A Flexible C++ Library for Wavelet Transforms of 3-D Polygon Meshes

by

Shengyang Wei

B.A.Sc., Huazhong University of Science and Technology, 2013

A Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

A Flexible C++ Library for Wavelet Transforms of 3-D Polygon Meshes

by

Shengyang Wei

B.A.Sc., Huazhong University of Science and Technology, 2013

Supervisory Committee

Dr. Michael Adams, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Pan Agathoklis, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

The lifted wavelet transforms of 3-D polygon meshes are introduced, and the details of Loop and Butterfly wavelet transforms are studied. Then, a library that implements a framework for computing lifted wavelet transforms of polygon meshes is presented. To compute Loop and Butterfly wavelet transforms, users can employ the built-in functionality of the library. In addition, users can also define custom wavelet transforms via a secondary application programming interface provided by the library. Some application programs implemented with this library are also provided for demonstration purposes, including application that perform wavelet-based polygon mesh simplification and denoising. Finally, the run-time performance of the library are measured. Our library is shown to perform lifted wavelet transforms, except the subdivision detection step in linear time with respect to the number of vertices. The main bottleneck is the subdivision detection step, since it includes sorting, which has a time complexity greater than linear.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

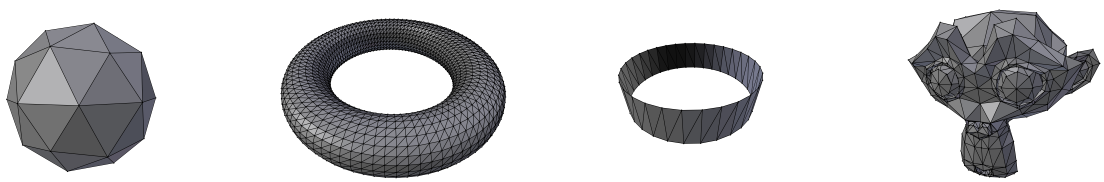
Introduction

1.1 Wavelet Transform of 3-D Triangle Mesh

Three-dimensional (3-D) modeling and animation play essential roles in various industries related to video games, virtual reality, films, and medicine. In 3-D modeling, a polygon mesh is one of the most popular representations of 3-D objects, where polygons are joined together to represent or approximate the surface of an object. The triangle mesh where all the polygons are triangles is the preferred representation in most modern graphical applications. Figure 1.1 includes some examples to illustrate triangle meshes.

Since triangles are planar, approximating a smooth surface with triangle meshes typically requires a large number of faces. To model real-world objects realistically, meshes could easily consist of millions of triangles. Such meshes would consume excessive resources to render, store, and transmit. Subdivision and multiresolution analysis are practical solutions to this kind of problem. Subdivision is capable of characterizing a smooth surface by a very simple mesh (i.e., with relatively few faces). In other words, a smooth surface is calculated on demand in a subdivision process. Subdivision provides a set of refinement rules such that repeatedly applying these rules to a coarse mesh results in an increasingly smooth mesh. An example of a subdivision process is shown in Figure 1.2. Figure 1.2(a) shows a sphere, and Figures 1.2(b), (c), and (d) show the sphere after applying subdivision one, two, and an infinite number of times, respectively.

An alternative approach, multiresolution analysis, handles the problem in a different yet similar way. By using a multiresolution analysis, a complicated mesh can be represented with multiple levels of detail, which has a flexible resource cost. A multiresolution representation of a complicated mesh consists of a coarse approximation mesh plus finer detail in increasing levels of resolution. Then, the mesh at a desirable resolution can be represented by the approximation incorporating the details that are at and below the desired resolution. Typically, a multiresolution analysis is calculated using a wavelet transform (WT). That is, for a complicated mesh, a WT iteratively computes an approximation at a lower resolution level and encodes the difference between meshes at two successive resolutions (i.e., the details) into a set of wavelet coefficients. Then, to obtain the mesh at a desirable level of resolution,



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

Figure 1.1: Examples of triangle meshes. (a) A sphere, (b) a torus, (c) a strip, and (d) a monkey.

