Vision, Modeling, and Visualization (2022)
J. Bender, M. Botsch, and D. Keim (Eds.)

Relaxed Parallel Priority Queue with Filter Levels for Parallel Mesh Decimation

Marvin Stümmel Felix Brüll Thorsten Grosch

TU Clausthal, Germany

Abstract

We propose a novel implementation of a parallel priority queue in the context of multithreaded mesh decimation. Previous parallel priority queues either have a major bottleneck when extracting nodes, cannot guarantee reasonable node quality for the extracted nodes, or cannot be used for mesh decimation. Our data structure allows the extraction of multiple high-priority elements at the same time. For this, we relax the requirement of returning the highest priority element to returning an element that belongs to the top k elements. We demonstrate its use in the context of parallel mesh decimation and show that our decimated mesh is almost indistinguishable from an optimally decimated mesh while being 2 to 2.6 times faster than a naive parallel priority queue implementation.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Mesh models; Parallel algorithms;***

1. Introduction

Most 3D models are represented by triangle meshes and are used in many fields such as architecture, movie production or games. The models are usually created by artists who like to work with a high number of triangles per mesh. Alternatively, the meshes can be obtained from 3D scans, which also have a high number of triangles. For real-time applications, such high-resolution 3D models can be difficult to render, and lower resolution meshes are often required to maintain real-time frame rates.

The state-of-the-art algorithms to reduce the triangle count are based on incremental mesh decimation and vertex clustering. Incremental mesh decimation generally results in higher quality meshes, but is sequential by definition, while vertex clustering can be easily parallelized and is much faster. Since mesh decimation is usually an offline process, meaning that it is performed once and not every frame, higher quality is desirable, and longer decimation times are tolerable. Therefore, we decided to focus on incremental mesh decimation.

1.1. Incremental Mesh Decimation

In incremental mesh decimation [BKP*10], two connected vertices (one edge) of the triangle mesh are selected and merged into a single vertex. This reduces the number of vertices by one and is repeated until the target number of vertices is reached. The edge selection is driven by an error metric, which is usually the quadric error metric by Garland and Heckbert [GH97]. The merging of the vertices is called edge collapse [KCS98] and we use the simple half-edge collapse, where one vertex collapses onto the other.

It is recommended to precompute all error quadrics and store the vertex IDs ordered by their lowest possible error in a (heap-based) priority queue. The best possible half-edge collapse can then be extracted in logarithmic time for each iteration. After performing an edge collapse, the quadric errors of all neighboring vertices need to be updated as well. In order to realize this with a priority queue, one needs the possibility to locate specific vertices in the priority queue, and also the ability to change their priorities afterwards. Vertices can be located in constant time by maintaining a lookup table that maps from vertex IDs to the position in the priority queue heap. The change of priority can be implemented in logarithmic time by performing a bubble down or bubble up of the heap element, depending on the priority change. The runtime of this algorithm is $\mathcal{O}(m \log_2 n)$ where n is the total number of vertices of the original mesh and $m < n$ is the number of vertices that need to be removed.

To summarize: A priority queue with mutable priorities, and a lookup table from vertex IDs to priority queue elements is required to implement an efficient incremental mesh decimation algorithm.

2. Related Work

In this section, we will first go over the most relevant mesh decimation algorithms and the available parallel implementations. In the second subsection, we will further list available parallel priority queues and discuss their suitability for parallelizing the incremental mesh decimation algorithm from Sec. 1.1.

DOI: 10.2312/vmv.20221202

2.1. Mesh Decimation

Incremental Mesh Decimation algorithms decimate the model through edge collapses [Hop96]. Garland and Heckbert [GH97] propose an additional aggregation step, which joins previously unconnected vertices that allows for more aggressive optimizations.

Parallel implementations of incremental mesh decimation have been attempted: Dehne et al. [DLR02] partitions the mesh and decimates each partition using the sequential algorithm. A similar approach has recently been implemented on the GPU using partitioning and a skip list based priority queue design [MH21]. Franc and Skala [FS01] create a *super independent set* of vertices, where each vertex can perform its edge collapse independently of all other vertices in the set. In their implementation, the set is created sequentially, and the collapses are executed in parallel. After a thread synchronization, this procedure is repeated until the target decimation is achieved. This was implemented on the GPU by Papageorgiou and Platis [PP15]. Note that all the parallel approaches do not work toward the same result as the sequential decimation algorithm because the order and choice of edge collapses differ significantly.

Multiple-Choice Mesh Decimation [WK02] is a variant of incremental mesh decimation, where the best edge collapse out of eight randomly chosen edges is executed in each iteration. The idea is that at least one out of eight randomly chosen edge collapses will be performed anyway at some point. This variant was implemented on the GPU by Koh et al. [KZZC18].

Vertex Decimation [SZL92,DFP95] describes algorithms that remove a vertex with all adjacent faces, and then retriangulate the resulting hole accordingly.

Vertex Clustering [RB93] uses a different approach by aligning the mesh in a uniform grid. In the next step, a single vertex representative is determined for all vertices, that fall into the same grid cell. Finally, proper connectivity is restored. This has been implemented on the GPU by DeCoro et al. [DT07].

Illumination-driven mesh reduction [RGG15,BJG20] sets the focus on producing a visually identical image from a fixed viewpoint with a reduced mesh representation. In addition to surface curvature, the change in visible radiance in the image, determined by a global illumination simulation, is used as the priority for edge collapses. The intentions are to reduce the overall rendering time and to transform out-of-core scenes to in-core scenes through mesh decimation.

