The Pennsylvania State University
The Graduate School
Department of Computer Science and Engineering

CPU- AND GPU-BASED TRIANGULAR SURFACE MESH SIMPLIFICATION

A Thesis in
Computer Science and Engineering
by
Dragos Nistor

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2012

The thesis of Dragos Nistor was reviewed and approved* by the following:

Suzanne M. Shontz
Assistant Professor of Computer Science and Engineering
Thesis Adviser

John Hannan
Associate Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Mesh simplification and mesh compression are important processes in the realms of computer graphics and high-performance computing, as they allow the mesh to take up less memory. In particular, current simplification and compression algorithms do not take advantage of both the central processing unit (CPU) and the graphics processing unit (GPU).

We propose and analyze the results of two mesh simplification algorithms based on the edge-collapse operation that take advantage of the GPU by allocating a portion of the computation to the CPU and a portion of the computation to the GPU. Our algorithms are the naïve marking algorithm and the inverse-reduction algorithm.

Experimental results show that when the algorithms take advantage of both the CPU and the GPU, there is a decrease in running time for simplification compared to performing all of the computation on the CPU. The marking algorithm provides higher simplification rates than the inverse-reduction algorithm, whereas the inverse-reduction algorithm has a lower running time than the marking algorithm.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my whole family and friends for the support they have given me during my time at The Pennsylvania State University. I would also like to thank my research advisor, Dr. Suzanne M. Shontz, for the guidance and support she has given me throughout my research and writing of this thesis. Finally, I would like to thank my honors advisor and committee member, Dr. John Hannan, for his suggestions and comments on my work.

Additionally, I would like to thank multiple institutions for allowing use of their models for research. I would like to thank the Georgia Institute of Technology for their Large Geometric Models Archive, from which we obtained the skeleton hand and horse models, the Stanford University Computer Graphics Laboratory, from which we obtained the bunny and armadillo models, and the ISTI Visual Computing Laboratory and Frank ter Haar, from which we obtained models of the gargoyle and kitten through the AIM@SHAPE Shape Repository, respectively.

Chapter 1

Introduction

1.1 Motivation and Previous Work

Three-dimensional geometric models of varying detail are useful for solving problems in areas such as computer graphics [20], surface reconstruction [15], computer vision [18], and communication [4]. Such models give rise to the problems of mesh simplification, mesh compression [14], and mesh optimization [16]. This thesis focuses on solving the problem of mesh simplification using both the central processing unit (CPU) and the graphics processing unit (GPU) found on most modern computers.

Mesh simplification is the process of removing elements and vertices from meshes to create a simpler model. Mesh simplification can be applied in areas such as surface reconstruction [3], three-dimensional scanning [17], computer animation [12], and terrain rendering [10]. For example, when rendering a movie scene, a model with extremely high detail is not required for an object that is located far away from the camera.

Multiple serial CPU-based algorithms have been proposed to solve this problem. Some algorithms use a simple edge-collapse operation [14], which involves repeatedly collapsing edges into vertices to obtain a simplified mesh. Others use a triangle-collapse operation [23]. While the triangle-collapse operation yields more simplified meshes per operation [6], there are more possible cases to handle when performing this operation. Our algorithms will be based on the edge-collapse operation for its simplicity.

Other algorithms focus on controlled vertex, edge, or element decimation [24], where a vertex,

edge, or element is removed from the mesh if it meets the decimation criteria. Any resulting holes in the mesh are patched through available methods such as triangulation. One other method for mesh simplification is vertex clustering [19]. When performing mesh simplification using vertex clustering, vertices are clustered by topological location and a new vertex is created to represent each cluster. Elements can then be created through surface reconstruction [15]. Our algorithms do not focus on decimation or clustering, as neither clustering, which requires remeshing after clustering, nor decimation, which also requires a clean-up process after decimation, are designed with the concurrency provided by the GPU in mind.

A few parallel CPU-based algorithms based on the serial algorithms have been proposed as well. One parallel algorithm is based on vertex decimation [11], where the importance of each vertex is evaluated, and vertices with low importance are removed. A GPU-based implementation of this algorithm has also been proposed [13] and is discussed later in this section. However, as we are interested in the results of an algorithm that takes advantage of both the CPU and GPU by splitting the simplification workload between the two. For example, we could potentially remove vertices with a high global importance simply because they have a low ranking among all vertices in the CPU or in the GPU. This issue could be solved by ample communication between the CPU and GPU to estimate the global ranking of the vertices. However, the transfer rate between the CPU and GPU is a significant bottleneck [25].

Other simplification algorithms focus on distributed systems [5] and efficient communication between nodes. Such algorithms would suffer from the same communication latency between the CPU and the GPU if implemented to take advantage of the GPU. Another algorithm [9] focuses on greedily splitting a mesh into equal subparts and assigning each part to a CPU core to be simplified by applying the edge-collapse operation. The techniques introduced could be used when implementing a GPU-based algorithm. However, as the GPU can support many more threads at any one time than a multi-core CPU can, there is no need to split our test meshes into subparts; each thread focuses on one element.

Some GPU-based simplification algorithms have also been proposed. One such algorithm offloads the computationally-intensive parts of vertex decimation to the GPU [13], while leaving the data structure representing the mesh in main memory. While this approach is valid, it assumes that the CPU will be available during the whole process. A popular method [8] based on vertex clustering exists as well. A downside of it is that it assumes that the surface mesh is

closed and that there is access to the full mesh during the simplification process, which excludes streaming input models.

We propose three simplification algorithms, one of which runs on the CPU and two of which run on the GPU. The algorithms are based on the edge-collapse operation, as it is an extremely simple and small-scale operation, and it works even if there is no access to the full mesh. The CPU algorithm visits every available element and performs the edge-collapse operation on the element if it is not yet marked as affected (defined in section 2.3, as does one of the GPU algorithms. The other GPU algorithm takes full advantage of the concurrency of the GPU and attempts to collapse more edges each iteration. All algorithms are described in more detail in Chapter 2.

1.2 Organization

In Chapter 2, we present three simplification algorithms: the CPU simplification algorithm, a naïve GPU simplification algorithm, and a GPU simplification algorithm based on reductions. We discuss the correctness of the algorithms, as well. In Chapter 3, we describe our experimental setup and results for various CPU-GPU workload splits for both GPU algorithms and for multiple iterations of each GPU algorithm. In Chapter 4, we draw conclusions based on our results and propose ideas for future work.

Chapter 2

Simplification Algorithms

We propose three CPU- and GPU-based algorithms which work in tandem to simplify a mesh. The algorithms simplify meshes uniformly and exhaustively, ensuring maximal simplification occurs. All proposed algorithms rely on the edge-collapse operation, which is defined in section 2.1. Additionally, all algorithms are lossless, so mesh compression is a natural extension to the algorithms.

2.1 Edge-Collapse Operation

Our simplification algorithms rely on the edge-collapse operation [14], which is defined as follows for an input mesh containing a set of vertices V and a set of elements T:

For some edge $e = (v_1, v_2)$ shared by elements $t_1 = (v_1, v_2, v_3)$ and $t_2 = (v_4, v_2, v_1)$, define

$$v_m = \frac{v_1 + v_2}{2}. \tag{2.1}$$

To collapse edge e, t_1 and t_2 are removed from the mesh, and any references to v_1 or v_2 are updated to refer to v_m. Figure 2.1 shows an edge-collapse operation on edge (v_1, v_2). For the edge-collapse operation, the listing order of the vertices that make up an element does not matter. If additional information, such as the original positions of v_1 and v_2 are stored, the edge-collapse operation is reversible. Therefore, any compression or simplification algorithms based on this operation are lossless, meaning that no information regarding the original mesh is

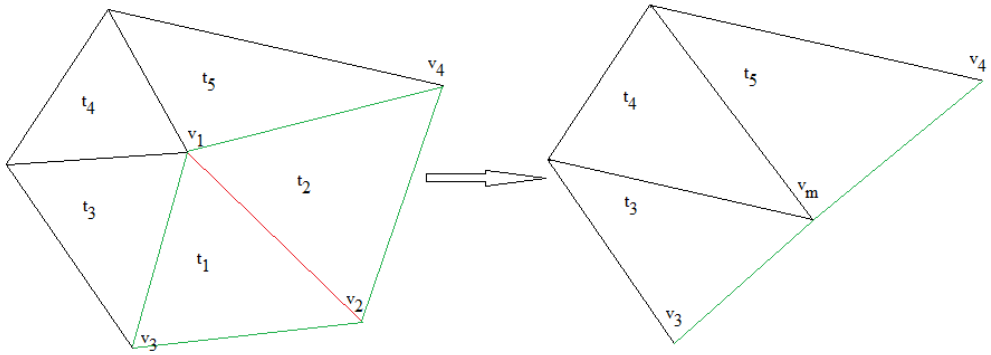


Figure 2.1: An edge-collapse on $e = (v_1, v_2)$.

lost when performing the compression or simplification. The original mesh can be recovered by reversing the steps taken to compress or simplify the mesh.

2.2 Workload Splitting

Our simplification algorithms allocate a portion of a mesh to the CPU and the rest to the GPU to simplify. We propose an extremely simple method for allocation. For a CPU-GPU split $k\%$ where $k \in \mathbb{R}$ and $0 \leq k \leq 100$ of a mesh $M = (\text{Vertices}, \text{Elements})$, the CPU simplifies the first $k\%$ of all elements, and the GPU simplifies the rest. For example, a CPU-GPU split of 30% means that the CPU simplifies the first 30% of all elements and the GPU simplifies the rest.

2.3 Definition of Affected Elements

Since edge-collapse operations should be performed uniformly across the mesh, we determine whether or not elements are affected by previous edge collapses. Elements which are affected will not take part in any new edge-collapse operations.

Define $N(t)$ for $t = (v_a, v_b, v_c)$ to be elements which contain any of the vertices v_a, v_b, or v_c. If edge $e = (v_1, v_2)$ between elements $t_1 = (v_1, v_2, v_3)$ and $t_2 = (v_4, v_2, v_1)$, for example, has been collapsed, then the elements in $N(t_1) \cup N(t_2)$ are considered affected. The elements shaded in gray in Figure 2.2 would be considered affected if edge (v_1, v_2) were collapsed.

Since each edge-collapse operation causes neighboring elements to become affected, there is a hard limit on the number of edge-collapse operations, and on the amount of simplification

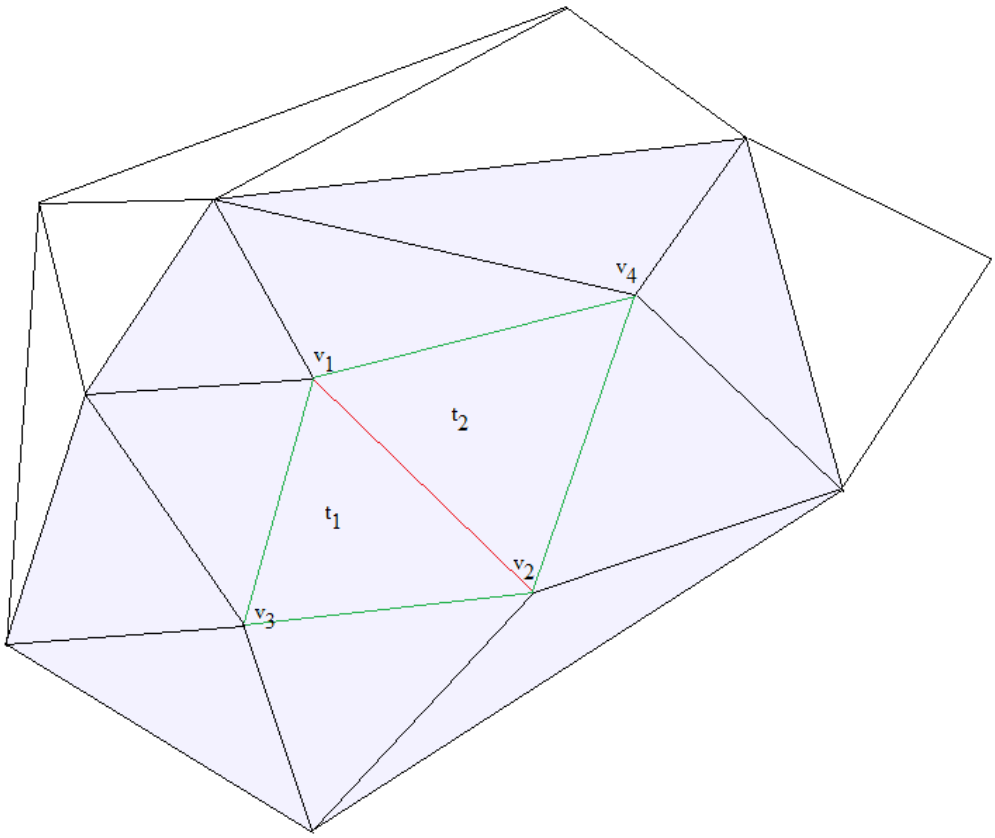


Figure 2.2: Elements considered affected by edge-collapse of (v_1, v_2).

per iteration of the algorithm. To assess the full range of simplification capabilities, multiple iterations of the algorithms will be performed on each test case.

