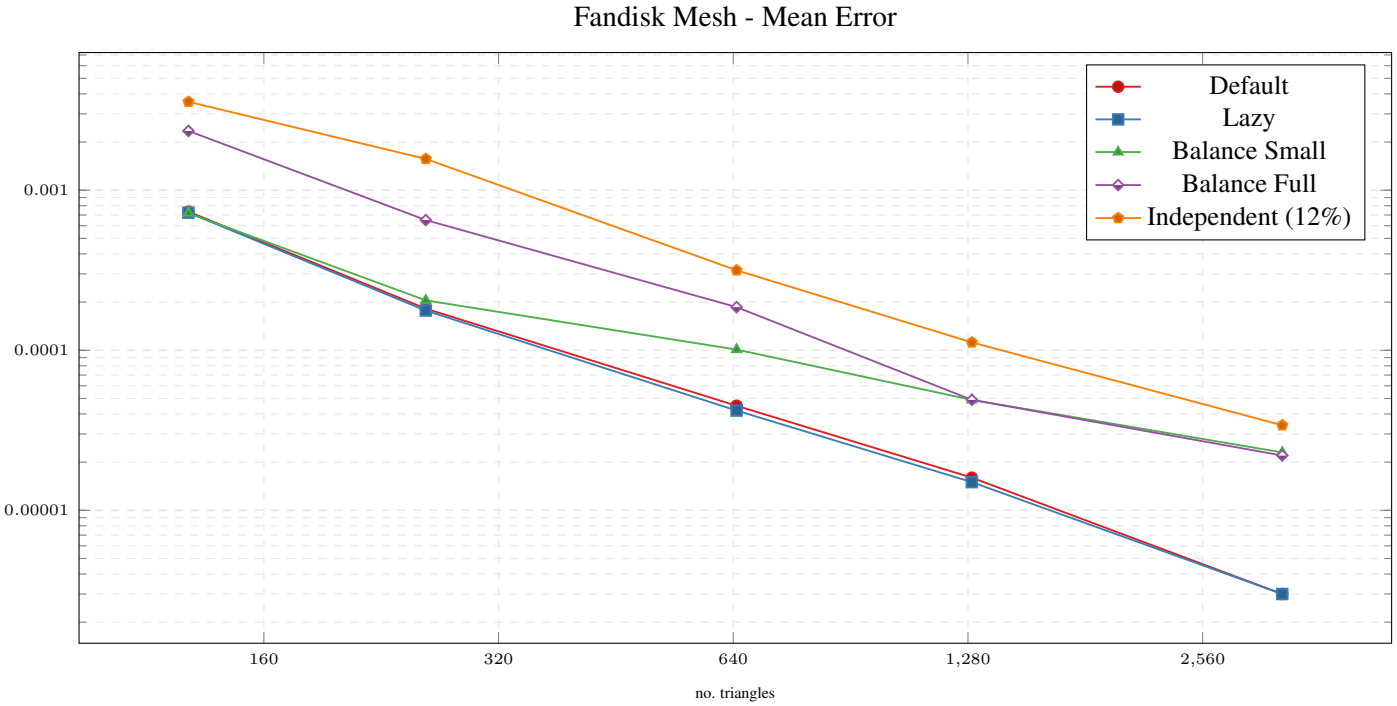



Figure 5.3: Plotting number of triangles versus mean errors of the Fandisk model using different queuing algorithms and tree balancing approaches.