2.2. Priority Queue

The *Heap-Based Concurrent Priority Queue with Mutable Priorities* [TMR16] implements all operations that are necessary for parallel mesh decimation. However, we will show in Sec. 4.3 that this priority queue has a major bottleneck for parallel mesh decimation, which results in almost no performance improvements when compared to the sequential decimation. The problem is that only the root element of the priority queue can be removed, which is problematic when many threads try to pop from the priority queue. To work around this problem, a couple of *relaxed priority queues* have been introduced, that allow the removal of an arbitrary good element, that is not necessarily the best:

The *Lock-free k-LSM Relaxed Priority Queue* [WGTT15] guarantees that a pop operation returns any of the $k \cdot t$ best elements, where k is a configurable parameter and t is the number of threads. Unfortunately, mutable priorities are not supported and are probably not possible without violating the lock-free policy.

MultiQueues [RSD15] describes another implementation of a relaxed priority queue. Internally, $t \cdot 2$ sequential priority queues are allocated, where t is the number of threads. For the insert operation, a random priority queue is locked, and the element is inserted. For the pop operation, two random priority queues are locked, and the best element from the two queues is popped and returned. Since each thread can hold at most two locks, each operation can seemingly be performed "lock-free" since one can always find two unlocked queues for a thread. Mutable priorities are not intended in the original design, but can be implemented trivially in this case.

The *SprayList* [AKLS15] is a relaxed priority queue that is implemented with a lock-free skip list. Here, the removal of a good element has a high probability, but no guarantees are given. Mutable priorities are again not intended, and such an addition does not appear trivial for the skip list based design.

3. Our Priority Queue and Mesh Decimation Algorithm

In this section, we will first discuss the problem of the state-of-the-art methods. Then we describe our novel relaxed parallel priority queue with mutable priorities. Next, we will describe how to decimate meshes using our priority queue.

In particular, our contributions are:

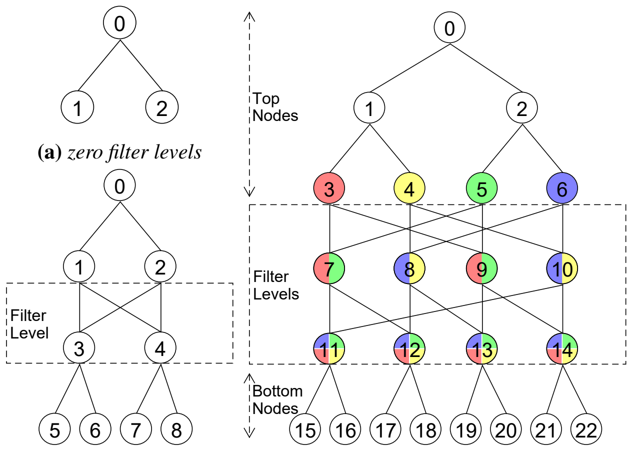
1. A novel relaxed parallel priority queue with *mutable priorities* and *filter levels*, that guarantees the extraction of a good element. The extracted element is guaranteed to be one of the top k elements, where k is a configurable parameter.
2. The integration of our novel priority queue into a mesh decimation algorithm. We will show that this parallel mesh decimation produces almost identical results to the sequential algorithm.

3.1. Problem Analysis

Mesh Decimation is usually an offline process, that is done once before shipping an application or deploying an asset. Therefore, a high-quality decimation is preferred, and longer processing times are tolerable. Current state-of-the-art mesh decimation algorithms focus on minimizing the processing time, but also accept significant quality losses to achieve their goal. We believe that a high-quality mesh decimation is possible when following certain rules:

1. Using incremental mesh decimation [BKP*10] with the quadric error metric [GH97] appears to be the best practical solution.
2. The best possible element should always be removed when possible (i. e. when it is not currently locked by another thread).
3. If the best possible element cannot be removed, there needs to be a strong guarantee for the removed element (i. e. the elements need to be in the top k of all elements).

Existing parallel incremental mesh decimation algorithms cannot guarantee a fixed k for the removed element [DLR02, FS01,



(a) *zero filter levels*

(b) *one filter level*

(c) *two filter levels*

Figure 1: *Our priority queue with filter levels.*

```
def bubbleUpDefault(nodeID):
    if nodeID == 0: return
    parent = getParentID(nodeID)
    if P(node) < P(parent):
        swapNodes(node, parent)
        bubbleUp*(parent)

def bubbleUpFilter(nodeID):
    parent = getParent1ID(nodeID)
    parent2 = getParent2ID(nodeID)
    # use parent with worst priority
    if P(parent2) > P(parent):
        parent = parent2
    if P(node) < P(parent):
        swapNodes(node, parent)
        bubbleUp*(parent)
```

Figure 2: *Pseudocode of the bubble up operation: bubbleUp*() needs to select between the default and the filter method, depending on the node location.*

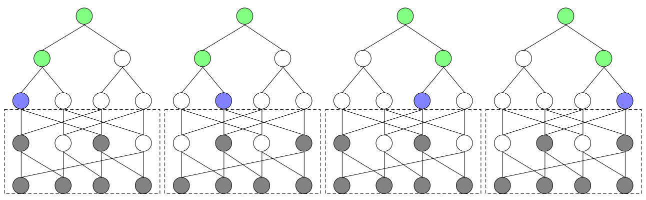


Figure 3: *Worst case illustration of the blue nodes. In each heap, green nodes are better than the blue node and gray nodes are worse due to the parent-child relationship. White nodes could potentially be better than the blue node.*