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

Figure 1.2: Example of subdivision. (a) Coarse mesh. (b), (c), (d) are generated meshes after repeating the refinement process once, twice, and ad infinitum.

a WT repetitively incorporate a set of wavelet coefficients to the approximation from the lowest resolution level to the desired one. In addition, a WT provides two basic operations. An operation that obtains a coarse mesh along with a set of wavelet coefficients is called the forward WT (FWT) (or wavelet analysis). An operation that combines a set of wavelet coefficients to a coarse mesh is called the inverse WT (IWT) (or wavelet synthesis). An example of the result obtained from wavelet analysis is shown in Figure 1.3. A refined mesh is shown in Figure 1.3(a), and the resulting mesh incorporating wavelet coefficients after one, two, and three levels of wavelet analysis is shown in Figures 1.3(b), (c), and (d), each of which shows the multiresolution representation of the refined mesh in two, three, and four resolution levels, respectively. In these figures, the approximation mesh is at resolution level 0, and sets of wavelet coefficients that capture details at resolution level 1, 2, and 3 are denoted by W_1, W_2, and W_3, respectively. Since wavelet synthesis is an inverse of wavelet analysis, Figure 1.3 also illustrates an example of wavelet synthesis. Applying one, two, and three levels of wavelet synthesis to the mesh and wavelet coefficients shown in Figures 1.3(b), (c), and (d), respectively, we can obtain the refined mesh shown in Figure 1.3(a).

Obviously, we can manipulate the wavelet coefficients calculated from wavelet analysis to change the mesh obtained from wavelet synthesis. For this reason, WTs are useful in many applications. A typical example is multiresolution editing. Instead of editing a mesh at full resolution, users can adjust the mesh under different resolutions by changing the corresponding sets of wavelet coefficients. Moreover, WTs can be used for mesh compression or simplification. After obtaining wavelet coefficients from wavelet analysis, a mesh can be reconstructed within a desirable error tolerance by using an appropriate portion of the wavelet coefficients. Figure 1.4 presents an example for wavelet compression. Figure 1.4(a) shows a bunny, and Figure 1.4(b) shows an approximation of the bunny after applying four levels of wavelet analysis. After four levels of wavelet synthesis, a reconstructed bunny by using the wavelet coefficients whose magnitudes are in the top 5% is shown in Figure 1.4(c).

Although not immediately obvious, the WT has a close relationship to subdivision. Subdivision provides the mathematical foundation for designing WTs of polygon meshes. Specifically, the subdivision connectivity which will be discussed in Chapter 2 is a prerequisite for WTs. Indeed most WT algorithms are derived from corresponding subdivision schemes.

1.2 Historical Perspective

Lounsbery [16] and Lounsbery et al. [17] innovatively extended the multiresolution analysis theory [18] to polygon meshes with subdivision connectivity, which serves as the mathematical foundation for the WT designs considered herein. In addition to the multiresolution analysis, these papers present an implementation of WTs for polygon meshes. The implementation utilizes a filterbank algorithm, where the coarse approximation is calculated by an averaging operation called lowpass filtering, and then the wavelet coefficients are calculated by a differencing operation called highpass filtering. The lowpass and highpass filters are designed to yield invertible transforms. Inspired by Lounsbery's work and combining with the lifting scheme [25], Schröder and Sweldens [23] proposed the second generation WT on polygon meshes, known as the lifted WT. The lifting scheme was initially developed for the custom design of classical wavelets. The basic idea behind lifting scheme is to start from trivial operations and construct new more

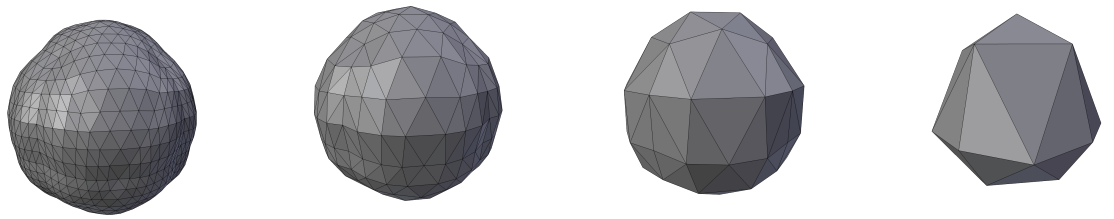


Figure 1.3: Example of WT. (a) A refined mesh. Resulting mesh and wavelet coefficients after applying (b) one, (c) two, (d) three, and (e) four levels of the FWT, where W_1, W_2, and W_3 represent wavelet coefficients at resolution level 1, 2, and 3, respectively.