2.4 CPU Edge-Collapse Algorithm

To simplify portions of the mesh using the CPU, we propose a simple edge-collapse algorithm. We describe it in mathematical terms below; the pseudocode is provided in Algorithm 2.2.

2.4.1 Description

The CPU edge-collapse algorithm works by searching through all elements assigned to the CPU one at a time. Each element is examined to see if it is affected, and if an element $t = (v_1, v_2, v_3)$ that is not affected is found, an edge-collapse is performed on (v_1, v_2). When all elements are affected or have taken part in an edge-collapse operation, the algorithm terminates.

This ensures that the edge-collapse operation is performed uniformly and exhaustively across the elements assigned to the CPU, and that no one area is more or less simplified or deformed.

2.4.2 Algorithm

The pseudocode for the CPU edge-collapse simplification algorithm is provided in Algorithm 2.2 below.

Algorithm 2.1 The CPU Edge-Collapse Simplification Algorithm

```
function MARK-AS-AFFECTED(element)
    for all v ∈ element do
        affected[v] ← true
    end for
end function

function MARK-AS-COLLAPSED((v1, v2))
    for all t ∈ elements ⊃ {v1, v2} do
        collapsed[t] ← true
    end for
end function

function CPU-SIMPLIFY(elements, vertices)
    for all t = (v1, v2, v3) ∈ elements do
        if t ∈N(affected elements) then
            mark-as-affected(t)
        else
            collapse((v1, v2))
            mark-as-collapsed((v1, v2))
        end if
    end for
end function
```

2.5 GPU Marking Algorithm

We propose a naïve GPU algorithm to simplify portions of the mesh based on the edge-collapse operation. We describe it in mathematical terms below; the pseudocode is provided in Algorithm 2.1.

2.5.1 Description

The naïve GPU marking simplification algorithm works by searching through all elements assigned to the GPU one at a time. If an element $t = (v_1, v_2, v_3)$ that is not affected is found,

an edge-collapse is performed on (v_1, v_2), and all $t_n \in N(t)$ are concurrently marked as affected. When all elements are affected or have taken part in an edge-collapse operation, the algorithm terminates. This also ensures that the edge-collapse operation is performed uniformly and exhaustively across all elements assigned to the GPU, and that no one area is more or less simplified or deformed.

2.5.2 Algorithm

The pseudocode for the naïve GPU marking simplification algorithm is provided in Algorithm 2.1 below.

Algorithm 2.2 The CPU Edge-Collapse Simplification Algorithm

```
function GPU-Mark(mark, elements, vertices)
    for all t ∈ N(mark) do GPU thread t : mark-as-affected(t)
    end for
end function

function GPU-Mark-Simplify(elements, vertices)
    for all t = (v1, v2, v3) ∈ elements do
        if not marked(t) then
          collapse((v1, v2))
          mark-as-collapsed((v1, v2))
          GPU-Mark(t)
        end if
    end for
end function
```

2.6 GPU Inverse-Reduction Algorithm

We propose a GPU mesh simplification algorithm that leverages the full strength of the GPU. We describe it in mathematical terms below; the pseudocode is provided in Algorithm 2.3.

2.6.1 Description

In our previous algorithms, one element was examined at each iteration of the main loop. Instead of performing a linear search to find the next element which is not affected, we will now examine twice as many elements each iteration, with each element examined by a different GPU thread. To fully take advantage of the architecture of the GPU, a soft-grained blocking

[21] method based on test-and-set [1] is used to decide if any edge of an element should be collapsed. We attempt to lock each vertex in an element by calling test-and-set on the affected bit of each vertex in the element. More details regarding correctness are provided in section 2.6.3.

2.6.2 Algorithm

The pseudocode of the GPU inverse-reduction simplification algorithm is provided in Algorithm 2.3.

Algorithm 2.3 The GPU Inverse-Reduction Simplification Algorithm

```
function GPU-Simp-Try(target = (v_0, v_1, v_2), elements, vertices)
    if affected(target) then
        return
    end if
    if test-and-set(collapsed[v_0]) = 0 then
        if test-and-set(collapsed[v_1]) = 0 then
            if test-and-set(collapsed[v_2]) = 0 then
                collapse((v_0, v_1))
                GPU-Mark(target)
            end if
          collapsed[v_1] ← 0
          collapsed[v_0] ← 0
        end if
        collapsed[v_0] ← 0
    end if
end function

function GPU-IR-Simplify(elements, vertices)
     i = |elements|
    while i ≥ 1 do
        if threadid mod i = 0 then
          GPU-Simp-Try(elements[threadid])
        end if
        i = i div 2
    end while
end function
```

2.6.3 Correctness

The GPU-Simp-Try method attempts to lock each vertex of an element by checking to make sure the vertex has not already been collapsed. If it finds that a vertex has already been locked, it releases all previouly-locked vertices. Therefore, if a thread successfully locks v_1, v_2, and v_3

for element t, it must mean that no other thread has locked any $t_n \in N(t)$, either currently or previously. Therefore, the algorithm simplifies the mesh both uniformly and exhaustively, ensuring that no one area is too simplified or deformed.

Chapter 3

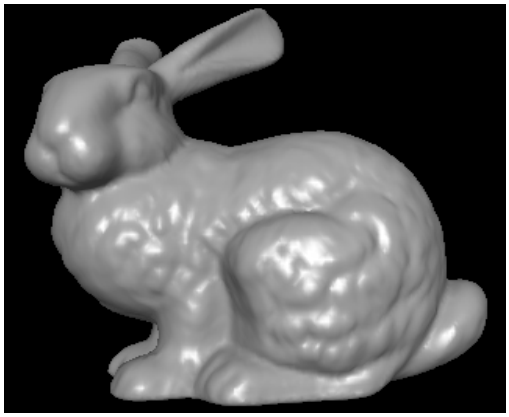
Experiments

3.1 Experimental Design

The algorithms were implemented in C++ and compiled with the NVIDIA C++ compiler included with the CUDA Toolkit [7]. They were tested on a Dell XPS 17 laptop running Windows 7 Professional equipped with an NVIDIA GeForce GT 550M GPU and an Intel Core i5-2430M CPU running at 2.4 GHz with 3.90 GB of usable main memory.

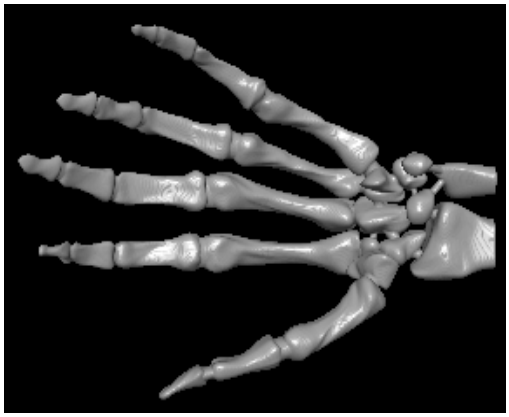
We ran our algorithms on the following six test cases: armadillo, bunny, gargoyle, hand, horse, and kitten, which are shown in Figure 3.1. Armadillo and bunny are included courtesy of Stanford [26]. Gargoyle and kitten are included courtesy of their owners through the AIM@Shape Shape Repository [2]. Hand and horse are included courtesy of the Georgia Institute of Technology [22].

(a) Armadillo

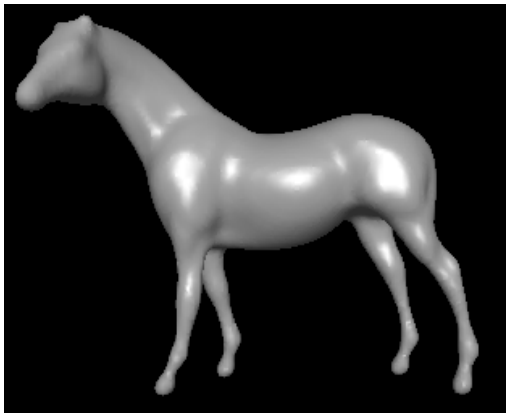


(b) Bunny

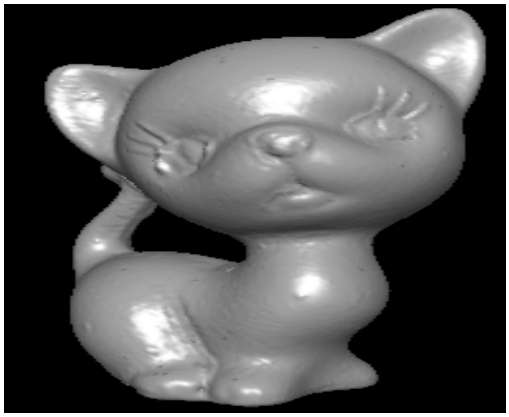
(c) Gargoyle



(d) Hand



(e) Horse



(f) Kitten

Figure 3.1: Initial meshes used for testing.

mesh	# vertices	# faces	min ∠	max ∠	min area	avg area	volume
armadillo	172974	345944	0.034750	170.907	7.424e-08	1.840e-1	1.42690e+6
bunny	34834	69664	0.494800	177.515	7.925e-08	1.573e-2	7.54700e+3
gargoyle	863182	1726364	0.000215	179.820	3.638e-12	4.553e-2	1.63730e+6
hand	327323	654666	0.545000	177.995	5.161e-11	1.078e-4	1.64936e+1
horse	15366	30728	0.385500	177.119	1.118e-14	2.118e-6	1.58866e−3
kitten	137098	274196	0.004609	179.936	1.427e-08	8.846e-2	8.38625e+5

Table 3.1: Various values of the metrics for the initial meshes.

We obtained the following regarding the output mesh: vertex count, element count, minimum angle, maximum angle, minimum element area, average element area, volume, and wall clock running time in seconds of the algorithm. These metrics are generally used in assessing the quality of a mesh. To properly measure running time, we collect the amount of wall clock time the algorithm spent on the computations for 100 times per test case and compute the average. Table 3.1 contains the values of the metrics for the initial meshes.

3.2 Results

3.2.1 Marking Algorithm

To examine the effects of splitting the mesh simplification workload between the CPU and the GPU using the naïve marking algorithm, we recorded the time spent during the simplification process for different CPU-GPU splits on all test cases. We tested the following CPU-GPU workload splits: 100-0, 95-5, 90-10, 85-15, 80-20, 75-25, 70-30, 65-35, 60-40, 55-45, 50-50, 45-55, 40-60, 35-65, 30-70, 25-75, 20-80, 15-85, 10-90, 5-95, and 0-100. The time taken in seconds for the tested splits can be seen in Table 3.2 and Figure 3.2. The proportion of running time spent in the GPU for the tested is shown in Table 3.3 and Figure 3.3. The GPU memory usage for the tested splits is shown in Table 3.4 and Figure 3.4.