PP15] or the value of k is too large to guarantee a good result [MH21]. Similarly, the *MultiQueues* [RSD15] and *SprayList* [AKLS15] cannot guarantee a fixed k. The *Lock-free k-LSM Relaxed Priority Queue* [WGTT15] guarantees a fixed k, but it appears difficult to adapt this data structure for efficient mesh decimation. Finally, the *Heap-Based Concurrent Priority Queue with Mutable Priorities* [TMR16] always returns the best elements, aside from the elements that are bubbling up in the heap in parallel (which can be neglected). Due to the strict policy of always returning the best element, the parallel implementation with this data structure is very slow. We would like to extend this data structure to allow the removal of not only the top element, but any good element, if the top element is being processed by another thread.

3.2. Parallel Priority Queue with filter levels

We extend the *Heap-Based Concurrent Priority Queue with Mutable Priorities* [TMR16] in two ways:

1. We allow the removal of more than just the top element. If the top element is locked, we traverse the heap to find the next unlocked element within a certain maximum distance.
2. To guarantee the removal of a good element, we add additional filter levels to the heap, which guarantees that all the best elements are within the top nodes.

First, we introduce our notation: Since our priority queue is heap based, we can refer to the individual elements through an array index. The root node has index zero, its children have index one and two (see Fig. 1a). The priority of node i is $P(i)$ and we work with a min-heap, where $P(0) \leq P(i)\ \forall i \in \{1,2,\cdots,n\}$.

Next, we will describe the implementation of our filter levels.

In Fig. 1b we show an example with a single filter level. The filter level is inserted into the heap after the second level. Nodes in the filter level always have *two* parents. This means, that such nodes need to be compared to two parents, e. g. for node 3 it should hold that: $P(1) \leq P(3)$ and $P(2) \leq P(3)$. The connections to the parent nodes are chosen in such a way that all elements above the filter levels will be better than all nodes below the filter levels. In our implementation, the first parent of a node is directly above. For the second parent, an offset is added which is doubling at each level when going from bottom to top.

Fig. 2 describes how a bubble up operation with filter levels would be implemented: If the priority of a node is better than at least one of its parents, the node is swapped with its worst parent and the operation is continued recursively. In Fig. 1b, this guarantees that the first three nodes are always the best three in the heap.

In Fig. 1c we show an example with two filter levels, that were inserted after the third level. Here, the connection of the nodes is more complicated. In the end, we need to ensure that each node in the last filter level was compared to each node above the filter levels. In our example, nodes 11, 12, 13 and 14 have to be compared to nodes 3, 4, 5 and 6. Let us inspect node 11 for instance: This node is directly compared with nodes 7 and 10. Node 7 has already been compared to nodes 3 and 5. Node 10 has already been compared to nodes 4 and 6. Therefore, node 11 is guaranteed have a lower priority than nodes 3, 4, 5 and 6.

Note that, node 7 in Fig. 1c might have a better priority than node 4 or node 6, since it has not been compared to either of them. This means that the first 7 elements of the heap are not necessarily the

Table 1: *Relations in our priority queue with l_f filter levels*

Description	Symbol	Formula
Filter levels	l_f	
Top levels	l_t	$l_f + 1$
Number of top nodes	n_t	$2^{l_t} - 1$
Width of one filter level	w	$2^{l_f} = \frac{n_t+1}{2}$
Total number of filter nodes	n_f	$l_f \cdot w$
Node index	i	
Filter column index (left to right)	i_c	$(i - n_t + w)\%w$
Filter row index (bottom to top)	i_r	$\lfloor \frac{n_t+n_f-i-1}{w} \rfloor$
1st child of node i if $i < n_t - w$		$2i+1$
if $n_t - w \leq i < n_t + n_f - w$		$i+w$
else		$2i+1-n_f$
2nd child of node i if $i < n_t - w$		$2i+2$
if $n_t - w \leq i < n_t + n_f - w$		$i+w-i_c+(i_c+2^{i_r-1})\%w$
else		$2i+2-n_f$
1st parent of node i if $i < n_t$		$\lfloor \frac{i-1}{2} \rfloor$
if $n_t \leq i < n_t + n_f$		$i-w$
else		$\lfloor \frac{i-1+n_f}{2} \rfloor$
2nd parent of i if $n_t \leq i < n_t + n_f$		$i-w-i_c+(w+i_c-2^{i_r})\%w$

Table 2: *Overview of the k-best node guarantee depending on the number of filter levels.*

Filter Levels	l_f	1	2	3	4	5
Top Nodes	n_t	3	7	15	31	63
Top Guarantee	$k = 2^{l_f} \cdot l_f + 1$	3	9	25	65	161

best 7 elements. However, it is guaranteed that the first 7 elements include only elements from the top 9. This is illustrated in Fig. 3: For each node marked in blue, there are at most 8 other nodes in the heap, that could have a lower P-value, due to the parent-child relationships.