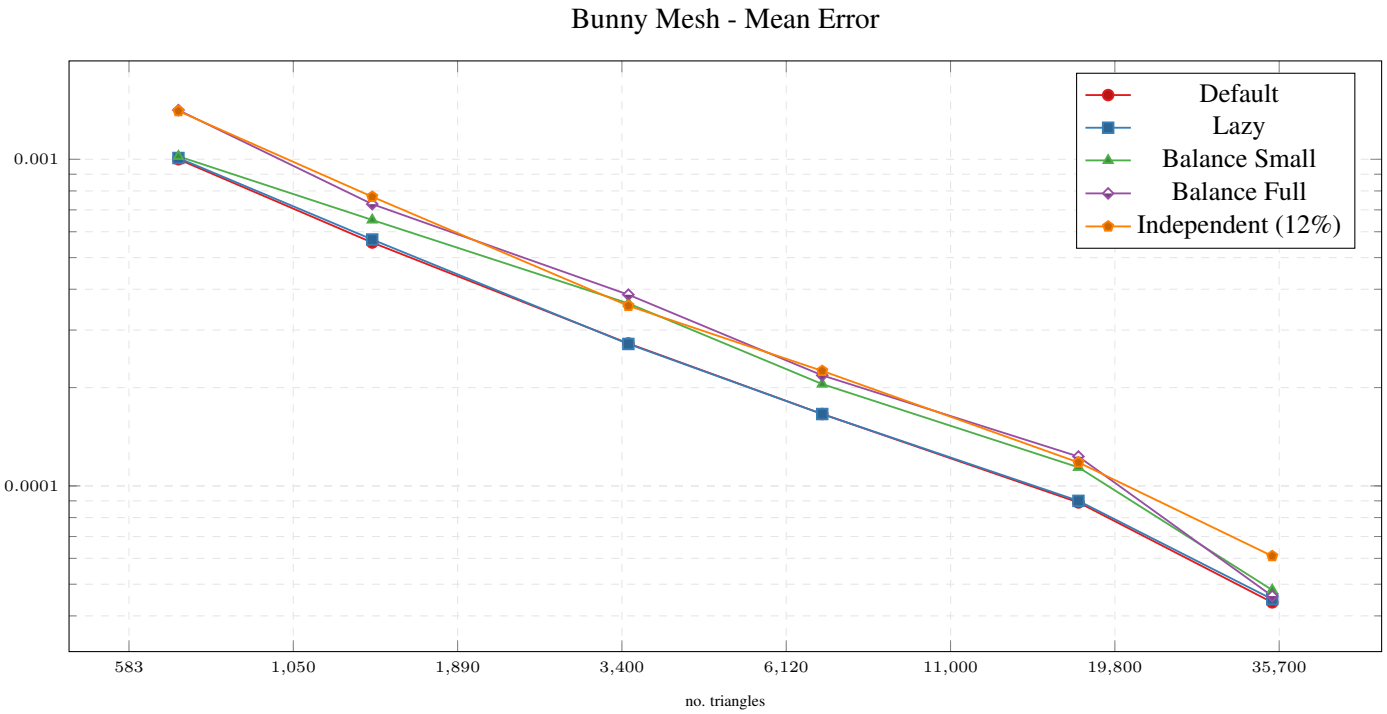



Figure 5.4: Plotting number of triangles versus mean errors of the Bunny model using different queuing algorithms and tree balancing approaches.