CPU-GPU split							
mesh	100-0	95-5	90-10	85-15	80-20	75-25	70-30
armadillo	12.4	12.4	12.3	12.3	12.3	12.2	12.2
bunny	3.51	3.51	3.52	3.52	3.53	3.53	3.53
gargoyle	68.4	68.0	67.4	66.9	66.3	65.8	65.4
hand	28.6	28.5	28.3	28.2	28.0	27.8	27.7
horse	2.12	2.13	2.13	2.15	2.16	2.16	2.17
kitten	13.3	13.3	13.2	13.2	13.2	13.1	13.1
mesh	65-35	60-40	55-45	50-50	45-55	40-60	35-65
armadillo	12.2	12.1	12.1	12.1	12.1	12.0	12.0
bunny	3.53	3.54	3.54	3.55	3.55	3.56	3.57
gargoyle	65.0	64.6	64.0	63.3	62.8	62.0	61.4
hand	27.7	27.6	27.4	27.2	27.1	27.0	26.8
horse	2.18	2.20	2.21	2.23	2.25	2.26	2.28
kitten	13.0	13.1	13.0	12.9	12.9	12.8	12.8
mesh	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	12.0	11.9	11.9	11.9	11.8	11.8	11.8
bunny	3.58	3.58	3.59	3.59	3.60	3.60	3.61
gargoyle	60.8	60.2	59.7	59.2	58.6	58.0	57.3
hand	26.6	26.6	26.5	26.3	26.2	26.0	25.9
horse	2.29	2.31	2.32	2.34	2.36	2.36	2.38
kitten	12.8	12.7	12.6	12.6	12.5	12.4	12.5

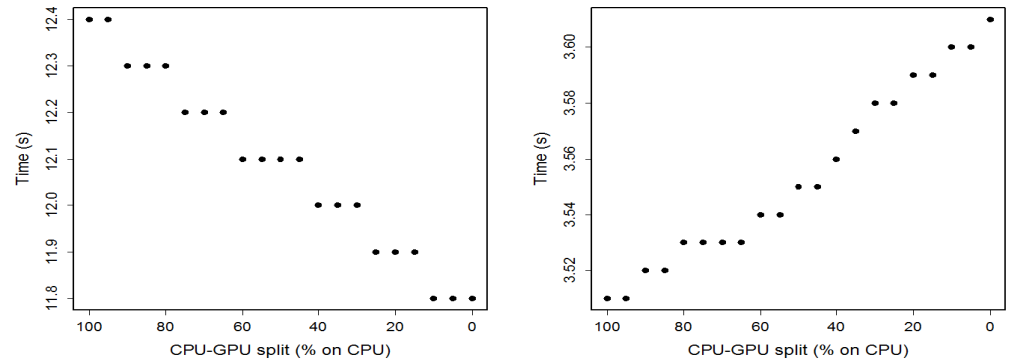
Table 3.2: The time taken to simplify for each CPU-GPU split using the naïve GPU algorithm.

GPU % time							
mesh	100-0	95-5	90-10	85-15	80-20	75-25	70-30
armadillo	0	9.2	14.5	19.6	24.6	29.4	34.1
bunny	0	5.4	8.8	12.1	15.2	18.2	21.1
gargoyle	0	10.4	15.8	21.1	26.3	31.3	36.1
hand	0	9.4	14.8	20.0	25.1	29.9	34.7
horse	0	5.0	8.0	10.9	13.6	16.2	18.7
kitten	0	8.2	13.4	18.4	23.2	27.9	32.5
mesh	65-35	60-40	55-45	50-50	45-55	40-60	35-65
armadillo	38.6	43.1	47.5	51.7	55.8	59.8	63.6
bunny	23.9	26.5	29.0	31.4	33.6	35.7	37.7
gargoyle	40.8	45.4	49.9	54.3	58.6	62.7	66.7
hand	39.3	43.9	48.4	52.7	56.9	61.0	64.8
horse	21.1	23.3	25.4	27.4	29.2	30.9	32.5
kitten	36.8	41.2	45.5	49.6	53.6	57.4	61.1
mesh	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	67.4	70.9	74.3	77.5	80.6	83.4	86.0
bunny	39.5	41.3	42.9	44.4	45.8	47.2	48.5
gargoyle	70.5	74.2	77.7	81.0	84.2	87.2	89.1
hand	68.7	72.3	75.8	79.0	82.2	84.2	87.8
horse	33.9	35.3	36.5	37.6	38.6	39.4	40.1
kitten	64.8	68.2	71.4	74.5	77.4	80.1	82.5

Table 3.3: The percentage of time spent on the GPU for different CPU-GPU splits using the naïve GPU algorithm.

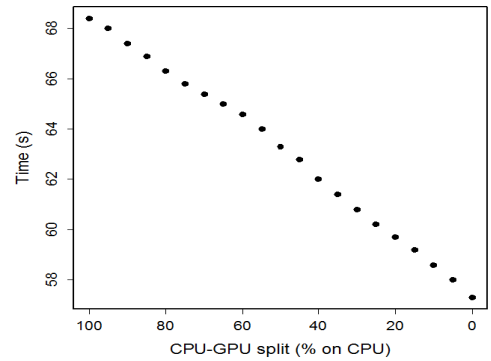
CPU-GPU split							
mesh	100-0	95-5	90-10	85-15	80-20	75-25	70-30
armadillo	0	2568	2770	2973	3176	3378	3581
bunny	0	517	558	599	640	680	721
gargoyle	0	12813	13824	14836	15847	16859	17871
hand	0	4859	5242	5626	6009	6393	6777
horse	0	228	246	264	282	300	318
kitten	0	2035	2196	2356	2517	2678	2838
mesh	65-35	60-40	55-45	50-50	45-55	40-60	35-65
armadillo	3784	3986	4189	4392	4595	4797	5000
bunny	762	803	844	884	925	966	1007
gargoyle	18882	19894	20905	21917	22928	23940	24951
hand	7160	7544	7927	8311	8695	9078	9462
horse	336	354	372	390	408	426	444
kitten	2999	3160	3320	3481	3642	3802	3963
mesh	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	5203	5405	5608	5811	6014	6216	6419
bunny	1048	1089	1129	1170	1211	1252	1293
gargoyle	25963	26974	27986	28998	30009	31021	32032
hand	9845	10229	10613	10996	11380	11763	12147
horse	462	480	498	516	534	552	570
kitten	4124	4284	4445	4606	4766	4927	5088

Table 3.4: The amount of memory used, in KB, by the GPU during simplification for each CPU-GPU split for both the naïve GPU algorithm and the inverse-reduction GPU algorithm.

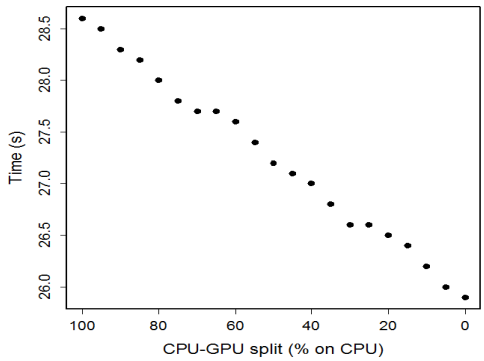


(a) Armadillo

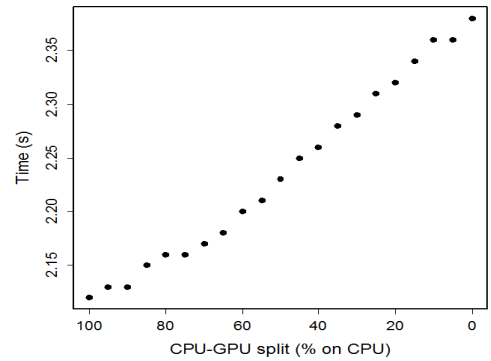
(b) Bunny



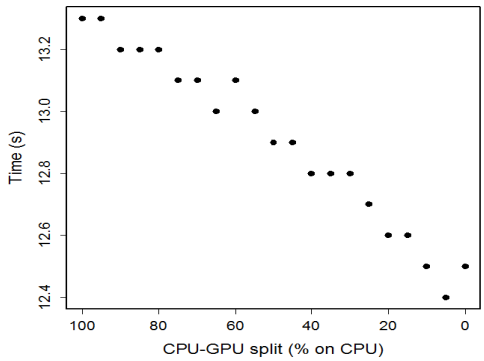
(c) Gargoyle



(d) Hand

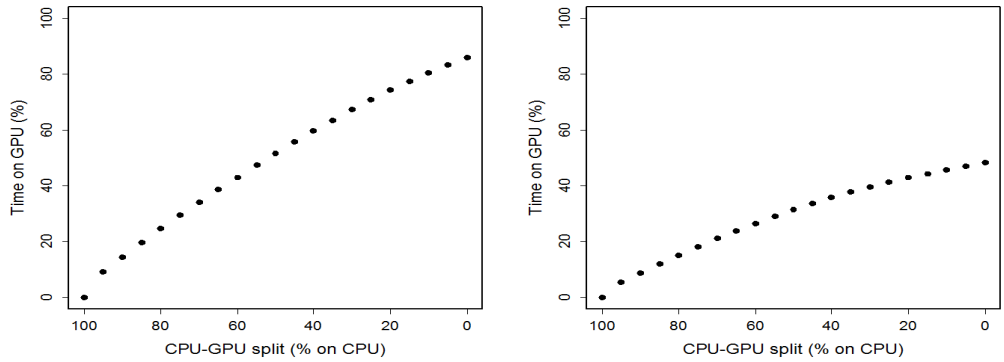


(e) Horse



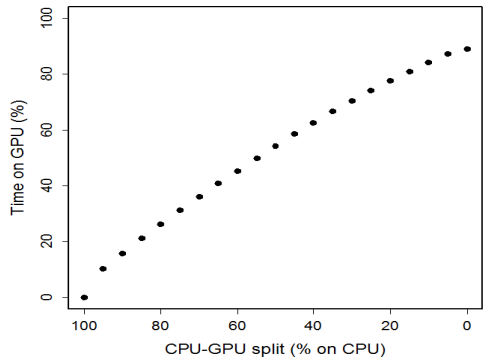
(f) Kitten

Figure 3.2: The time taken to simplify each test case for every split using the naïve GPU algorithm. As the CPU-GPU split increases, the CPU workload increases, and the GPU workload decreases.

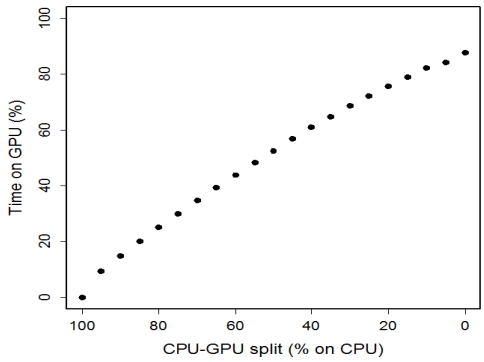


(a) Armadillo

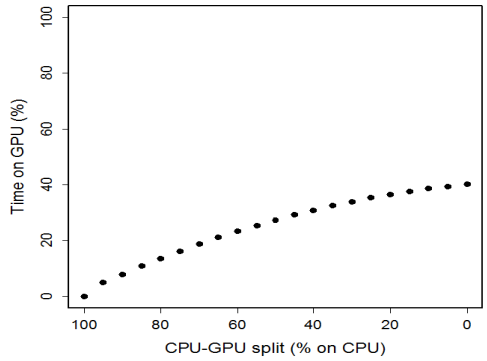
(b) Bunny



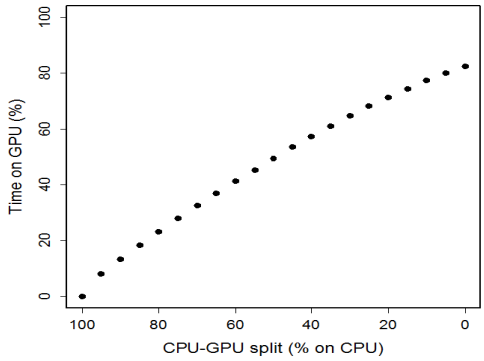
(c) Gargoyle



(d) Hand

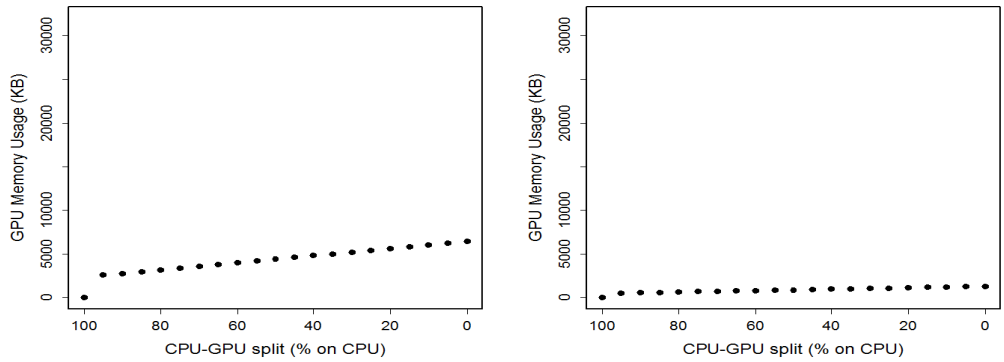


(e) Horse



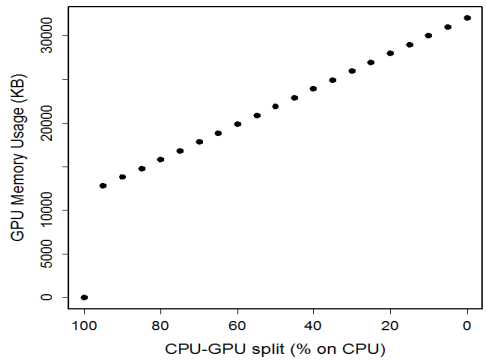
(f) Kitten

Figure 3.3: The percentage of the time spent on the GPU for every split using the naïve GPU algorithm.