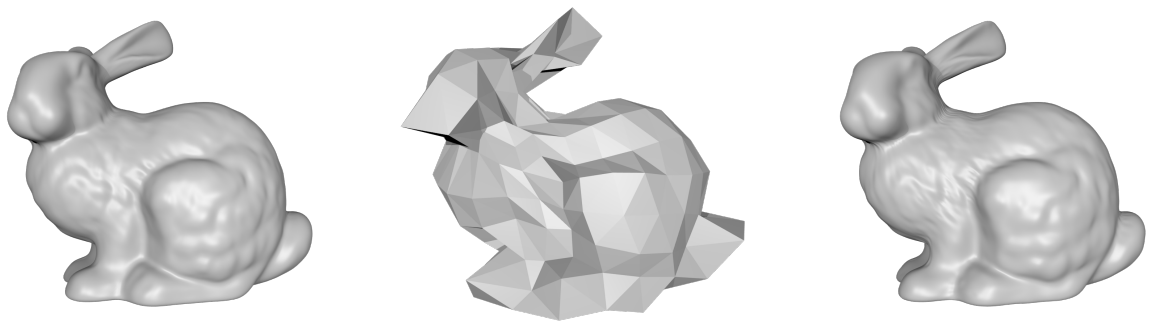


Figure 1.4: Example of wavelet compression.(a) A bunny. (b) A coarse approximation after applying four levels of the FWT. (c) Reconstructed mesh by using the wavelet coefficients whose magnitudes are in the top 5%.

performant ones by adjusting these operations. In case of polygon meshes, a realization of a lifted WT is obtained by factorizing operations in subdivision into a combination of local calculations and then lifting these operations to achieve the desired properties. Since all the calculations are local, the lifting scheme can speed up the WTs. Along with the establishment of the generalized lifted WT, Schröder and Sweldens [23] have also included a Butterfly WT, where they use a Butterfly subdivision template to customize the wavelet construction. Powered by the lifting scheme, Bertram [3] introduced the Loop WT, which is derived from Loop subdivision [15]. Wang et al. [30] considered the mesh-boundary case of the Loop WT in the presentation of a lifted wavelet construction for triangle/quad mesh by using Loop/Catmull-Clark [6] templates. Also, Wang et al. [29] proposed the lifted $\sqrt{3}$ WT for triangle meshes based on $\sqrt{3}$ subdivision [14].

1.3 Overview and Organization of Report

In this report, we mainly focus on the implementation details of the lifted Loop and Butterfly WTs for triangle meshes. Since designing a lifted WT is quite complicated and beyond the scope of this report, the design of WTs is not considered herein. Because subdivision connectivity is a prerequisite for WTs, we include a discussion of a subdivision connectivity detection algorithm proposed by Taubin [26]. To help users of our library, detailed documentation, demonstration applications, and performance profiling of our library are included in this report as well. The remainder of this report is organized as follows.

Chapter 2 introduces the essential background information needed to understand the material presented herein. First, polygon and triangle meshes, as well as related concepts, are introduced. This is then followed by a discussion of subdivision. After that, we introduce multiresolution mesh representations and present the algorithms for the lifted Loop and Butterfly WTs. Lastly, a general description of the subdivision connectivity detection algorithm is given.

Chapter 3 presents the software developed by the author. This chapter begins with an overview of the software which is then followed by instructions on how to build and install the software. Then, we introduce the high-level application programming interfaces (APIs) of the library that can be used to compute the Loop and Butterfly WTs. After that, the low-level APIs used to define a custom WT is documented. The command line interfaces of the demonstration programs are provided at the end.

Chapter 4 evaluates the performance of this library. To begin, some examples produced by the demonstration programs are presented. This is then followed by an overview of the test datasets. Next, we profile the code in order to determine the time consumed by each stage in the lifted WT. In addition, the memory consumption of this library is analyzed.

Chapter 5 concludes this report with some closing remarks and some suggestions for future work.

Chapter 2

Background

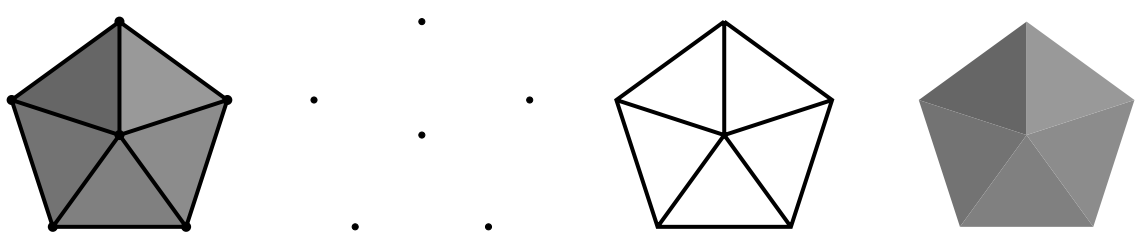
2.1 Introduction

This chapter provides the necessary background information for readers to understand the work presented by this report. We start by introducing the concept of a polygon mesh and some related concepts. This is then followed by a description of subdivision including the Loop subdivision. After that, the lifting scheme, which is used for implementing lifted a WT, is discussed. Then, we illustrate a subdivision connectivity detection algorithm which is a preprocessing step in the WT implementation. Finally, algorithms for the Loop and Butterfly WTs are presented.