| Model | Ratio | Triangles | Default | Lazy | Small Balancing | Full Balancing | Independent 12% |
|---|---|---|---|---|---|---|---|
| Bunny | 1% | 696 | 1.00e-03 | 1.01e-03 | 1.02e-03 | 1.41e-03 | 1.41e-03 |
| | 2% | 1392 | 5.56e-04 | 5.68e-04 | 6.52e-04 | 7.29e-04 | 7.68e-04 |
| | 5% | 3480 | 2.73e-04 | 2.72e-04 | 3.62e-04 | 3.85e-04 | 3.56e-04 |
| | 10% | 6962 | 1.66e-04 | 1.66e-04 | 2.05e-04 | 2.18e-04 | 2.25e-04 |
| | 25% | 17406 | 8.90e-05 | 9.00e-05 | 1.14e-04 | 1.23e-04 | 1.18e-04 |
| | 50% | 34814 | 4.40e-05 | 4.50e-05 | 4.80e-05 | 4.60e-05 | 6.10e-05 |
| Dragon | 1% | 2499 | 6.57e-04 | 6.67e-04 | 6.70e-04 | 9.17e-04 | 8.48e-04 |
| | 2% | 4998 | 3.74e-04 | 3.73e-04 | 3.71e-04 | 4.94e-04 | 4.88e-04 |
| | 5% | 12498 | 1.83e-04 | 1.83e-04 | 1.84e-04 | 2.55e-04 | 2.46e-04 |
| | 10% | 24998 | 1.14e-04 | 1.14e-04 | 1.14e-04 | 1.51e-04 | 1.56e-04 |
| | 25% | 62498 | 6.50e-05 | 6.50e-05 | 6.70e-05 | 8.60e-05 | 8.40e-05 |
| | 50% | 124998 | 3.40e-05 | 3.40e-05 | 3.60e-05 | 3.50e-05 | 4.30e-05 |
| Fandisk | 1% | 128 | 7.35e-04 | 7.24e-04 | 7.19e-04 | 2.35e-03 | 3.57e-03 |
| | 2% | 258 | 1.82e-04 | 1.77e-04 | 2.05e-04 | 6.51e-04 | 1.57e-03 |
| | 5% | 646 | 4.50e-05 | 4.20e-05 | 1.01e-04 | 1.86e-04 | 3.16e-04 |
| | 10% | 1294 | 1.60e-05 | 1.50e-05 | 4.90e-05 | 4.90e-05 | 1.12e-04 |
| | 25% | 3236 | 3.00e-06 | 3.00e-06 | 2.30e-05 | 2.20e-05 | 3.40e-05 |
| | 50% | 6472 | 0.00e+00 | 0.00e+00 | 1.00e-06 | 0.00e+00 | 1.20e-05 |
| Happy | 1% | 986 | 3.22e-03 | 3.37e-03 | 3.97e-03 | 4.15e-03 | 3.88e-03 |
| | 2% | 1972 | 1.95e-03 | 2.02e-03 | 2.28e-03 | 2.64e-03 | 2.38e-03 |
| | 5% | 4930 | 9.35e-04 | 9.37e-04 | 1.16e-03 | 1.18e-03 | 1.11e-03 |
| | 10% | 9860 | 5.12e-04 | 5.20e-04 | 6.25e-04 | 6.46e-04 | 6.25e-04 |
| | 25% | 24649 | 2.25e-04 | 2.26e-04 | 2.68e-04 | 2.90e-04 | 2.76e-04 |
| | 50% | 49299 | 9.70e-05 | 9.90e-05 | 1.00e-04 | 9.90e-05 | 1.23e-04 |
| Lucy | 1% | 998 | 1.28e-03 | 1.28e-03 | 1.28e-03 | 1.72e-03 | 1.67e-03 |
| | 2% | 1998 | 7.15e-04 | 7.28e-04 | 7.15e-04 | 9.55e-04 | 9.14e-04 |
| | 5% | 4998 | 3.16e-04 | 3.18e-04 | 3.16e-04 | 4.65e-04 | 4.13e-04 |
| | 10% | 9996 | 1.78e-04 | 1.78e-04 | 1.78e-04 | 2.34e-04 | 2.46e-04 |
| | 25% | 24992 | 8.60e-05 | 8.70e-05 | 8.60e-05 | 1.17e-04 | 1.12e-04 |
| | 50% | 49984 | 4.10e-05 | 4.20e-05 | 4.10e-05 | 4.30e-05 | 5.50e-05 |
| Ogre | 1% | 1239 | 1.56e-03 | 1.65e-03 | 1.58e-03 | 1.35e-03 | 1.33e-03 |
| | 2% | 2479 | 7.18e-04 | 7.28e-04 | 7.37e-04 | 6.36e-04 | 6.49e-04 |
| | 5% | 6199 | 2.94e-04 | 3.02e-04 | 2.51e-04 | 3.17e-04 | 2.97e-04 |
| | 10% | 12400 | 1.49e-04 | 1.46e-04 | 1.20e-04 | 1.50e-04 | 1.60e-04 |
| | 25% | 31001 | 5.40e-05 | 5.40e-05 | 6.40e-05 | 7.30e-05 | 6.70e-05 |
| | 50% | 62004 | 1.70e-05 | 1.70e-05 | 1.90e-05 | 1.80e-05 | 3.30e-05 |