Our priority queue can be implemented with an arbitrary amount of filter levels, and the important variables are described in Tab. 1. When we use l_f filter levels, which implies $l_t = l_f + 1$ top levels, we can guarantee that any element in the top nodes belongs to the best $k = 2^{l_f} \cdot l_f + 1$ elements in the heap. This can be proven by inspecting Fig. 4: One of the top nodes can only be worse than the total number of nodes in the top and filter levels, minus the nodes it is directly connected to, in the filter levels (top nodes + filter nodes - gray nodes). Formally:

$$k = n_t + n_f - g \tag{1}$$

Where $g = 2+4+8+16 = 30$ are the number of gray nodes in Fig. 4 and can be described in general through $g = 2+4+8+\cdots+w/2+w = 2w-2$:

$$\begin{aligned} k &= n_t + n_f - (2w-2) \\ &= (2w-1) + n_f - (2w-2) \\ &= n_f + 1 = l_f \cdot w + 1 \\ &= l_f \cdot 2^{l_f} + 1 \end{aligned} \tag{2}$$

The most common values for k are shown in Tab. 2.

Finally, after the addition of filter levels, arbitrary elements from the top of the heap can be removed while ensuring a fixed quality. Originally, the priority queue obtains a lock for the root element whenever a pop- or insert-operation is executed [TMR16]. This lock also protects the node count from concurrent accesses.

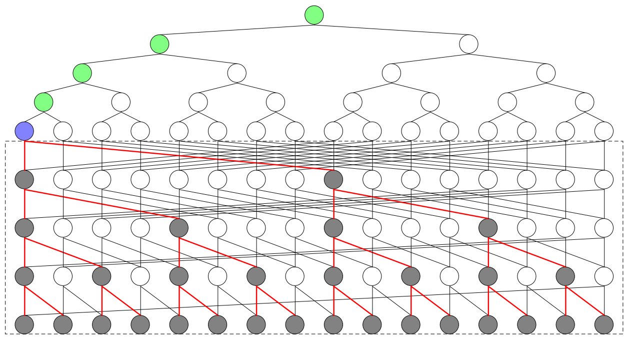


Figure 4: *Node guarantee of the blue nodes visualized: The green nodes are better than the blue node and the gray nodes are worse due to the parent-child relationship marked in red. White nodes could potentially be better than the blue node.*

In our implementation, we allow the removal of *any* of the top nodes, that are above the filter levels (see Fig. 1). To allow a concurrent removal of elements, we first change the node count of the heap to an atomic integer, to prevent lost updates. During a pop operation, we try to lock the root node first. If the lock is being held by another thread, we try locking the next node. If we could not obtain a lock for any of the top nodes, we repeat the process again. Next, the locked node is removed with the default pop-operation as described by Tamir et al. [TMR16]: The locked node is replaced with the last node of the heap, and a bubble down is executed.

3.3. Mesh Decimation

The vertex data of our mesh is stored in an array and contains positions, the error quadric, the error of its best half-edge collapse and the vertex ID for its best half-edge collapse. The priority queue only contains vertex IDs, and the priority of each element is represented by the error stored by its vertex.

The triangles are stored in a default index array.

To store connected triangles we use the data structure by Papageorgiou and Platis [PP15] which is a map from vertex IDs to triangle IDs.

For querying adjacent vertices, we utilize a similar data structure but store vertex IDs of the connected vertices instead.

Additionally, we protect each vertex with an atomic integer similar to Koh et al. [KZZC18]. If the corresponding atomic integers of a vertex and its connected vertices could be exchanged from 0 to 1, we have acquired a lock for the vertices and can perform a half-edge collapse without race conditions.

We use multiple threads for the decimation, and each thread executes the kernel from Fig. 5 until the target number of vertices is reached.

```
 1  # extension of pop() for decimation
 2  def popVertexToCollapse():
 3   i = 0 # node ID
 4   while true:
 5     if !tryLock(i):
 6       i = (i + 1) % n_t
 7       continue
 8
 9     v = getVertex(i)
10     if !lockConnectedVerticesAtomic(v):
11       # could not lock surroundings
12       unlock(i)
13       i = (i + 1) % n_t
14       continue
15
16     # vertex sourrounding is locked
17     # (do topology check)
18     return pop(i)
19
20  # core decimation algorithm
21  def decimateOneVertex():
22   # obtain vertex for collapse
23   v = popVertexToCollapse()
24   # collapse vertex to v2
25   v2 = v.collapseTarget()
26   v2.updateErrorQuadric(v.errorQuadric)
27   for tri in v.Triangles():
28     tri.replaceVertices(old=v, new=v2)
29   for cv in v.connectedVertices():
30     cv.updateConnVertices(old=v, new=v2)
31     # update priority of vertex
32     cv.updateBestError(old=v, new=v2)
33     unlockVertexAtomic(cv)
34   unlockVertexAtomic(v)
```

Figure 5: *Pseudocode of the decimation algorithm: First, we attempt to lock a vertex from the top nodes of the priority queue (line 4-7). After obtaining a vertex, we first test if none of the surrounding vertices are involved in another collapse (line 10-14). A check is done afterwards, to prevent the collapse if it causes topological inconsistencies, but was left out for simplicity in the pseudocode (line 17). Finally, we remove the vertex from the priority queue (line 18) and continue with the collapse from vertex* v *to vertex* v2 *(line 25+). At first, the error quadric of* v2 *is updated (line 26). For the collapse, we replace the vertex IDs of all connected triangles (line 27-28). Next, we need to update the connected vertices of all vertices* cv *that were connected to* v *(line 29-30). (Note: It is important to update v2 last because of the used data structure.) Additionally, the error values are recalculated and the priority queue is updated accordingly (line 32). Finally, we release the atomic lock of the connected vertices* cv *to allow collapses in this area again (line 33). At last, the now unused lock of* v *is also unlocked (line 34).*