2.2 Polygon Meshes

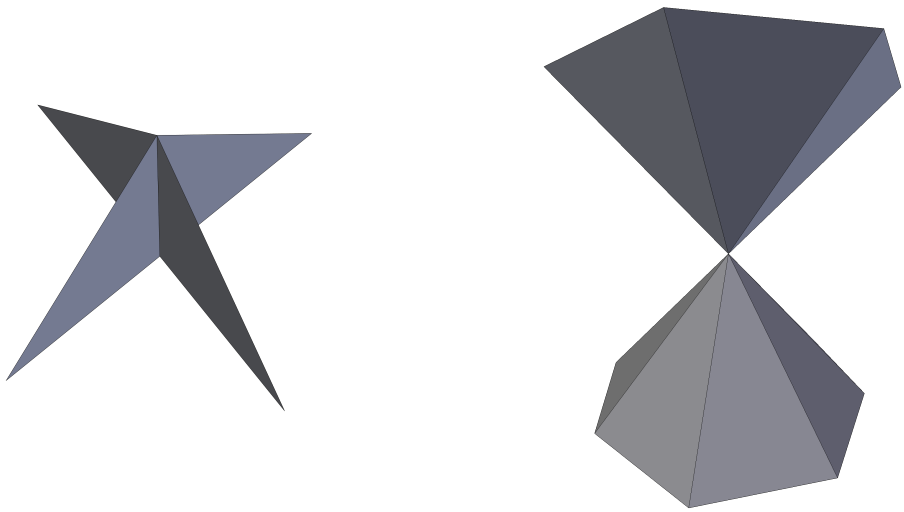
A polygon mesh is a collection of vertices, edges, and faces, and their adjacency relationships. A vertex is a point, and an edge is a line segment that has two vertices as endpoints. A face is a polygon, which is a 2-D shape enclosed by a finite number of edges. In 3-D modelling, faces are joined together by common edges and vertices to approximate the surface of an object. A triangle mesh is a polygon mesh whose faces are all triangles. In Figure 2.1, we provide an example of a triangle mesh. Figure 2.1(a) shows a simple triangle mesh. Figures 2.1(b), (c), and (d) show the vertices, edges, and faces of the mesh, respectively. In this report, we are interested primarily in triangle meshes.

A polygon mesh consists of two types of information: 1) geometric information and 2) topologic information. The geometry of a mesh is essentially the positions (i.e., the x, y, and z coordinates) of its vertices. The topology of a mesh is the adjacency relationships between its vertices, edges, and faces. Next, we introduce some concepts about polygon meshes that will be used in the subsequent sections. Two faces are adjacent if they are incident on the same edge. Two edges are adjacent if they are incident on the same vertex. Two vertices are adjacent if they are connected by the same edge. The **1-ring neighbours** of a vertex v are the vertices that are adjacent to the vertex v. The **valence** of a vertex is the number of its 1-ring neighbours, which is also the number of its incident edges. The edges and vertices of a polygon mesh can be categorized as **interior**, **boundary**, or **singular** [26]. An interior edge is an edge with exactly



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

Figure 2.1: Example of a triangle mesh and its elements. (a) A triangle mesh. (b) vertices, (c) edges, (d) faces of the mesh.



(a) (b)

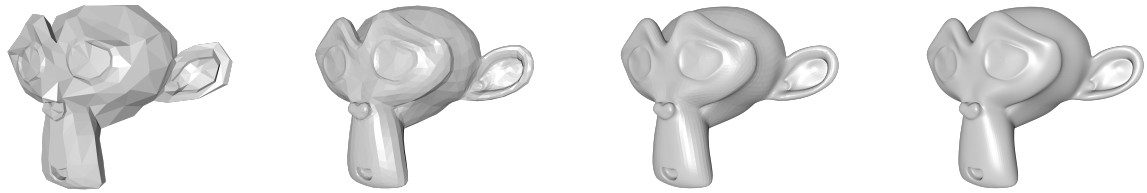
Figure 2.2: Example of non-manifolds. (a) Four triangles joining by a common edge. (b) Two cones intersecting at a common vertex.