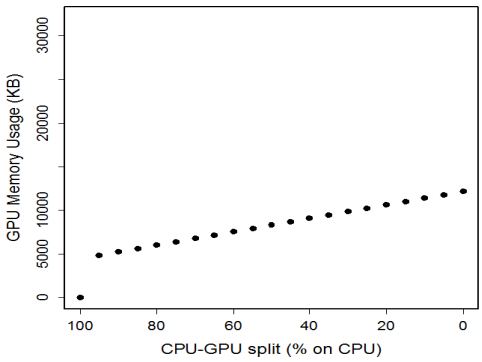


(a) Armadillo

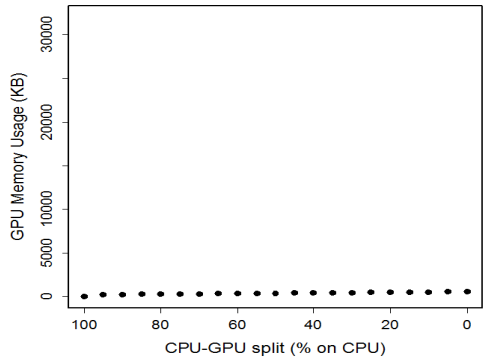
(b) Bunny



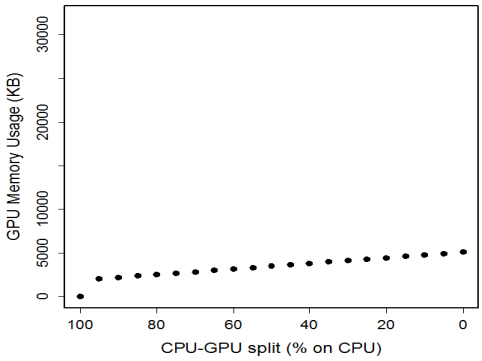
(c) Gargoyle



(d) Hand



(e) Horse



(f) Kitten

Figure 3.4: The amount of memory used by the GPU, in KB, for every split using both the naïve GPU algorithm and the inverse-reduction GPU algorithm.

mesh	# vertices	# faces	min ∠	max ∠	min area	avg area	volume
armadillo	147739	295496	2.938e-3	179.991	2.140e-08	2.149e-1	1.42733e+6
bunny	30102	60194	1.618e-3	179.997	8.211e-07	1.820e-2	7.56100e+3
gargoyle	735345	1470688	3.079e-4	179.987	2.303e-09	5.349e-2	1.64247e+6
hand	278849	557718	1.442e-2	179.673	1.992e-10	1.265e-4	1.65230e+1
horse	13180	26598	6.280e-2	179.811	3.006e-12	2.446e-6	1.58903e−3
kitten	116952	233901	4.609e-3	179.843	6.234e-08	1.038e-1	8.38671e+5

Table 3.5: Various values of the metrics for the meshes after one iteration.

mesh	# vertices	# faces	min ∠	max ∠	min area	avg area	volume
armadillo	56469	111617	3.091e-2	179.86	3.901e-07	6.077e-1	1.42548e+6
bunny	9979	20129	1.477e-1	179.645	6.242e-06	5.759e-2	7.56014e+3
gargoyle	278305	629188	3.688e-4	179.981	8.967e-08	1.143e-1	1.64053e+6
hand	99635	213061	1.969e-2	179.912	2.674e-09	3.937e-4	1.65117e+1
horse	5428	10927	1.874e-2	179.339	1.032e-11	7.162e-6	1.58650e−3
kitten	39991	87997	1.530e-2	179.799	2.320e-06	5.259e-1	8.38547e+5

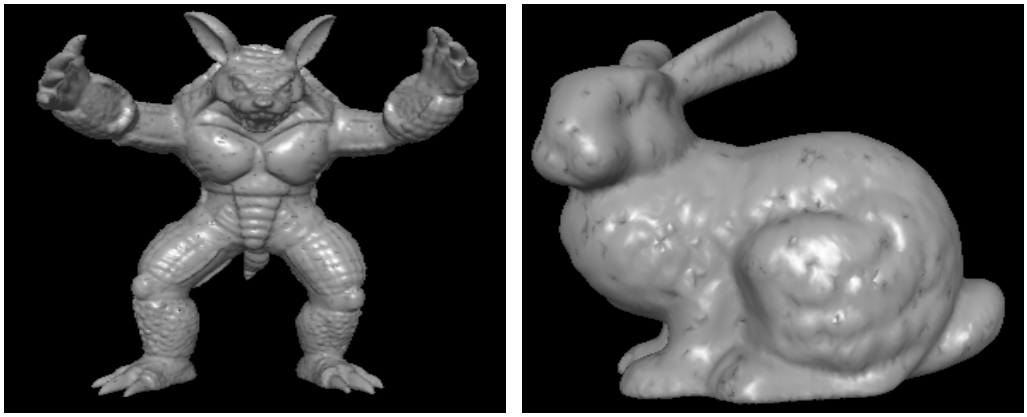
Table 3.6: Various values of the metrics for the meshes after ten iterations.

As seen in Figure 3.2, the armadillo, gargoyle, hand, and kitten meshes show an increasing trend in running time, whereas the bunny and horse meshes exhibit a decreasing trend, with regard to increasing the workload of the GPU. If the mesh is large, the time taken decreases as the GPU workload increases. If the mesh is small, the reverse holds. This result can be attributed to the extra time taken to allocate memory in the GPU and copying the data from main memory to the GPU cache in addition to the time required for simplification.

We obtain the following metrics after one iteration and after ten iterations of the algorithm with a CPU-GPU split of 0-100: vertex count, face count, minimum angle, maximum angle, minimum area, average area, volume, vertex simplification percentage, and face simplification percentage. The metrics can be seen in Table 3.5 and Table 3.6.

Simplification using the naïve marking algorithm does not affect the volume of the test meshes significantly. It does, however, increase the average area of each element, which is to be expected. Additionally, the simplification rate is approximately 14% to 15% for one iteration of the algorithm. We will also look at the simplification rate of multiple iterations of the algorithm.

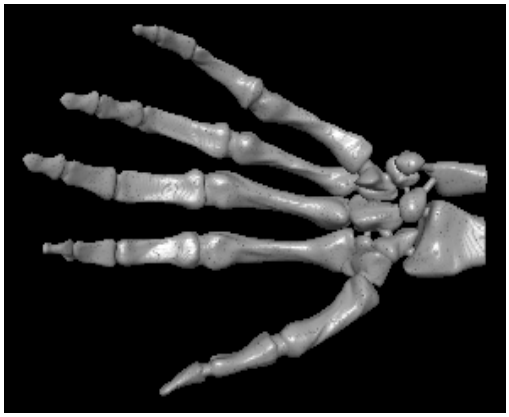
We also consider the simplification rate after performing multiple iterations of the naïve GPU algorithm on the test meshes. The simplification rates for ten iterations of the algorithm run on the armadillo, bunny, gargoyle, hand, horse, and kitten meshes can be see in Tables 3.7, 3.8, 3.9, 3.10, 3.11, and 3.12, respectively.



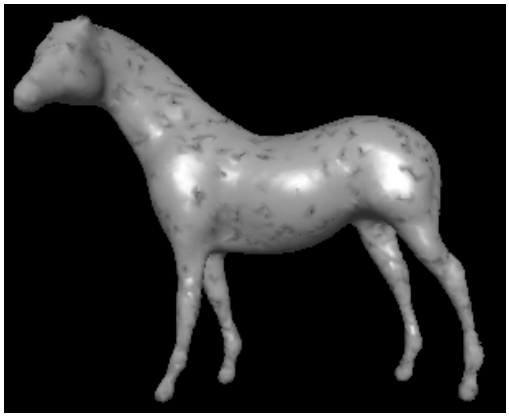
(a) Armadillo

(b) Bunny

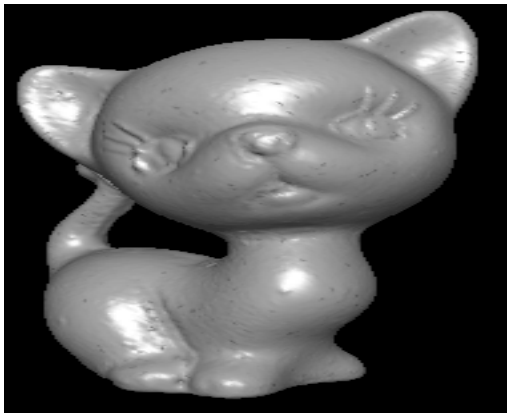
(c) Gargoyle



(d) Hand



(e) Horse



(f) Kitten

Figure 3.5: The resulting meshes after three iterations of the marking algorithm.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	172974	345944	0.0	0.0
1	147739	295456	14.6	14.6
2	129444	258746	25.2	25.2
3	114272	228141	33.9	34.0
4	101551	202291	41.3	41.5
5	90807	180363	47.5	47.9
6	81659	161547	52.8	53.3
7	73903	145486	57.3	57.9
8	67203	131575	61.1	62.0
9	61407	119454	64.5	65.5
10	56469	111617	67.4	67.7

Table 3.7: The simplification percentage of the armadillo mesh over multiple iterations. This data is shown in Figures 3.8 and 3.7.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	34834	69664	0.0	0.0
1	30102	60194	13.6	13.6
2	26103	52183	25.1	25.1
3	22772	45487	34.6	34.7
4	19976	39846	42.7	42.8
5	17603	35026	49.5	49.7
6	15588	30942	55.3	55.6
7	13870	27413	60.2	60.6
8	12396	24391	64.4	65.0
9	11093	21703	68.2	68.8
10	9979	20129	71.4	71.1

Table 3.8: The simplification percentage of the bunny mesh over multiple iterations. This data is shown in Figures 3.8 and 3.7.

The simplification rate hovers around 64% to 71% for both vertices and faces after ten iterations. After three iterations, we can see that there are visible areas where simplification occurred on the bunny and horse meshes as shown in Figure 3.5, but not so on the larger meshes. After ten iterations, the bunny, horse, and kitten meshes as shown in Figure 3.6 exhibit an extreme loss of detail. The armadillo lost some detail in the head area, and the other meshes do not show too much loss of detail. This reinforces the notion that the larger a mesh is initially, the more that it can be simplified without any visible effects.

Figure 3.7 shows the simplification percentage as a function of iteration for the vertices in each mesh. Figure 3.8 shows the simplification percentage as a function of the iteration number for the faces in each mesh. After each iteration, the increase in the simplification percentage is smaller. This suggests that as the vertex and element count of a mesh increase, the decrease in

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	863182	1726364	0.0	0.0
1	735345	1470688	14.8	14.8
2	646764	1292036	25.1	25.2
3	573431	1142601	33.6	33.8
4	512166	1042650	40.7	39.6
5	458718	952966	46.9	44.8
6	412135	872794	52.3	49.4
7	371523	801356	57.0	53.6
8	336120	737405	61.1	57.3
9	305243	680268	64.6	60.6
10	278305	629188	67.8	63.6

Table 3.9: The simplification percentage of the gargoyle mesh over multiple iterations. This data is shown in Figures 3.8 and 3.7.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	327323	654666	0.0	0.0
1	278849	557718	14.8	14.8
2	242541	484654	25.9	26.0
3	212871	423987	35.0	35.2
4	188294	387082	42.5	40.9
5	167095	354151	49.0	45.9
6	148251	289761	54.7	49.4
7	133491	267845	59.2	55.7
8	120632	247917	63.1	62.1
9	109398	229686	66.6	64.9
10	99635	213061	69.6	67.5

Table 3.10: The simplification percentage of the hand mesh over multiple iterations. This data is shown in Figures 3.8 and 3.7.