Table 5.3: Mean errors in comparison of different queuing and tree balancing approaches.

| Model | Ratio | Triangles | Ratio | Default | Lazy | Small Balancing | Full Balancing | Independent 12% |
|---|---|---|---|---|---|---|---|---|
| Bunny | 1% | 696 | 1% | 7.82E-03 | 6.03E-03 | 8.25E-03 | 1.37E-02 | 1.22E-02 |
| | 2% | 1392 | 2% | 4.58E-03 | 4.94E-03 | 6.05E-03 | 8.42E-03 | 8.24E-03 |
| | 5% | 3480 | 5% | 2.99E-03 | 2.66E-03 | 6.95E-03 | 6.32E-03 | 5.94E-03 |
| | 10% | 6962 | 10% | 1.50E-03 | 1.83E-03 | 3.46E-03 | 4.06E-03 | 5.20E-03 |
| | 25% | 17406 | 25% | 8.66E-04 | 8.54E-04 | 2.93E-03 | 3.74E-03 | 3.87E-03 |
| | 50% | 34814 | 50% | 7.19E-04 | 7.14E-04 | 7.39E-04 | 7.39E-04 | 1.85E-03 |
| Dragon | 1% | 2499 | 1% | 1.01E-02 | 8.63E-03 | 1.18E-02 | 1.94E-02 | 1.85E-02 |
| | 2% | 4998 | 2% | 5.44E-03 | 6.00E-03 | 7.04E-03 | 1.49E-02 | 1.85E-02 |
| | 5% | 12498 | 5% | 3.52E-03 | 2.96E-03 | 2.92E-03 | 1.49E-02 | 1.85E-02 |
| | 10% | 24998 | 10% | 2.91E-03 | 2.93E-03 | 3.01E-03 | 1.49E-02 | 1.85E-02 |
| | 25% | 62498 | 25% | 2.55E-03 | 2.55E-03 | 2.56E-03 | 1.49E-02 | 1.45E-02 |
| | 50% | 124998 | 50% | 2.56E-03 | 2.52E-03 | 2.56E-03 | 2.56E-03 | 1.35E-02 |
| Fandisk | 1% | 128 | 1% | 9.85E-03 | 8.98E-03 | 1.12E-02 | 5.62E-02 | 5.18E-02 |
| | 2% | 258 | 2% | 2.49E-03 | 2.20E-03 | 3.95E-03 | 2.95E-02 | 5.05E-02 |
| | 5% | 646 | 5% | 6.15E-04 | 5.39E-04 | 2.56E-03 | 1.71E-02 | 2.80E-02 |
| | 10% | 1294 | 10% | 2.18E-04 | 2.18E-04 | 4.85E-03 | 6.03E-03 | 1.37E-02 |
| | 25% | 3236 | 25% | 5.60E-05 | 5.60E-05 | 5.49E-03 | 6.03E-03 | 1.17E-02 |
| | 50% | 6472 | 50% | 4.00E-06 | 3.00E-06 | 2.10E-05 | 9.00E-06 | 5.74E-03 |
| Happy | 1% | 986 | 1% | 1.80E-02 | 2.22E-02 | 2.12E-02 | 2.50E-02 | 2.19E-02 |
| | 2% | 1972 | 2% | 1.70E-02 | 1.57E-02 | 1.59E-02 | 1.69E-02 | 1.68E-02 |
| | 5% | 4930 | 5% | 8.82E-03 | 8.40E-03 | 1.15E-02 | 1.02E-02 | 8.52E-03 |
| | 10% | 9860 | 10% | 9.25E-03 | 9.13E-03 | 7.68E-03 | 7.74E-03 | 7.46E-03 |
| | 25% | 24649 | 25% | 7.97E-03 | 7.92E-03 | 7.97E-03 | 7.97E-03 | 7.96E-03 |
| | 50% | 49299 | 50% | 7.85E-03 | 7.85E-03 | 7.85E-03 | 7.85E-03 | 7.78E-03 |
| Lucy | 1% | 998 | 1% | 2.46E-02 | 2.37E-02 | 2.46E-02 | 2.71E-02 | 2.09E-02 |
| | 2% | 1998 | 2% | 2.16E-02 | 2.15E-02 | 2.16E-02 | 1.84E-02 | 1.62E-02 |
| | 5% | 4998 | 5% | 8.26E-03 | 9.80E-03 | 8.26E-03 | 7.92E-03 | 8.94E-03 |
| | 10% | 9996 | 10% | 1.99E-03 | 1.87E-03 | 1.99E-03 | 3.00E-03 | 4.31E-03 |
| | 25% | 24992 | 25% | 1.31E-03 | 1.10E-03 | 1.31E-03 | 1.55E-03 | 1.59E-03 |
| | 50% | 49984 | 50% | 1.60E-03 | 8.57E-04 | 1.60E-03 | 1.31E-03 | 1.53E-03 |
| Ogre | 1% | 1239 | 1% | 1.22E-02 | 1.78E-02 | 1.72E-02 | 1.56E-02 | 1.86E-02 |
| | 2% | 2479 | 2% | 9.07E-03 | 8.64E-03 | 8.18E-03 | 1.08E-02 | 1.31E-02 |
| | 5% | 6199 | 5% | 4.77E-03 | 4.96E-03 | 3.07E-03 | 9.87E-03 | 9.19E-03 |
| | 10% | 12400 | 10% | 1.49E-03 | 1.32E-03 | 2.50E-03 | 3.53E-03 | 4.85E-03 |
| | 25% | 31001 | 25% | 9.12E-04 | 6.94E-04 | 1.20E-03 | 2.46E-03 | 2.15E-03 |
| | 50% | 62004 | 50% | 3.14E-04 | 3.14E-04 | 2.78E-04 | 2.78E-04 | 1.50E-03 |