two incident faces, and a boundary edge is an edge with exactly one incident face, and a singular edge is an edge with more than two incident faces. A **closed** polygon mesh has interior edges only. A vertex is said to be interior if all of its incident edges are interior, while a vertex is said to be boundary if exactly two of its incident edges are boundary and other incident edges are interior. For either an interior vertex or a boundary vertex, the edges that connect its 1-ring neighbours must form exactly one loop or open path. Otherwise, it is singular. A polygon mesh is said to be a manifold if none of its edges and vertices is singular. Since non-manifolds are special, in order to illustrate the case clearly, we provide some examples of non-manifolds in Figure 2.2. Because the mesh presented in Figure 2.2(a) has a singular edge and the mesh presented in Figure 2.2(b) has a singular vertex, they are non-manifolds. In this report, we only consider meshes that are manifold. In addition, for a triangle mesh, interior vertices with a valence of six or boundary vertices with a valence of four are said to be **regular**. Vertices with other valences are said to be **extraordinary**.

2.3 Subdivision

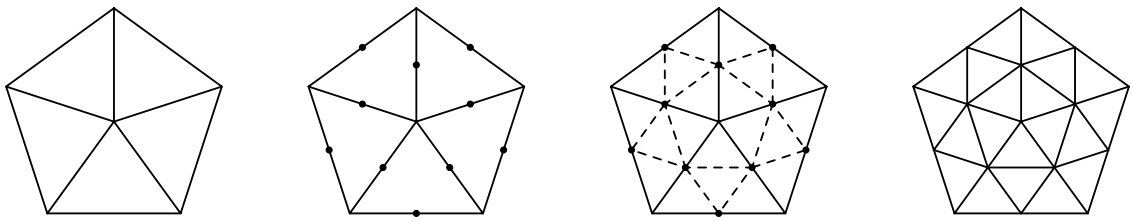
A surface represented by a polygon mesh is approximated by planar faces. The denser the faces, the better is the approximation. To represent an object accurately, a mesh with a potentially very large number of polygon faces would be required, which would consume excessive amounts of memory. In this section, we introduce a solution to this problem, namely, subdivision. Subdivision can characterize a complicated mesh by a much simpler one. That is, a refined mesh can be obtained by algorithmically inserting vertices, edges, and faces into a coarse mesh. The coarse mesh that serves as the starting point of subdivision is called the **control mesh**. Repeating subdivision iteratively refines the mesh, and as a result, the number of vertices, edges, and faces in the mesh is increased. A smooth surface obtained from repeating subdivision ad infinitum is called the **limit surface**. In practice, a subdivision process is defined so as to yield a smooth limit surface. In Figure 2.3, we present an example of subdivision for a monkey head. Figure 2.3(a) shows a coarse mesh that roughly models a monkey head. The refined meshes, after repeating subdivision one, two, and three times are shown in Figures 2.3 (b), (c), and (d), respectively.

Since inserting vertices leads to changes in both topology and geometry of a mesh, a subdivision scheme requires specific topologic and geometric refinement rules to control the changes in topology and geometry. Generally, one round of subdivision is completed by applying topologic refinement rules followed by geometric refinement rules to a mesh. In what follows, we explain the topologic and geometric refinement rules in detail.



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

Figure 2.3: Example of subdivision for a monkey head. (a) A coarse mesh. Refined mesh after applying (b) one, (c) two, and (d) three levels of subdivision.



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

Figure 2.4: Primal triangle quadrisection. (a) The topology of a mesh. (b) The topology of the mesh after inserting edge vertices. (c) Splitting faces by connecting edge vertices. (d) The final topology of the mesh after applying primal triangle quadrisection.

2.3.1 Topologic Refinement Rules

A topologic refinement rule determines how the connectivity of a mesh is to be modified to insert new vertices. The geometry is not modified by such a rule, however. Depending on whether vertices or faces are split, a topologic refinement rule is either dual or primal, respectively. For triangle meshes, one of the most common topologic refinement rules is primal triangle quadrisection (PTQ). The name PTQ suggests the type of modifications made to a mesh's topology, namely, each face is split in four in order to insert new vertices. In Figure 2.4, we illustrate the process of applying PTQ to a mesh. Figure 2.4(a) shows the topology of a mesh with a boundary. First, PTQ splits each edge in two by inserting a new vertex called an edge vertex, as shown in Figure 2.4(b). The new vertices are placed arbitrarily at the midpoint of each edge for illustration only, since a topologic refinement rule does not define vertex positions. Then, PTQ splits each face in four by connecting the three edge vertices for each face, as shown in Figure 2.4(c). Finally, Figure 2.4(d) shows the topology obtained after applying PTQ to the mesh.