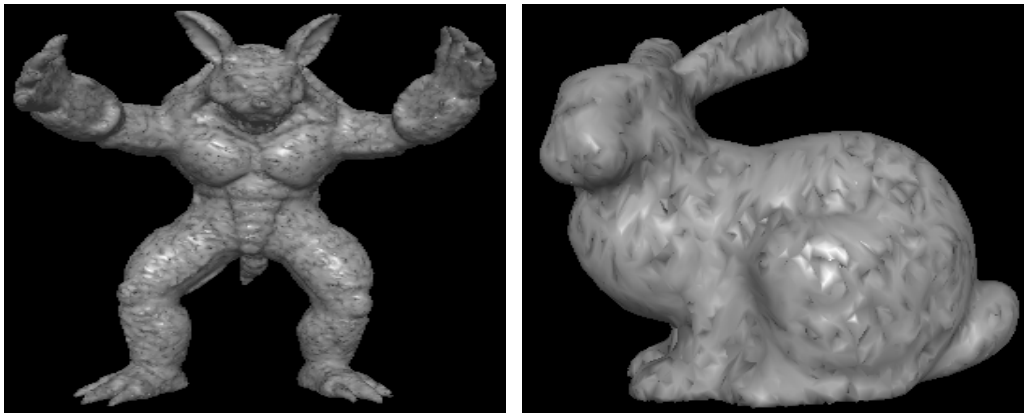
running time achieved by using the GPU simplification algorithm instead of the CPU simplification algorithm increases as well.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	15366	30728	0	0
1	13180	26598	14.2	13.4
2	11486	23363	25.3	24.0
3	10079	19766	34.4	35.7
4	9095	17950	40.8	41.6
5	8258	16393	46.3	46.7
6	7534	15037	51.0	51.1
7	6916	13850	55.0	54.9
8	6364	12767	58.6	58.5
9	5874	11788	61.8	61.6
10	5428	10927	64.7	64.4

Table 3.11: The simplification percentage of the horse mesh over multiple iterations. This data is shown in Figures 3.8 and 3.7.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	137098	274196	0.0	0.0
1	116952	233901	14.7	14.7
2	101877	203656	25.7	25.7
3	89283	178272	34.9	35.0
4	78753	162117	42.6	40.9
5	69578	147564	49.2	46.2
6	61558	122015	55.1	55.5
7	55093	112134	59.8	59.1
8	49391	103193	64.0	62.4
9	44383	95182	67.6	65.3
10	39991	87997	70.8	67.9

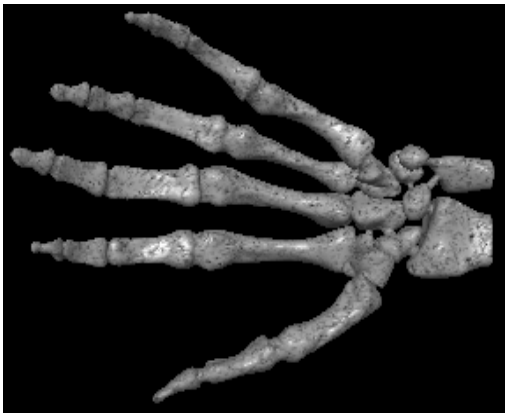
Table 3.12: The simplification percentage of the kitten mesh over multiple iterations. This data is shown in Figures 3.8 and 3.7.



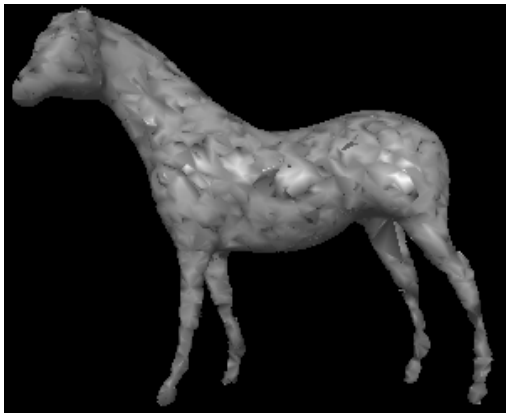
(a) Armadillo

(b) Bunny

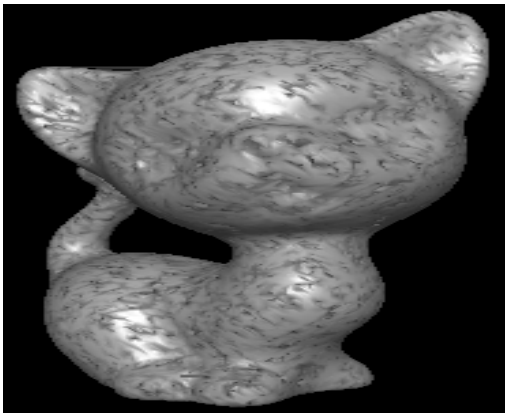
(c) Gargoyle



(d) Hand

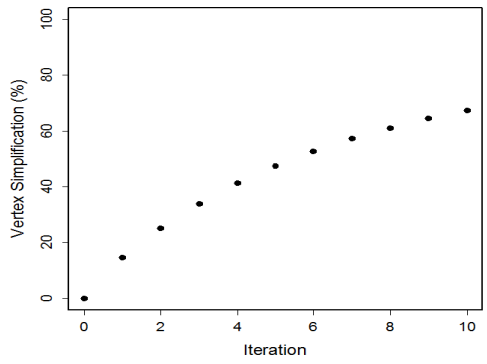


(e) Horse

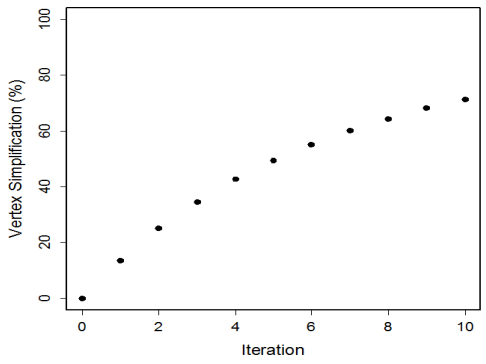


(f) Kitten

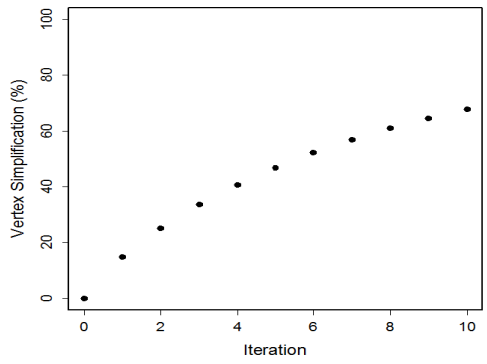
Figure 3.6: The resulting meshes after ten iterations of the marking algorithm.



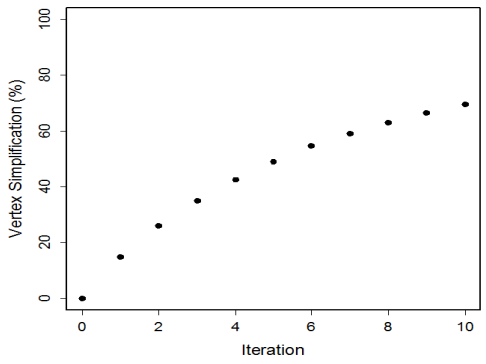
(a) Armadillo



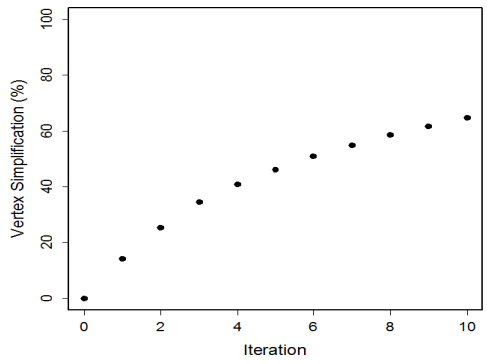
(b) Bunny



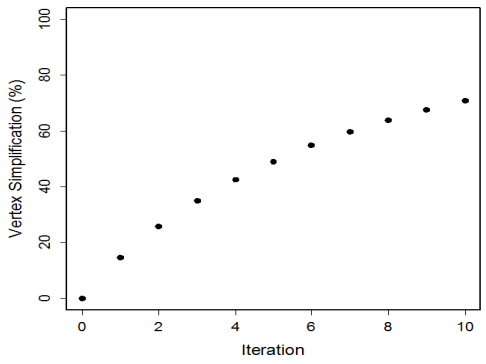
(c) Gargoyle



(d) Hand

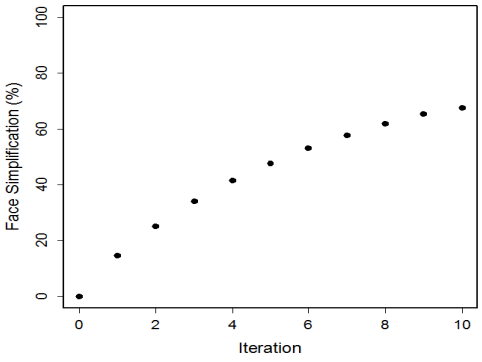


(e) Horse

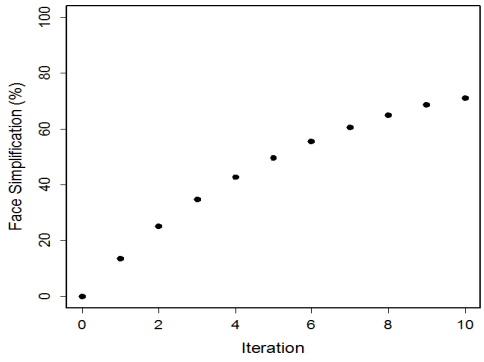


(f) Kitten

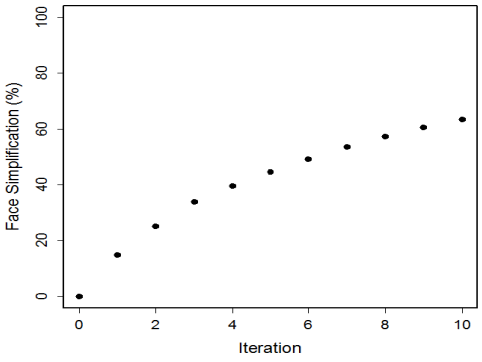
Figure 3.7: The simplification percentage of the vertices as a function of the iteration number.



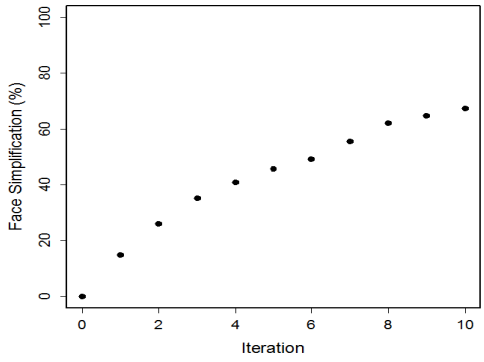
(a) Armadillo.



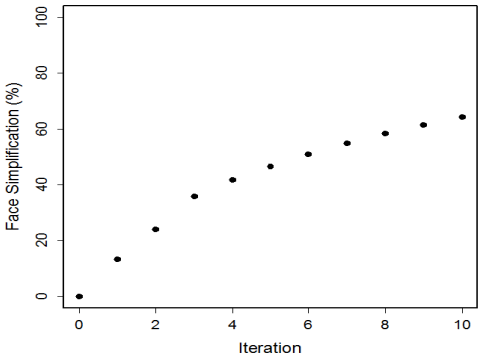
(b) Bunny.



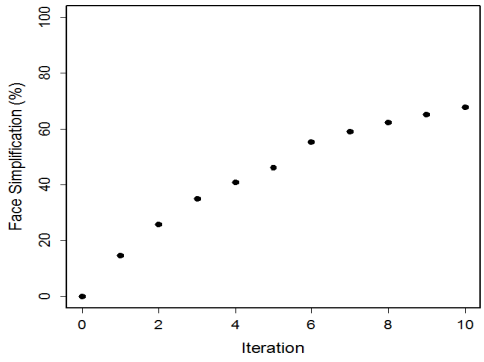
(c) Gargoyle.



(d) Hand.



(e) Horse.



(f) Kitten.

Figure 3.8: The simplification percentage of the faces as a function of the iteration number.