Table 3: *Hausdorff Distance compared to the original mesh.*

Model	MultiQ	PPQ	RPPQ	FPPQ
Dragon 1%	**9.22e-4**	**9.22e-4**	9.51e-4	**9.22e-4**
ADragon 0.1%	5.70e-3	5.71e-3	5.76e-3	**5.69e-3**
Lucy 0.1%	**5.41e-4**	**5.41e-4**	5.46e-4	**5.41e-4**

4. Results

We compare our *Filter Parallel Priority Queue* (**FPPQ**) in the context of mesh decimation with:

- *OpenMesh* (**OM**) [Kob20]. An open source project with a single-threaded incremental decimation algorithm using half-edge collapses.
- *Multi Queues* (**MultiQ**) [RSD15] with our decimation algorithm (Sec. 3.3) and adjusted priority queue operations.
- *Parallel Priority Queue* (**PPQ**) by Tamir et al. [TMR16] with our decimation algorithm (Sec. 3.3) and adjusted priority queue operations.
- *Relaxed Parallel Priority Queue* (**RPPQ**), which is a variant of PPQ, that also allows the extraction of any top node, but does not have any filter levels. We include this variant to emphasize the relevance of filter levels.

For our Filter-PPQ, we use $l_f = 4$ filter levels which guarantees that any of the removed nodes are within the top $k = 65$ elements. We chose $l_f = 4$ filter levels because our CPU has 32 logical cores and this results in $n_t = 31$ top nodes, which minimizes the probability of thread starvation. Adding more filter levels would only decrease the mesh quality and introduces more overhead. All tests were performed on an AMD Ryzen 3950X using DDR4-3600 RAM.

We used three models from the Stanford 3D Repository [Sta14]:

- Dragon: 405k vertices, 810k triangles.
- Asian Dragon: 3.6m vertices, 7.2m triangles.
- Lucy: 14m vertices, 28m triangles.

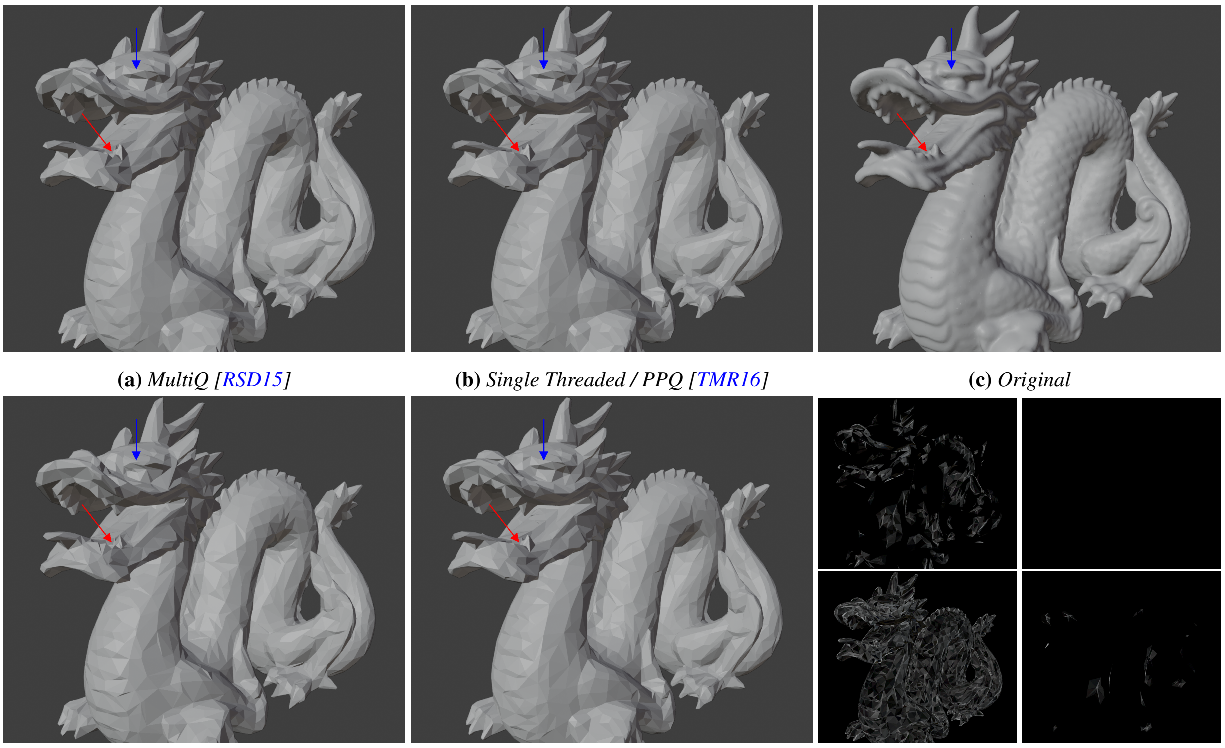
Some meshes were modified to make them manifold.

We inspect the quality after mesh decimation in Sec. 4.1. Then we will examine the performance in Sec. 4.2. Finally, we will analyze the scalability of the presented methods in Sec. 4.3.