Observe that PTQ always introduces interior vertices with a valence of six and boundary vertices with a valence of four, and maintains the regularity of old vertices. Therefore, the new vertices of a mesh obtained from PTQ are all regular. Furthermore, a new vertex has exactly two old vertices as 1-ring neighbours, and the 1-ring neighbours of an old vertex are all new vertices. That is, the regular new vertices isolate the old vertices in the refined mesh. Such a mesh is said to have **subdivision connectivity**. Subdivision connectivity is not limited to PTQ. If a refined mesh can be obtained from a simpler mesh by applying several levels of topologic refinement rules, regardless of geometry, we say the refined mesh has subdivision connectivity.

2.3.2 Geometric Refinement Rules

Since the positions of the vertices are not specified by topologic refinement, another rule known as a geometric refinement rule is required to determine the positions of old and new vertices after topologic refinement. The topology of the mesh is not altered in the geometric refinement. To update the geometry of a mesh, geometric refinement rule is applied to each vertex whose position is to be computed. The vertex whose position is being computed is called a

target vertex. The position of a target vertex is computed as a weighted sum of the old nearby vertices only. In other words, each vertex of a mesh obtained from subdivision is a linear combination of the vertices of the original mesh. That is how a simple control mesh can characterize a more complicated one. The control mesh can yield a sequence of progressively refined meshes via several levels of subdivision.

A geometric refinement rule is usually defined by a mask which can be viewed as a filter. A mask specifies the vertices and corresponding weights used to calculate the position of a target vertex. The vertices that participate in this computation are called **support vertices**. Each support vertex is assigned an individual weight in the calculation. Then, the mask slides over the mesh and uses the support vertices to modify each target vertex. Through this process, the new geometry of the mesh is obtained. For instance, Figure 2.5 shows the four masks used in the Loop subdivision. In Figure 2.5, a solid or empty dot refers to a vertex, and a solid or dashed line refers to an edge. Each of the masks describes the topologic relations between the support vertices and the target vertex. The target vertex is denoted by a solid dot, and the support vertices are denoted by empty dots with weights. The other empty dots without weights represent vertices that are used as references in locating the support vertices correctly. Typically, the reference vertices and support vertices have different types (i.e., new and old) so that the support vertices of a target vertex can be fetched from the mesh without ambiguity. In one geometric updating step of the Loop subdivision, the center of a mask is aligned first to each target vertex to determine its support vertices. Then, the new position of the vertex is computed from the support vertices and their respective weights. To further illustrate subdivision, we provide the details of the Loop subdivision in the following sections.

2.3.3 Loop Subdivision

The Loop subdivision, originally proposed in [15], is defined for triangle meshes with or without boundaries. The topologic refinement rule employed is PTQ. To perform Loop subdivision, PTQ is first applied to the mesh in order to introduce new vertices. After topologic refinement, the vertices of the mesh fall into two types: edge vertices (i.e., new vertices) and old vertices, and both of them are modified in geometric refinement. In addition, the Loop subdivision has different treatment to vertices depending on whether they are on the boundary. Thus, four types of vertices need to be handled in geometric refinement: 1) interior edge vertices, 2) old interior vertices, 3) boundary edge vertices, and 4) old boundary vertices. Loop subdivision provides four masks, as shown in Figure 2.5 to handle these four cases. We use a solid dot to represent the target vertex and empty dots to represent the support vertices, in each of the masks shown in Figure 2.5. To avoid confusion, we use a dashed line to represent an edge that connects two edge vertices and use a solid line to represent an edge that connects an old vertex and an edge vertex. Then, the new positions of the four types of vertices are computed in turn as follows:

1. **Interior edge vertex**. First, we consider computing the position of an interior edge vertex. Let v_e be an interior edge vertex, and v_1, v_2, v_3, and v_4 be the support vertices. The support vertices' weights and their topologic relation to v_e are specified by the mask shown in Figure 2.5(a). Then, v_e is chosen as

$$v_e = \frac{3}{8}(v_1 + v_2) + \frac{1}{8}(v_3 + v_4).$$

2. **Old interior vertex**. Next, we consider computing the position of an old interior vertex. Let v be an old interior vertex, and $\{v_i\}_{i=1}^n$ be the support vertices. The support vertices' weights and their topologic relation to v are specified by the mask shown in Figure 2.5(b). The original position of v also participates in the computation. Then, the new position v' of v is given by

$$v' = (1 - n\beta_n)v + \beta_n \sum_{i=1}^{n} v_i,$$

where

$$\beta_n = \frac{1}{n}\left[\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2\right]$$

and n is the valence of v.