CPU-GPU split							
mesh	100-0	95-5	90-10	85-15	80-20	75-25	70-30
armadillo	12.4	12.4	12.3	12.3	12.2	12.1	12.1
bunny	3.51	3.51	3.51	3.50	3.50	3.49	3.49
gargoyle	68.4	67.6	66.9	66.2	65.4	64.5	63.7
hand	28.6	28.4	28.2	28.0	27.7	27.4	27.1
horse	2.12	2.12	2.12	2.13	2.13	2.13	2.14
kitten	13.3	13.2	13.1	13.1	13.0	13.0	12.9
mesh	65-35	60-40	55-45	50-50	45-55	40-60	35-65
armadillo	12.0	11.9	11.8	11.8	11.7	11.6	11.6
bunny	3.48	3.48	3.48	3.47	3.47	3.46	3.46
gargoyle	63.0	62.2	61.3	60.5	59.5	58.6	57.9
hand	26.8	26.7	26.4	26.2	25.9	25.7	25.5
horse	2.14	2.14	2.14	2.15	2.15	2.16	2.16
kitten	12.8	12.8	12.7	12.7	12.6	12.5	12.4
mesh	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	11.6	11.5	11.4	11.3	11.1	11.1	11.0
bunny	3.45	3.43	3.44	3.43	3.43	3.42	3.42
gargoyle	57.2	56.3	55.4	54.5	53.5	52.3	51.4
hand	25.3	25.0	24.7	24.4	24.1	23.9	23.6
horse	2.17	2.17	2.18	2.18	2.19	2.20	2.20
kitten	12.2	12.2	12.1	11.9	11.7	11.7	11.6

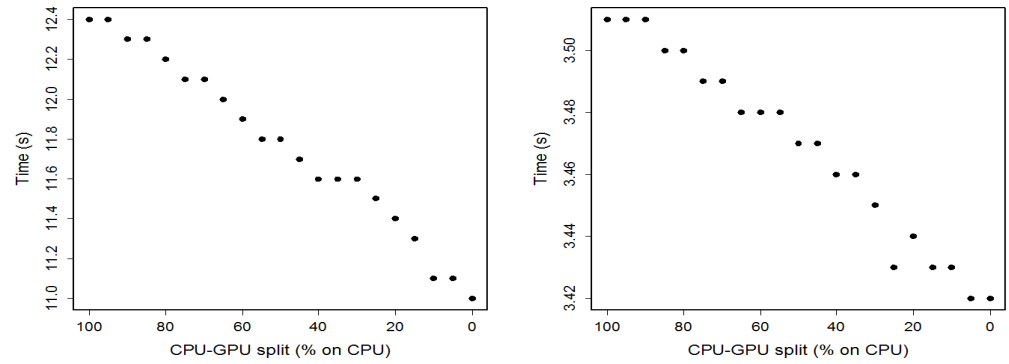
Table 3.13: The time taken to simplify for each CPU-GPU split using the inverse-reduction GPU algorithm.

3.2.2 Inverse-Reduction Algorithm

To examine the effects of splitting the mesh simplification workload between the CPU and the GPU using the inverse-reduction algorithm, we recorded the time spent during the simplification process for different CPU-GPU splits on all test cases. We tested the following CPU-GPU workload splits: 100-0, 95-5, 90-10, 85-15, 80-20, 75-25, 70-30, 65-35, 60-40, 55-45, 50-50, 45-55, 40-60, 35-65, 30-70, 25-75, 20-80, 15-85, 10-90, 5-95, and 0-100. The time taken in seconds for the tested splits can be seen in Table 3.13 and Figure 3.9. The proportion of running time spent in the GPU for the tested is shown in Table 3.14 and Figure 3.10. The GPU memory usage for the tested splits is shown in Table 3.4 and Figure 3.4.

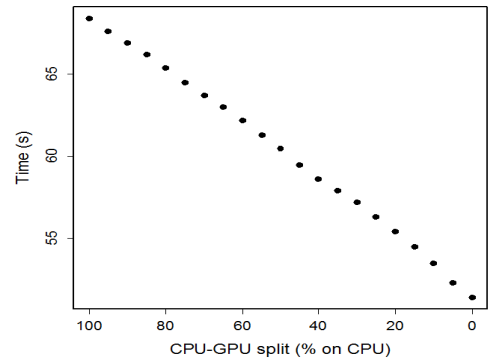
GPU % time							
mesh	100-0	95-5	90-10	85-15	80-20	75-25	70-30
armadillo	0	8.2	13.3	18.2	22.8	27.4	31.9
bunny	0	5.0	8.3	11.5	14.4	17.3	20.1
gargoyle	0	9.2	14.4	19.5	24.4	29.2	33.8
hand	0	8.3	13.5	18.4	23.1	27.8	32.5
horse	0	4.7	7.7	10.5	13.1	15.7	18.1
kitten	0	7.4	12.4	17.2	21.8	26.3	30.7
mesh	65-35	60-40	55-45	50-50	45-55	40-60	35-65
armadillo	36.2	40.5	44.7	48.7	52.6	56.4	59.8
bunny	22.7	25.2	27.6	29.8	31.9	33.9	35.7
gargoyle	38.3	42.6	46.8	51.0	55.1	58.9	62.7
hand	36.9	41.3	45.6	49.7	53.7	57.6	61.1
horse	20.4	22.5	24.6	26.5	28.2	28.8	31.4
kitten	34.8	38.9	43.0	46.8	50.7	54.3	57.8
mesh	30-70	25-75	20-80	15-85	10-90	5-95	0-100
armadillo	63.4	66.7	69.7	72.4	75.0	77.2	79.0
bunny	37.4	39.1	40.5	41.9	43.2	44.5	45.7
gargoyle	66.2	69.7	73.0	76.0	79.0	81.7	83.4
hand	64.9	68.3	71.4	74.2	76.9	79.2	81.1
horse	32.6	33.9	34.9	35.9	36.8	37.5	38.1
kitten	61.3	64.5	67.5	70.3	73.0	75.6	77.1

Table 3.14: The percentage of time spent using the GPU for different CPU-GPU splits using the inverse-reduction GPU algorithm.

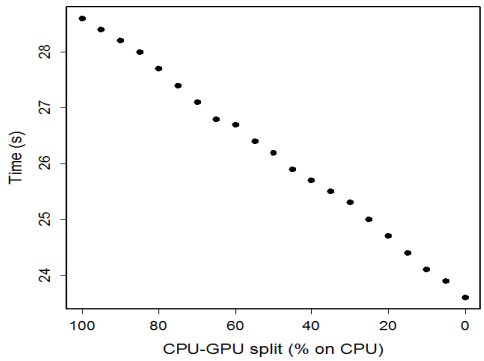


(a) Armadillo

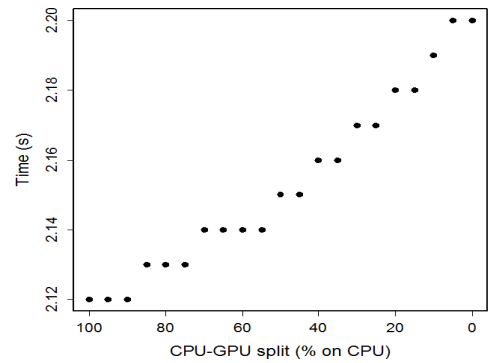
(b) Bunny



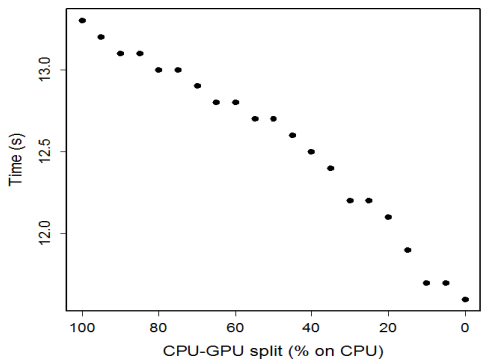
(c) Gargoyle



(d) Hand

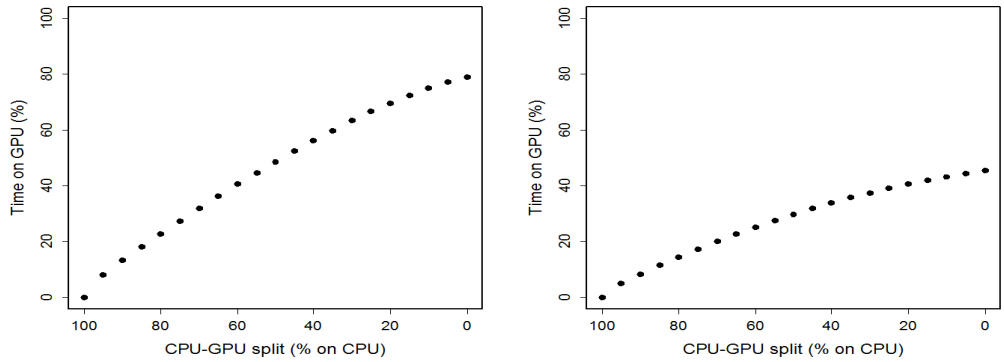


(e) Horse



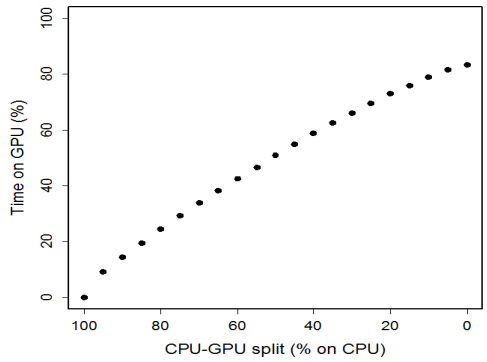
(f) Kitten

Figure 3.9: The time taken to simplify each test case for every split using the inverse-reduction GPU algorithm. As the CPU-GPU split increases, the CPU workload increases, and the GPU workload decreases.

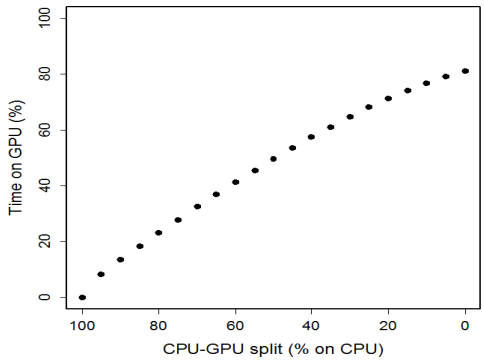


(a) Armadillo

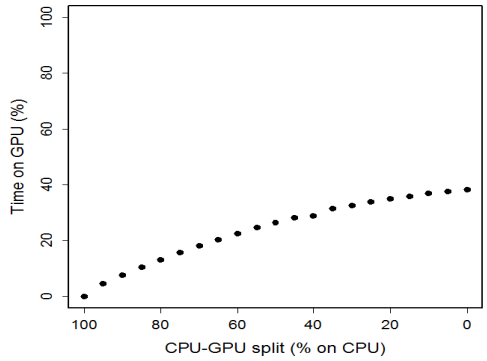
(b) Bunny



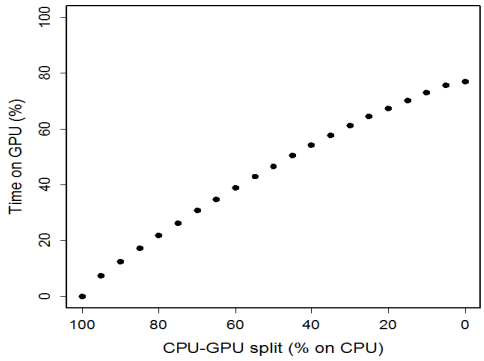
(c) Gargoyle



(d) Hand



(e) Horse



(f) Kitten

Figure 3.10: The percentage of the time spent on the GPU for every split using the inverse-reduction GPU algorithm.

mesh	# vertices	# faces	min ∠	max ∠	min area	avg area	volume
armadillo	154583	309156	5.975e-3	179.982	1.157e-09	2.050e-1	1.42706e+6
bunny	31142	62278	4.948e-1	178.476	1.020e-06	1.760e-2	7.55900e+3
gargoyle	779354	1558590	2.154e-4	179.993	1.802e-10	5.046e-2	1.64223e+6
hand	294585	589173	1.520e-2	179.558	1.158e-10	1.198e-4	1.65060e+1
horse	13916	27823	2.965e-3	179.621	1.118e-14	2.341e-6	1.58783e−3
kitten	123496	246988	4.609e-3	179.843	1.424e-08	9.830e-2	8.38419e+5

Table 3.15: Various values of the metrics for the meshes after one iteration.

mesh	# vertices	# faces	min ∠	max ∠	min area	avg area	volume
armadillo	65091	129131	2.271e-2	179.949	1.656e-07	5.190e-1	1.42659e+6
bunny	13043	25936	5.214e-2	179.805	1.000e-07	4.455e-2	7.55771e+3
gargoyle	351150	693992	3.044e-3	179.991	3.046e-08	1.211e-1	1.64145e+6
hand	132063	259374	1.449e-2	179.920	8.594e-10	2.940e-4	1.64892e+1
horse	6623	12719	1.251e-2	179.520	6.054e-11	5.698e-6	1.58656e−3
kitten	53725	106863	1.076e-2	179.912	1.476e-06	2.404e-1	8.38289e+5

Table 3.16: Various values of the metrics for the meshes after one iteration.