Table 5.4: Max errors in comparison in comparison of different queuing and tree balancing approaches.

### 5.1.2 Performance Measurements

#### Comparison With External Libraries

We evaluated the performance of our mesh decimation algorithm and compared it to three popular mesh processing libraries: OpenMesh, CGAL and Libigl. We conducted the tests by integrating the libraries into our viewer application and ran a dedicated benchmark procedure on each implementation for a number of different meshes. In order to evaluate the performance on larger meshes, we included three models from the Stanford 3D Scanning Repository whose size range from 1 million to around 7 million triangles. The measurement for the smaller models was averaged over 10 runs. Each mesh was decimated to 10% of its original triangles.

We additionally measured the time it took to initialize each of the libraries mesh data structures. In order to avoid measuring file I/O, we initialized each library from memory. The results of these measurements are, however, not directly comparable since each of the libraries handles initialization differently. Furthermore, these measurements for $time_{init}$ also include the running time of our own code since we were unable to pass contiguous buffers to the libraries but rather had to add each vertex and face individually.

Apart from the external libraries, we measured the decimation and initialization times for different configurations of our own library: Pmesh Lazy corresponds to the lazy queuing algorithm. For *Pmesh View* the ViewRefiner module was initialized, which includes building edge collapse hierarchy during the decimation process. For *Pmesh Independent* the independent sets algorithm was used.

All tests were performed on a Lenovo laptop with an AMD Ryzen 7 5800H 3.20 GHz CPU and 16GB of RAM running Windows 10. The application and all external libraries were compiled with Visual Studio 2019 with maximum optimization (/O2) and debugging checks disabled. For each of the mesh libraries in the comparison, we used the latest versions available, .i.e., OpenMesh 8.1, CGAL 5.3, and Libigl 2.2.0.

The results of the measurements is shown in Table 5.5.

We find that for each of the meshes in the test, our library performs faster than any of the external libraries tested. Even in the *Pmesh View* case, which includes initializing the edge collapse hierarchy and computing the view-test parameters for each node, the time taken is lower in all but one case, where the test result matches that of the fastest external library. Out of the external libraries, OpenMesh has the best performance. The measured times for CGAL and Libigl are significantly worse in each case. For three of the models, Libigl failed to produce a result. For the Happy model, OpenMesh got stuck, and we had to manually kill the process.