4.1. Quality

Fig. 6-Fig. 8 show the results of mesh decimation for various priority queues with 32 threads. We observed that the PPQ produces almost indistinguishable results to a single-threaded implementation. Our Filter-PPQ produces results very similar to the PPQ (see subfigure (f)). The absence of filter levels is clearly noticeable for the Relaxed-PPQ: The Asian Dragon in Fig. 7 has a flat toe compared to the PPQ (red arrow) and Lucy in Fig. 8 has a pointy nose and different features for her mouth and eyes. The MultiQueue has some differences as well, as the flat toe in Fig. 7, but is overall ranked between the Filter-PPQ and the Relaxed-PPQ.

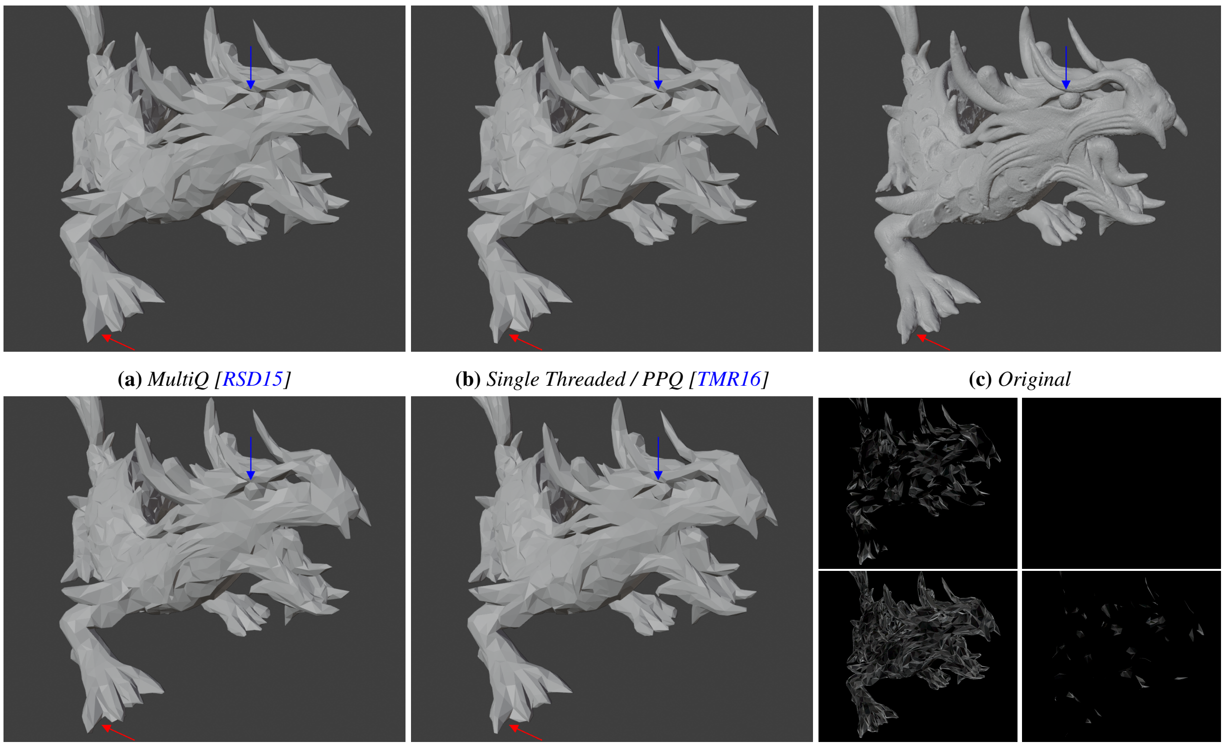
We also inspected the Hausdorff distances [CRS98] to the original mesh in Tab. 3. However, we use a modified Hausdorff distance



(a) *MultiQ [RSD15]* **(b)** *Single Threaded / PPQ [TMR16]* **(c)** *Original*

(d) *Relaxed-PPQ* **(e)** *Filter-PPQ* **(f)** *Differences of (a, b, d, e) to b*

Figure 6: *Dragon reduced to 1% with different parallel priority queues and 32 threads.*



(a) *MultiQ [RSD15]* **(b)** *Single Threaded / PPQ [TMR16]* **(c)** *Original*

(d) *Relaxed-PPQ* **(e)** *Filter-PPQ* **(f)** *Differences of (a, b, d, e) to b*

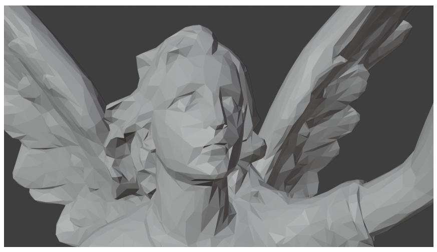
Figure 7: *Asian Dragon reduced to 0.1% with different parallel priority queues and 32 threads.*

(a) *MultiQ [RSD15]*

(b) *Single Threaded / PPQ [TMR16]*

(c) *Original*

(d) *Relaxed-PPQ*


(e) *Filter-PPQ*

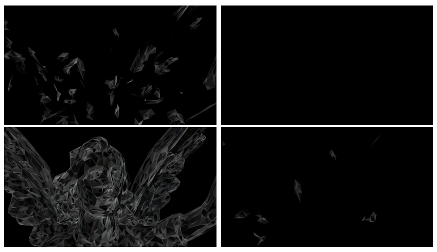

(f) *Differences of (a, b, d, e) to b*

Figure 8: *Lucy reduced to 0.1% with different parallel priority queues and 32 threads.*

Table 4: *Vertex similarity compared to sequential decimation.*

Model	MultiQ	PPQ	RPPQ	FPPQ
Dragon 1%	86.6%	100%	22.2%	98.3%
ADragon 0.1%	79.3%	100%	8.6%	96%
Lucy 0.1%	91.2%	100%	8.5%	98.9%

that contains the average minimal distances:

$$h(M,N) = \sqrt{\frac{\sum_{v \in M} \min_{q \in N} (v-q)^2}{|M|}} \quad (3)$$

Where M and N are sets with vertices and M is the original mesh.