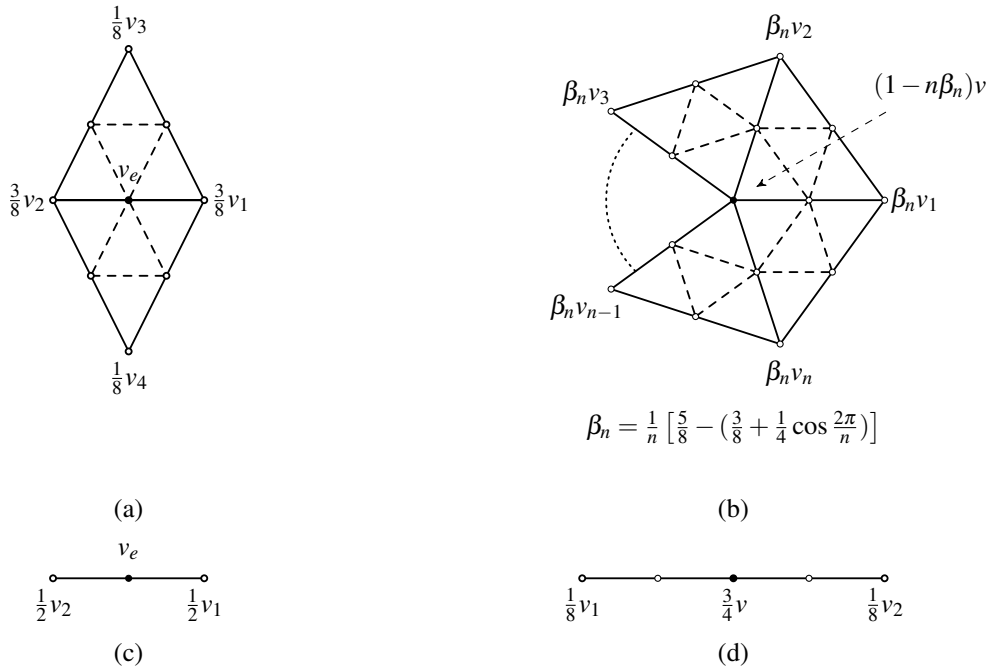


Figure 2.5: The masks used in geometric refinement of Loop subdivision. The mask used to compute (a) an interior edge vertex, (b) an old interior vertex, (c) a boundary edge vertex, and (d) an old boundary vertex.

3. **Boundary edge vertex**. Next, we consider computing the position of a boundary edge vertex. Let v_e be an boundary edge vertex, and v_1 and v_2 be the support vertices. The support vertices' weights and their topologic relation to v_e are specified by the mask shown in Figure 2.5(c). Note that v_1 and v_2 are also on the boundary of the mesh. Then, v_e is chosen as

$$v_e = \frac{1}{2}(v_1 + v_2).$$

4. **Old boundary vertex**. Finally, we consider computing the position of an old boundary vertex. Let v be an old boundary vertex, and v_1 and v_2 be the support vertices. The support vertices' weights and their topologic relation to v are specified by the mask shown in Figure 2.5(d). Note that all the vertices in the mask are on the boundary and v_1 and v_2 are old vertices. Then, the new position v' of v is given by

$$v' = \frac{3}{4}v + \frac{1}{2}(v_1 + v_2).$$

2.4 Multiresolution Analysis and WTs

Having described subdivision, we now introduce the concept of a multiresolution analysis, which has a close relationship to subdivision. With a multiresolution analysis a mesh is represented at several levels of resolution. In particular, we have mesh approximation corresponding to the lowest level of resolution and additional detail for each higher resolution. When more detail is required, the representation at a higher or full resolution can be used; when some detail can be ignored, the representation at a lower resolution can be used. Multiresolution analysis guarantees that the representation of the mesh at any intermediate resolution can be computed from the corresponding intermediate mesh plus detail information. This allows a mesh with a desired level of detail to be computed on demand. In Figure 2.6, we provide an example of multiresolution analysis of a mesh. Figure 2.6(a) shows the original mesh, which is a bunny at full resolution. Figure 2.6(b) shows the multiresolution representation of the bunny, where the mesh is decomposed using four resolution levels. The mesh in Figure 2.6(b) is at resolution level 0. Let W_1, W_2, and W_3 denote details at level 1, 2, and 3, respectively. Then, incorporating W_1 into the level 0 mesh yields the bunny at resolution level-1