Again, the horse mesh shows an increasing trend, and the armadillo, bunny, gargoyle, hand, and kitten show a decreasing trend with regard to an increasing workload on the GPU. If the mesh is large, the time taken decreases as the GPU workload increases. If the mesh is small, the reverse trend holds. This result can be attributed to the extra time taken for memory allocation in the GPU and copying the data from main memory to the GPU cache, shown in Table 3.4 and Figure 3.4, in addition to the time required for simplification. Additionally, because the inverse-reduction algorithm is more efficient than the naïve marking algorithm, we see that the time taken to simplify the bunny decreases as the GPU workload increases, now decreasing anywhere between 5.55% for the bunny mesh to 11.48% for the gargoyle mesh when the GPU is given the full workload when compared to the naïve algorithm.

We collected the following metrics after one iteration and after ten iterations of the algorithm with a CPU-GPU split of 0-100: vertex count, face count, minimum angle, maximum angle, minimum area, average area, volume, vertex simplification percentage, and face simplification percentage. The values for iteration one and iteration ten can be seen in Tables 3.15 and 3.16 respectively.

Simplification using the inverse-reduction algorithm does not affect the volume of the test cases significantly. It does, however, increase the average area of each element, which is to be expected. Additionally, the simplification rate is approximately 9% to 10% after one iteration of

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	172974	345944	0.0	0.0
1	154583	309156	10.6	10.6
2	138843	277675	19.7	19.7
3	125244	250215	27.6	27.7
4	113079	225885	34.6	34.7
5	102365	204419	40.8	40.9
6	93153	185910	46.1	46.3
7	84855	169216	50.9	51.1
8	77459	154266	55.2	55.4
9	70931	141022	59.0	59.2
10	65091	129131	62.4	62.7

Table 3.17: The simplification percentage of the armadillo mesh over multiple iterations. This data is shown in Figures 3.14 and 3.13.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	34834	69664	0.0	0.0
1	31142	62278	10.6	10.6
2	27980	55954	19.7	19.7
3	25216	50383	27.6	27.7
4	22809	45567	34.5	34.6
5	20657	41261	40.7	40.8
6	18762	37463	46.1	46.2
7	17069	34064	51.0	51.1
8	15548	31013	55.4	55.5
9	14230	28349	59.1	59.3
10	13043	25936	62.6	62.8

Table 3.18: The simplification percentage of the bunny mesh over multiple iterations. This data is shown in Figures 3.14 and 3.13.

the algorithm, which is much lower than that of the naïve algorithm. There is a tradeoff between speed and rate of compression. We will also look at the simplification rate for multiple iterations of the algorithm.

We also consider the simplification rate after performing ten iterations of the inverse-reduction GPU algorithm on the test meshes. The simplification rates for performing ten iterations of the algorithm on the armadillo, bunny, gargoyle, hand, horse, and kitten meshes can be seen in Tables 3.17, 3.18, 3.19, 3.20, 3.21, and 3.22 respectively.

The simplification rate of both vertices and faces after running ten iterations of the algorithm hovers around 57% to 63%. After three iterations, we can see that there are visible areas where simplification occurred on the horse mesh as shown in Figure 3.11, but this is not so on the larger meshes. After ten iterations, the the bunny, horse, and kitten meshes exhibit an

iteration	# vertices	# faces	vertex simplification %	face simplification%
0	863182	1726364	0.0	0.0
1	779354	1558590	9.7	9.7
2	704379	1408640	18.4	18.4
3	639370	1276589	25.9	26.1
4	581774	1161369	32.6	32.7
5	531048	1059586	38.5	38.6
6	486874	970716	43.6	43.8
7	446693	889413	48.3	48.5
8	410960	816781	52.4	52.7
9	379788	753007	56.0	56.4
10	351150	693992	59.3	59.8

Table 3.19: The simplification percentage of the gargoyle mesh over multiple iterations. This data is shown in Figures 3.14 and 3.13.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	327323	654666	0.0	0.0
1	294585	589173	10.0	10.0
2	266077	532157	18.7	18.7
3	241322	481523	26.3	26.4
4	219254	437357	33.0	33.2
5	199875	398439	38.9	39.1
6	183105	364576	44.1	44.3
7	167838	333558	48.7	49.0
8	154553	306309	52.8	53.2
9	142677	281686	56.4	57.0
10	132063	259374	59.6	60.4

Table 3.20: The simplification percentage of the hand mesh over multiple iterations. This data is shown in Figures 3.14 and 3.13.

extreme loss of detail, as shown in Figure 3.12. The armadillo lost some detail in the head area, and the other meshes do not show too much loss of detail. This reinforces the notion that the larger a mesh is initially, the more that it can be simplified without any visible bad effects.

Figure 3.13 shows the simplification percentage as a function of the iteration number for the vertices in each mesh. Figure 3.14 shows the simplification percentage as a function of the iteration number for the faces in each mesh. After each iteration, the increase in the simplification percentage is decreased. This suggests that as a mesh increases in size, the decrease in running time achieved by using the GPU simplification algorithm instead of the CPU simplification algorithm increases as well.

iteration	# vertices	# faces	vertex simplification %	face simplification %
0	15366	30728	0.0	0.0
1	13916	27823	9.4	9.5
2	12643	25277	17.7	17.7
3	11546	22920	24.9	25.4
4	10559	20934	31.3	31.9
5	9696	19185	36.9	37.6
6	8932	17605	41.9	42.7
7	8251	16179	46.3	47.3
8	7640	14904	50.3	51.5
9	7104	13779	53.8	55.2
10	6623	12719	56.9	58.6

Table 3.21: The simplification percentage of the horse mesh over multiple iterations. This data is shown in Figures 3.14 and 3.13.

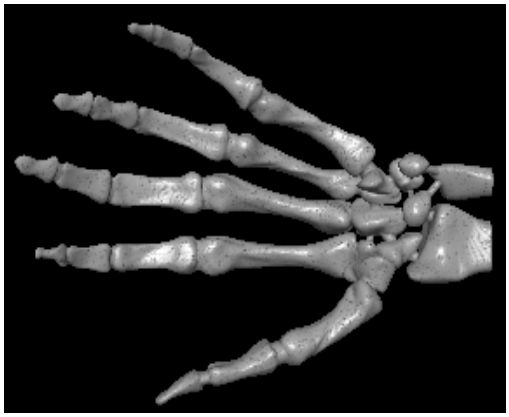
iteration	# vertices	# faces	vertex simplification %	face simplification %
0	137098	274196	0.0	0.0
1	123496	246988	9.9	9.9
2	111461	222918	18.7	18.7
3	100828	201453	26.5	26.5
4	91563	182928	33.2	33.3
5	83234	166259	39.3	39.4
6	75871	151511	44.7	44.7
7	69330	138378	49.4	49.5
8	63537	126736	53.7	53.8
9	58359	116252	57.4	57.6
10	53725	106863	60.8	61.0

Table 3.22: The simplification percentage of the kitten mesh over multiple iterations This data is shown in Figures 3.14 and 3.13..

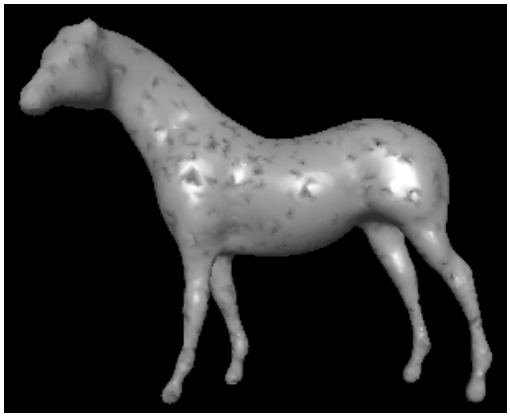
(a) Armadillo

(b) Bunny

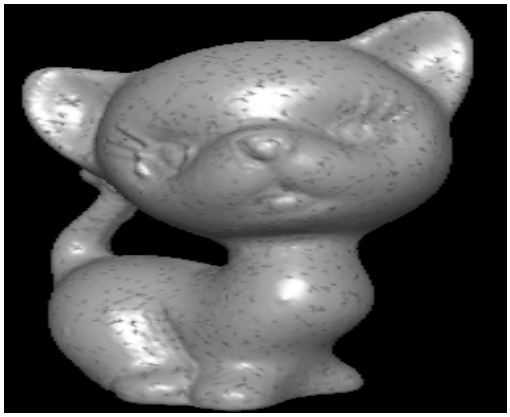
(c) Gargoyle



(d) Hand

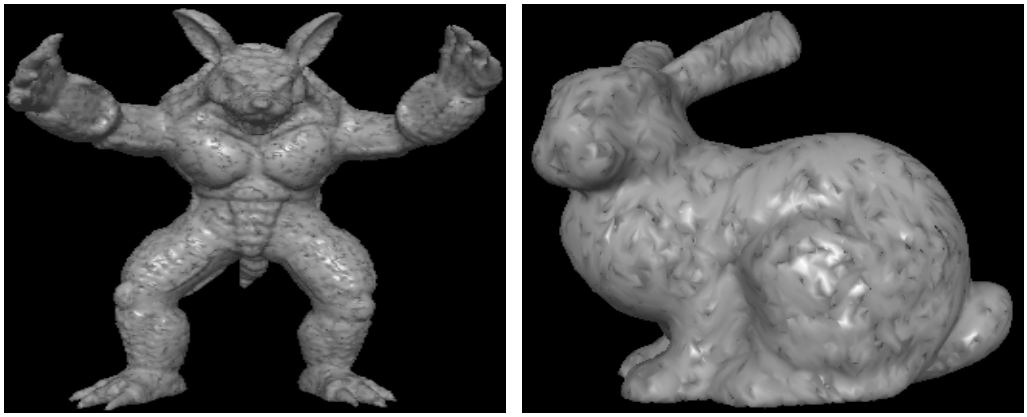


(e) Horse



(f) Kitten

Figure 3.11: The resulting meshes after three iterations of the inverse-reduction algorithm.

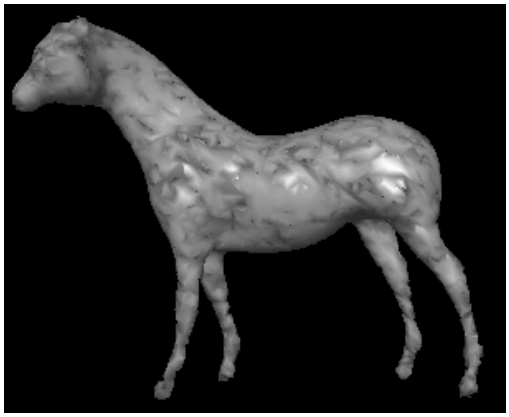


(a) Armadillo

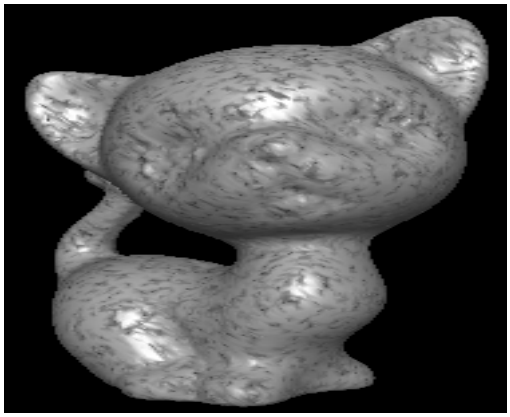
(b) Bunny

(c) Gargoyle

(d) Hand

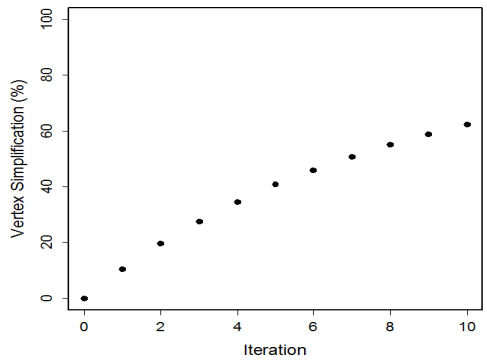


(e) Horse

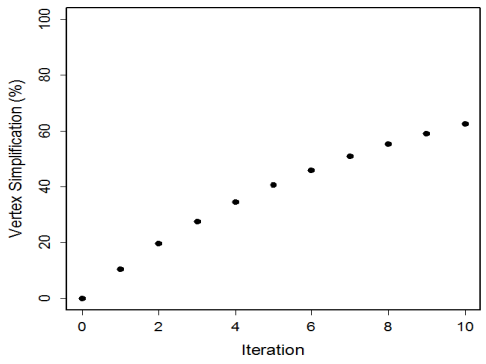


(f) Kitten

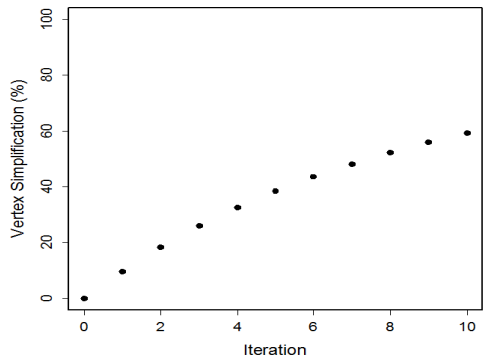
Figure 3.12: The resulting meshes after ten iterations of the inverse-reduction algorithm.