In Tab. 3 we can see that the Hausdorff distances between PPQ, Filter-PPQ and MultiQ are almost the same. Only the distances of the Relaxed-PPQ appear larger.

For a better analysis of the similarity to a sequential decimation algorithm, we developed a vertex similarity metric: The vertex similarity is defined as the number of vertices that are identical between the sequential and the parallel decimation result, divided by the total number of remaining vertices. In Tab. 4 we can see that the PPQ and the sequential decimation algorithm have a (rounded) 100% vertex similarity in all meshes. This is followed by the Filter-PPQ with 96%-99%, the MultiQ with 80%-91% and the Relaxed-PPQ with 8%-22%.

4.2. Performance

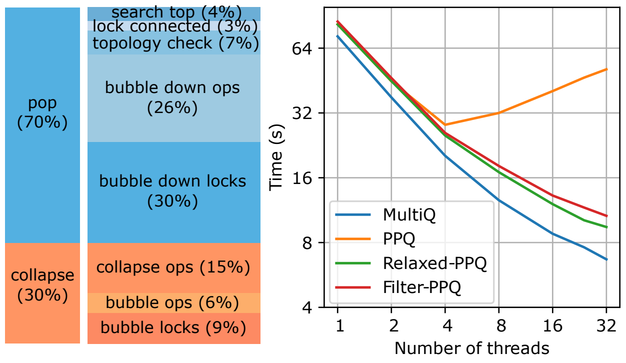
We measured the runtimes for all methods with 32 threads in Tab. 5. The only exceptions are OpenMesh, which is a single-threaded implementation, and the PPQ, which performed best with 4 threads (see Sec. 4.3). The MultiQ is faster than any other method and is followed by the Relaxed-PPQ, which is around 40% slower. The addition of filter levels does further impact the runtime by another 20%-30% compared to the Relaxed-PPQ. The normal PPQ and the single-threaded OpenMesh are significantly slower than the previous methods. Our Filter-PPQ is around 2 to 2.6 times faster than the original PPQ in the context of mesh decimation.

Table 5: *Times in seconds with 32 threads for our tested models with remaining vertex count in percent (OM is single-threaded, PPQ uses 4 threads for best performance).*

Model	OM	MultiQ	PPQ	RPPQ	FPPQ
Dragon 25%	3.29	0.128	0.432	0.172	0.226
Dragon 5%	4.20	0.139	0.535	0.195	0.256
Dragon 1%	4.36	0.141	0.558	0.202	0.263
ADragon 5%	43.0	1.44	5.56	2.00	2.38
ADragon 1%	45.9	1.47	5.76	2.05	2.44
ADragon 0.1%	46.4	1.47	5.80	2.05	2.45
Lucy 5%	189	6.52	27.4	8.88	10.5
Lucy 1%	204	6.67	28.3	9.13	10.8
Lucy 0.1%	206	6.68	28.6	9.21	10.9

Fig. 9a depicts a profiling of our Filter-PPQ: 70% of the time is spent in our extended pop operation (Fig. 5 *popVertexToCollapse()*): Only 4% of the time is spent to find an unlocked top node (line 4-7). Around 3% is spent to lock the connected vertices, followed by 7% for the topology check. The remaining 56% are spent in the actual pop operation, where 30% of the total time is required to obtain locks during the bubble down.

30% of the time is used for the actual collapse (line 25+). Here, the priorities of the connected vertices may need to be updated in the priority queue (line 32). Depending on the new priority, a node needs to either bubble up or bubble down in the heap. Moving the node inside the heap costs 6% of the time, however, obtaining the required locks to safely perform the move requires 9%. The remaining 15% are used to perform the half-edge collapse.



(a) *Time spent in our FPPQ with 32 threads.* **(b)** *Benchmarks with varying threads. Times and threads are on a log-scale.*

Figure 9: *Performance analysis for Lucy decimation to 0.1%*

4.3. Scalability

Fig. 9b shows the presented multithreaded methods with varying thread counts for our biggest mesh (Lucy). The ranking remains the same as in Sec. 4.2. The data of the original PPQ confirms our assumptions from Sec. 2.2: Too many threads try to extract from the queue and the performance gets even worse after a specific thread count, since every thread is fighting for locks.

5. Conclusion

We showed that the original parallel priority queue by Tamir et al. [TMR16] has a significant bottleneck for incremental mesh decimation. We developed a priority queue with filter levels (FPPQ) for a better performance while also maintaining a high decimation quality (96%-99% vertex similarity). We proved the relevance of the filter level concept with various error metrics. We also compared against an alternative technique, the Multi Queues [RSD15], which is around 60% faster than our approach for bigger models. However, Multi Queues result in lower vertex similarity and a higher deviation to the sequential decimation algorithm.