Comparing the timings for the different configurations of Pmesh, we find that the lazy queuing algorithm does indeed provide a significant speed advantage compared to the default queueing approach. This is especially noteworthy since the comparison in 5.1.1 showed that there is virtually no difference between the two in terms of decimation quality. The independent sets algorithm is slower in most cases than the default implementation. We expect, however, that the timings could be improved by using a different sorting algorithm as described in [PD04].

| Model | $N_{InTriangles}$ | Algorithm | $N_{Triangles}$ | $time_{init}$ (s) | $time_{dec}$ (s) | $time_{total}$ (s) |
|---|---|---|---|---|---|---|
| Happy | 1,087,716 | Libigl | N/A | N/A | N/A | N/A |
| | | OpenMesh | N/A | N/A | N/A | N/A |
| | | Pmesh Lazy | 108,540 | 0.13 | 4.65 | 4.77 |
| | | Pmesh | 108,540 | 0.13 | 6.15 | 6.29 |
| | | Pmesh View | 108,540 | 0.13 | 6.51 | 6.65 |
| | | Pmesh Independent | 108,540 | 0.13 | 6.65 | 6.78 |
| | | CGAL | 108,520 | 0.47 | 21.41 | 21.88 |
| Dragon | 7,219,045 | Libigl | N/A | N/A | N/A | N/A |
| | | Pmesh Lazy | 721,903 | 0.80 | 40.72 | 41.52 |
| | | Pmesh | 721,903 | 0.79 | 45.65 | 46.45 |
| | | Pmesh Independent | 721,903 | 0.83 | 47.52 | 48.35 |
| | | Pmesh View | 721,903 | 0.79 | 50.17 | 50.96 |
| | | OpenMesh | 721,877 | 1.30 | 54.47 | 55.76 |
| | | CGAL | 721,861 | 3.08 | 133.33 | 136.41 |
| Nefertiti | 2,018,232 | Pmesh Lazy | 201,822 | 0.26 | 9.56 | 9.81 |
| | | Pmesh | 201,822 | 0.27 | 11.88 | 12.15 |
| | | Pmesh Independent | 201,822 | 0.26 | 12.05 | 12.31 |
| | | Pmesh View | 201,822 | 0.26 | 13.03 | 13.29 |
| | | OpenMesh | 201,818 | 0.44 | 15.28 | 15.72 |
| | | CGAL | 201,822 | 0.92 | 34.40 | 35.32 |
| | | Libigl | 201,822 | 0.01 | 46.71 | 46.72 |
| Max Planck | 99,991 | Libigl | N/A | N/A | N/A | N/A |
| | | Pmesh Lazy | 9,998 | 0.01 | 0.22 | 0.23 |
| | | Pmesh | 9,999 | 0.01 | 0.33 | 0.34 |
| | | Pmesh Independent | 9,998 | 0.01 | 0.35 | 0.36 |
| | | Pmesh View | 9,999 | 0.01 | 0.38 | 0.39 |
| | | OpenMesh | 9,960 | 0.02 | 0.41 | 0.43 |
| | | CGAL | 9,972 | 0.04 | 1.76 | 1.80 |
| Bunny | 69,630 | Pmesh Lazy | 6,962 | 0.01 | 0.15 | 0.16 |
| | | Pmesh Independent | 6,962 | 0.01 | 0.22 | 0.23 |
| | | Pmesh | 6,962 | 0.01 | 0.23 | 0.23 |
| | | Pmesh View | 6,962 | 0.01 | 0.27 | 0.28 |
| | | OpenMesh | 6,958 | 0.01 | 0.28 | 0.29 |
| | | CGAL | 6,962 | 0.03 | 1.13 | 1.17 |
| | | Libigl | 6,962 | 0.00 | 1.23 | 1.24 |
| Horse | 96,966 | Pmesh Lazy | 9,696 | 0.01 | 0.26 | 0.26 |
| | | Pmesh | 9,696 | 0.01 | 0.39 | 0.40 |
| | | Pmesh Independent | 9,696 | 0.01 | 0.40 | 0.41 |
| | | OpenMesh | 9,692 | 0.02 | 0.43 | 0.45 |
| | | Pmesh View | 9,696 | 0.01 | 0.44 | 0.45 |
| | | Libigl | 9,696 | 0.00 | 1.66 | 1.66 |
| | | CGAL | 9,736 | 0.04 | 1.90 | 1.94 |