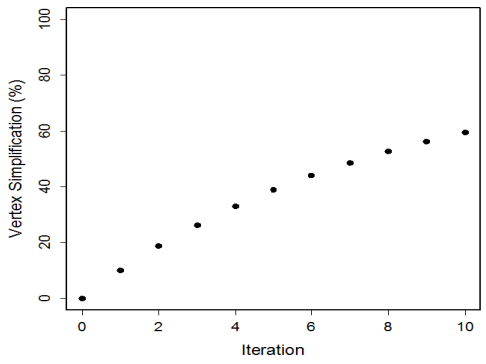
(a) Armadillo.



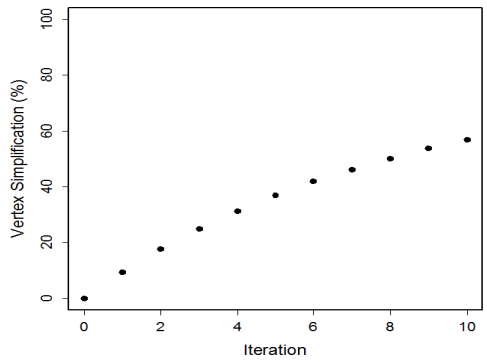
(b) Bunny.



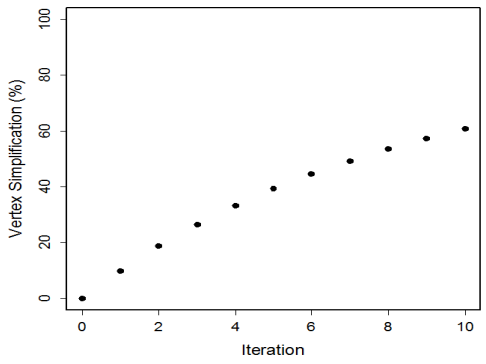
(c) Gargoyle.



(d) Hand.

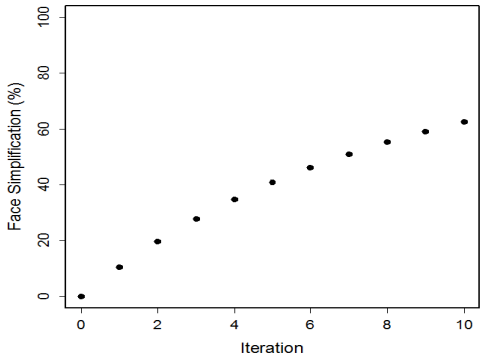


(e) Horse.

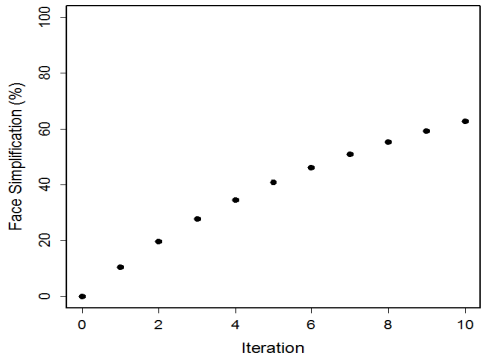


(f) Kitten.

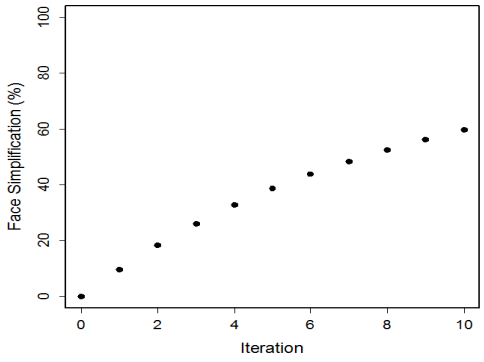
Figure 3.13: The simplification percentage of the vertices as a function of the iteration number.



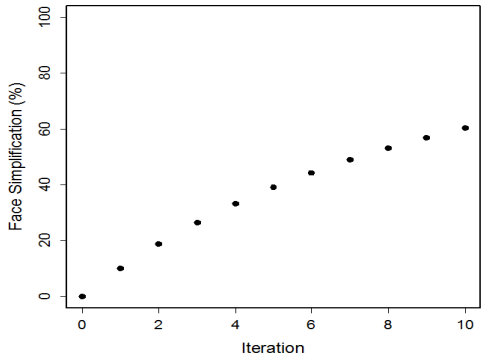
(a) Armadillo



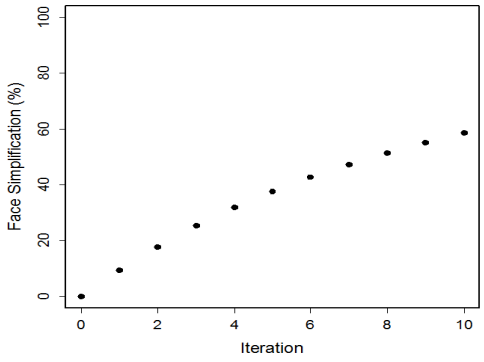
(b) Bunny



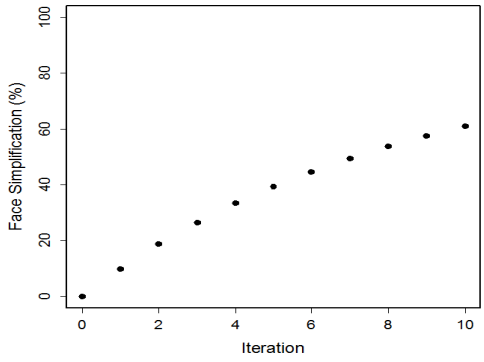
(c) Gargoyle



(d) Hand



(e) Horse



(f) Kitten

Figure 3.14: The simplification percentage of the faces as a function of the iteration number.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

In this thesis, we proposed two GPU-based surface mesh simplification algorithms: the marking algorithm, which uses multiple GPU threads to concurrently mark elements as affected, and the inverse-reduction algorithm, which attempts to perform the edge-collapse operation on twice as many edges at each step. Each of these algorithms were tested on six test meshes: an armadillo, a bunny, a gargoyle, a hand, a horse, and a kitten. The combinations of the CPU algorithm with the two GPU algorithms were novel; they leveraged the concurrency of the GPU to aid in simplifying our test meshes. These algorithms can be used in various areas of computer graphics, where there is a clear benefit in creating multiple unique meshes from a single source mesh or a simplified version of an original mesh, such as video games and computer imaging.

The simplification rate and running time of the algorithms on different meshes depended on multiple factors, including the backing GPU algorithm itself. Both algorithms used the same amount of memory on the GPU for any specific CPU-GPU split, as they both needed to keep track of the same amount of vertices and elements on the GPU. As the workload of the GPU was increased, the amount of memory used by the GPU increased as well. For both algorithms, as the size of a mesh increased, the decrease in simplification time got bigger, as the GPU workload was increased. The simplification rate of approximately 68% that the marking algorithm achieved was higher than the simplification rate of approximately 59% achieved by the inverse-reduction algorithm, over ten iterations. This slight increase in simplification rate provided by the mark-

ing algorithm was counterbalanced by an increase of anywhere between 5.55% to 11.48% in the running time when the GPU is given the full workload. This suggests that the GPU marking algorithm should be used when a larger simplification rate per iteration is needed and running time is not a limiting factor, and that the inverse-reduction GPU algorithm should be used when time is a limiting factor or a smaller simplification rate per iteration is required, such as for smaller meshes.

4.2 Future Work

We saw that specific areas of the mesh were repeatedly simplified over multiple iterations, causing the meshes to look excessively simplified in certain areas, as as can be seen by the dark spots on the meshes. An idea to prevent the simplification of the same areas would be to reorder the elements of the mesh. An element of randomness such as reordering would make it less likely that elements in the same area are chosen for simplification each iteration.

Second, it would be interesting to implement a hybrid algorithm that takes advantage of the speed of the inverse-reduction algorithm and the simplification rate of the marking algorithm. This could potentially increase the simplification rate and decrease the running time of a simplification algorithm.

Third, the effects of these algorithms on textured meshes should be studied. Our results regarding how well the algorithms perform may differ from the results obtained by running the algorithms on textured meshes.

Fourth, a mesh optimization algorithm should be included as a post-processing step. This would yield better quality meshes after each iteration, and it would minimize the visual effects of simplifying the same area multiple times over many iterations.

Finally, it may be beneficial to modify the algorithms to utilize all CPU cores. While the algorithms do show an increase in speed over the CPU-only algorithm in most cases, they may possibly show additional speed increases if all CPU cores are utilized, especially in the context of computer graphics, where running time is often a critical factor.

Bibliography

[1] Yehuda Afek, Eli Gafni, John Tromp, and Paul Vitanyi. Wait-free test-and-set. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 85–94. Springer Berlin / Heidelberg, 1992.

[2] AIM@SHAPE. Aim@shape shape repository 4.0, February 2012. `http://shapes.aimatshape.net/`.

[3] Maria-Elena Algorri and Francis Schmitt. Mesh simplification. *Computer Graphics Forum*, 15(3):77–86, 1996.

[4] C.L. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings Visualization '99*, Visualization '99, pages 307–537, 1999.

[5] Dmitry Brodsky and Jan Baekgaard Pedersen. A parallel framework for simplification of massive meshes. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 17–24, Washington, DC, USA, 2003. IEEE Computer Society.

[6] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22:37–54, 1997.

[7] NVIDIA Corporation. CUDA toolkit, January 2012. `http://developer.nvidia.com/cuda-toolkit-41`.

[8] Christopher DeCoro and Natalya Tatarchuk. Real-time mesh simplification using the gpu. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 161–166, New York, NY, USA, 2007. ACM.

[9] F. Dehne, C. Langis, and G. Roth. Mesh simplification in parallel. ICA3PP '00, pages 281–290, 2000.

[10] Christian Dick, Jens Schneider, and Rüdiger Westermann. Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum*, 28(1):67–83, 2009.

[11] Martin Franc and Václav Skala. Parallel triangular mesh decimation without sorting. In *Proceedings of the 17th Spring Conference on Computer graphics*, SCCG '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[12] Paul Heckbert and Michael Garl. Multiresolution modeling for fast rendering. In *Proceedings of Graphics Interface*, pages 43–50, 1994.

[13] Jon Hjelmervik and Jean-Claude Leon. GPU-accelerated shape simplification for mechanical-based applications. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, SMI '07, pages 91–102, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[15] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. *SIGGRAPH Comput. Graph.*, 26(2):71–78, July 1992.

[16] 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*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[17] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In Greg Turk, Jarke J. van Wijk, and Robert J. Moorhead II, editors, *IEEE Visualization*, pages 465–472. IEEE Computer Society, 2003.

[18] Andrew E. Johnson and Martial Herbert. Control of polygonal mesh resolution for 3-d computer vision. *Graph. Models Image Process.*, 60(4):261–285, July 1998.

[19] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 75–ff., New York, NY, USA, 1997. ACM.

[20] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[21] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[22] Georgia Institute of Technology. Large geometric models archive, February 2012. `http://www.cc.gatech.edu/projects/large_models/`.

[23] Zhigeng Pan, Kun Zhou, and Jiaoying Shi. A new mesh simplification algorithm based on triangle collapses. *J. Comput. Sci. Technol.*, 16(1):57–63, 2001.

[24] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70, July 1992.

[25] Sudipta N. Sinha, Jan michael Frahm, Marc Pollefeys, and Yakup Genc. Gpu-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[26] Stanford University. The stanford 3D scanning repository, January 2012. `http://graphics.stanford.edu/data/3Dscanrep/`.