We believe that our novel priority queue with filter levels might be useful in other areas of research due to its top node guarantee.

Source code is available at: https://github.com/Darkwilli/FilteredParallelPriorityQueue

The work was supported by the German Research Foundation (DFG) grant GR 3833/3-2.

References

[AKLS15] Alistarh D., Kopinsky J., Li J., Shavit N.: The spraylist: A scalable relaxed priority queue. *SIGPLAN Not. 50*, 8 (jan 2015), 11–20. doi:10.1145/2858788.2688523. 2, 3

[BJG20] Bethe F., Jendersie J., Grosch T.: Preserving shadow silhouettes in illumination-driven mesh reduction. *CGF 39*, 6 (2020). doi:https://doi.org/10.1111/cgf.14008. 2

[BKP*10] Botsch M., Kobbelt L., Pauly M., Alliez P., Lévy B.: *Polygon Mesh Processing*. AK Peters / CRC Press, Sept. 2010. Chapter 7.2. URL: https://hal.inria.fr/inria-00538098. 1, 2

[CRS98] Cignoni P., Rocchini C., Scopigno R.: Metro: Measuring error on simplified surfaces. *CGF 17*, 2 (1998). doi:https://doi.org/10.1111/1467-8659.00236. 5

[DFP95] De Floriani L., Puppo E.: Hierarchical triangulation for multiresolution surface description. *ACM Trans. Graph. 14*, 4 (oct 1995), 363–411. doi:10.1145/225294.225297. 2

[DLR02] Dehne F., Langis C., Roth G.: Mesh simplification in parallel. In *Algorithms and Architectures for Parallel Processing*. 2002, pp. 281–290. doi:10.1142/9789812792037_0025. 2

[DT07] DeCoro C., Tatarchuk N.: Real-time mesh simplification using the GPU. In *I3D* (2007). doi:10.1145/1230100.1230128. 2

[FS01] Franc M., Skala V.: Parallel triangular mesh decimation without sorting. In *Proceedings Spring Conference on Computer Graphics* (2001), pp. 22–29. doi:10.1109/SCCG.2001.945333. 2

[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. SIGGRAPH '97. doi:10.1145/258734.258849. 1, 2

[Hop96] Hoppe H.: Progressive meshes. SIGGRAPH '96. doi:10.1145/237170.237216. 2

[KCS98] Kobbelt L., Campagna S., Seidel H.-P.: A general framework for mesh decimation. In *Graphics Interface 1998 Conference, Canada* (June 1998). URL: http://graphicsinterface.org/wp-content/uploads/gi1998-6.pdf. 1

[Kob20] Kobbelt L.: Openmesh 8.1, 2020. Accessed: 2022-06-29. URL: www.openmesh.org. 5

[KZZC18] Koh N., Zhang W., Zheng J., Cai Y.: GPU-based multiple-choice scheme for mesh simplification. CGI 2018, Assoc. for Computing Machinery. doi:10.1145/3208159.3208195. 2, 4

[MH21] Mousa M. H., Hussein M. K.: High-performance simplification of triangular surfaces using a GPU. *PLOS ONE 16*, 8 (08 2021). doi:10.1371/journal.pone.0255832. 2, 3

[PP15] Papageorgiou A., Platis N.: Triangular mesh simplification on the GPU. *Vis. Comput. 31*, 2 (feb 2015), 235–244. doi:10.1007/s00371-014-1039-x. 2, 4

[RB93] Rossignac J., Borrel P.: Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics* (Berlin, Heidelberg, 1993), Springer Berlin Heidelberg, pp. 455–465. 2

[RGG15] Reich A., Günther T., Grosch T.: Illumination-driven mesh reduction for accelerating light transport simulations. *CGF 34*, 4 (2015). doi:https://doi.org/10.1111/cgf.12688. 2

[RSD15] Rihani H., Sanders P., Dementiev R.: Multiqueues: Simple relaxed concurrent priority queues. SPAA '15, Assoc. for Computing Machinery. doi:10.1145/2755573.2755616. 2, 3, 5, 6, 7, 8

[Sta14] Stanford: The stanford 3d scanning repository, 2014. Accessed: 2022-06-29. URL: http://graphics.stanford.edu/data/3Dscanrep/. 5

[SZL92] Schroeder W. J., Zarge J. A., Lorensen W. E.: Decimation of triangle meshes. *SIGGRAPH Comput. Graph. 26*, 2 (jul 1992), 65–70. doi:10.1145/142920.134010. 2

[TMR16] Tamir O., Morrison A., Rinetzky N.: A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms. In *OPODIS 2015* (Dagstuhl, Germany, 2016), vol. 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*. doi:10.4230/LIPIcs.OPODIS.2015.15. 2, 3, 4, 5, 6, 7, 8

[WGTT15] Wimmer M., Gruber J., Träff J. L., Tsigas P.: The lock-free k-lsm relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN* (New York, NY, USA, 2015), PPoPP 2015, Assoc. for Computing Machinery. doi:10.1145/2688500.2688547. 2, 3

[WK02] Wu J., Kobbelt L.: Fast mesh decimation by multiple-choice techniques. In *VMV, Germany* (2002). URL: https://www.graphics.rwth-aachen.de/publication/03121/. 2