Table 5.5: Comparison of mesh decimation times for different library implementations. The meshes were decimated to a ratio of 10%.

| Model | Input Triangles | Algorithm | Tree Depth | Processing [ms] | Triangles | Delta Triangles | Operations |
|---|---|---|---|---|---|---|---|
| Happy | 1,087,716 | BalancingFull | 20 | 3.5 | 39,200 | 228 | 267 |
| | | BalancingSmall | 18 | 3.5 | 38,178 | 222 | 253 |
| | | Default | 45 | 3.4 | 38,162 | 219 | 254 |
| | | Independent12 | 29 | 3.6 | 39,091 | 228 | 260 |
| Nefertiti | 2,018,232 | BalancingFull | 24 | 1.5 | 22,909 | 164 | 239 |
| | | BalancingSmall | 38 | 1.6 | 23,545 | 168 | 255 |
| | | Default | 35 | 1.6 | 23,616 | 168 | 254 |
| | | Independent12 | 32 | 1.6 | 23,534 | 169 | 245 |
| Dragon | 7,219,045 | BalancingFull | 26 | 2.6 | 30,031 | 203 | 197 |
| | | BalancingSmall | 40 | 2.7 | 30,287 | 202 | 203 |
| | | Default | 41 | 2.8 | 30,351 | 204 | 208 |
| | | Independent12 | 37 | 2.7 | 30,724 | 208 | 207 |
| Igea | 268,686 | BalancingFull | 19 | 1.1 | 25,658 | 189 | 286 |
| | | BalancingSmall | 17 | 1.2 | 23,504 | 172 | 258 |
| | | Default | 27 | 1.1 | 23,136 | 171 | 267 |
| | | Independent12 | 26 | 1.1 | 24,561 | 179 | 270 |
| Lucy | 99,970 | BalancingFull | 18 | 0.9 | 28,270 | 179 | 158 |
| | | BalancingSmall | 34 | 0.9 | 28,654 | 181 | 161 |
| | | Default | 34 | 0.9 | 28,654 | 181 | 161 |
| | | Independent12 | 26 | 0.9 | 28,794 | 182 | 156 |
| Max Planck | 99,991 | BalancingFull | 19 | 0.5 | 18,082 | 118 | 152 |
| | | BalancingSmall | 27 | 0.5 | 18,102 | 118 | 157 |
| | | Default | 27 | 0.5 | 18,107 | 118 | 156 |
| | | Independent12 | 26 | 0.5 | 18,478 | 120 | 154 |
| Bunny | 69,630 | BalancingFull | 17 | 0.5 | 17,771 | 108 | 146 |
| | | BalancingSmall | 20 | 0.5 | 17,640 | 108 | 146 |
| | | Default | 25 | 0.5 | 17,783 | 108 | 152 |
| | | Independent12 | 23 | 0.5 | 18,311 | 111 | 148 |

Table 5.6: Comparison of frame times for view dependent refinement.

### Run-time Performance of View-Dependent Refinement

We measured the runtime performance of the view-dependent refinement algorithm using the same setup as described in Section 5.1.2. We tested the default algorithm without any tree balancing, as well as three different configurations for tree balancing, *BalancingFull*, *BalancingSmall*, and *Independent12*. In order to the conduct test, we recorded the position and viewing direction of a virtual camera and averaged the collected data over a total of 6166 frames for each of the different algorithms.

The results of the measurements are shown in Table 5.6. The column Processing represents the time processing per frame in milliseconds. The column Triangles is the average number of triangles per frame. Delta Triangles shows the average number of triangles that were added or removed in each frame. The column operations is the average of the sum of vertex splits and edge collapses performed in each frame.

We can see that the processing times remain roughly the same for each of the algorithms tested. The processing time is not proportional to the number of operations per frame or the number of active triangles but also depends on the overall model size. As can be seen, when comparing the Happy model with roughly 1 million triangles to the dragon model, roughly 7 million triangles other factors such as the mesh topology can be significant.

# 6 Conclusion and discussion

The main objective of this thesis was to design and implement an efficient and parametrizable algorithm for edge-collapses and vertex splits.

In this thesis, we presented how we constructed our data structures and explained the workings of the algorithms we used. We conducted a range of tests to compare the performance and general quality of our solution to other existing programs and libraries.

We developed a self-contained library that covers a range of application areas. We built components that can be used for continuous level of detail representations of meshes or for more general mesh simplification tasks. We also investigated and implemented algorithms for the selective refinement of meshes, which includes a component for view-dependent refinement. We believe that our implementation is well parameterizable and can be extended by additional components through the use of well-defined interfaces.

The comparing analysis with other programs in terms of mesh decimation quality showed that the presented solution matches or exceeds that of the tested programs. In terms of performance, we find that our implementation is significantly faster than some of the other libraries we tested.

# Bibliography


[Bau72] Bruce G Baumgart. Winged edge polyhedron representation. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

[BKP+10] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon mesh processing*. CRC press, 2010.

[BPK+08] M. Botsch, M. Pauly, L. Kobbelt, P. Alliez, and B. Lévy. Geometric modeling based on polygonal meshes. In *Eurographics*, 2008.

[CKS98] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges—a scalable representation for triangle meshes. *Journal of Graphics tools*, 3(4):1–11, 1998.

[Cla76] James H Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[Coh99] Jonathan David Cohen. Appearance-preserving simplification of polygonal models. *Chapel Hill*, pages 1–130, 1999.

[DFMP97] L De Fioriani, Paola Magillo, and Enrico Puppo. Building and traversing a surface at variable resolution. In *Proceedings. Visualization'97 (Cat. No. 97CB36155)*, pages 103–110. IEEE, 1997.

[ESV99] Jihad El-Sana and Amitabh Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, 1999.

[Gar99] Michael Garland. *Quadric-based polygonal surface simplification*. Carnegie Mellon University, 1999.

[GH97] Michael Garland and Paul S Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, 1997.

[Gra02] Markus Grabner. Tree balancing for mesh simplification. In *Eurographics (Short Presentations)*, 2002.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM transactions on graphics (TOG)*, 4(2):74–123, 1985.

[Gué96] André Guéziec. *Surface simplification inside a tolerance volume*. IBM TJ Watson Research Center, 1996.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26, 1993.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, 1997.

[Hor15] Kai Hormann. *Geometry processing*, pages 593–606. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[HSH09] Liang Hu, Pedro V Sander, and Hugues Hoppe. Parallel view-dependent level-of-detail control. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):718–728, 2009.

[KL01] Junho Kim and Seungyong Lee. Truly selective refinement of progressive meshes. In *Graphics Interface*, volume 1, pages 101–110. Citeseer, 2001.

[LRC+03] David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[LT99] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, 1999.

[Paj01] Renato Pajarola. Fastmesh: Efficient view-dependent meshing. In *Proceedings Ninth Pacific Conference on Computer Graphics and Applications. Pacific Graphics 2001*, pages 22–30. IEEE, 2001.

[PD04] Renato Pajarola and Christopher DeCoro. Efficient implementation of real-time view-dependent multiresolution meshing. *IEEE Transactions on Visualization and Computer Graphics*, 10(3):353–368, 2004.

[PR00] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, 2000.

[RB93] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in computer graphics*, pages 455–465. Springer, 1993.

[SB11] Daniel Sieger and Mario Botsch. Design, implementation, and evaluation of the surfacemesh data structure. In *Proceedings of the 20th international meshing roundtable*, pages 533–550. Springer, 2011.

[XESV97] Julie C. Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive real-time level-of-detail based rendering for polygonal models. *IEEE Transactions on Visualization and Computer graphics*, 3(2):171–183, 1997.

[XV96] Julie C Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of Seventh Annual IEEE Visualization'96*, pages 327–334. IEEE, 1996.