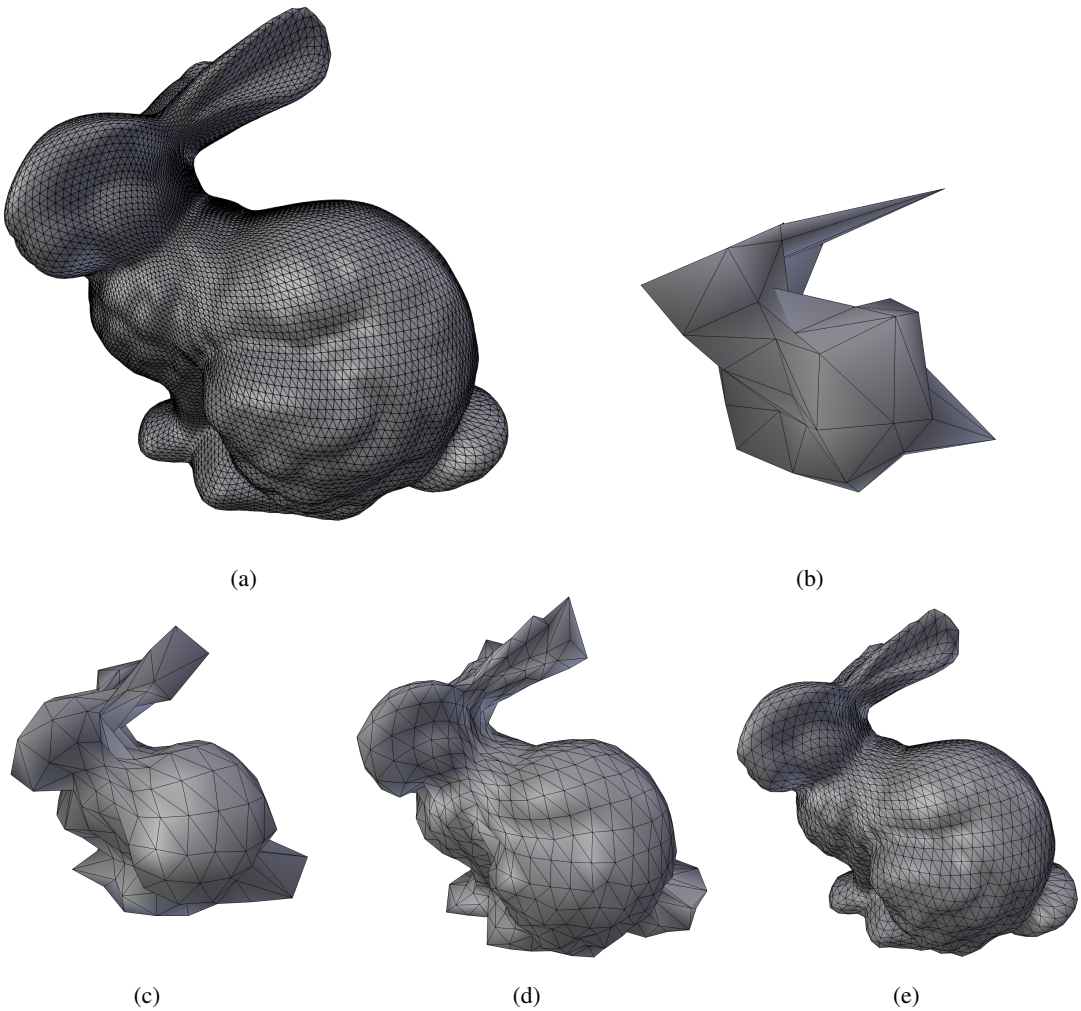


Figure 2.6: An example of multiresolution analysis of a mesh. (a) A bunny. The multiresolution representation of the bunny, which includes four resolution levels. (b) The base mesh is at resolution level-0. (c) The mesh at resolution level- 1. (d) The mesh at resolution level-2. (e) The mesh at resolution level-3.

as shown in Figure 2.6(c); incorporating W_2 into the level-1 mesh yields the bunny at resolution level-2 as shown in Figure 2.6(d); incorporating W_3 into the level-2 mesh yields the bunny at resolution level-3 as shown in Figure 2.6(e).

A multiresolution analysis is associated with a WT. A WT consists of two operations: a FWT and an IWT. For a mesh M at a given resolution level, a single-level FWT (i.e., wavelet analysis) yields a coarser mesh approximating M at the next lower resolution level along with a set of wavelet coefficients that encodes the difference between the coarse mesh and M. A single-level IWT (i.e., wavelet synthesis) combines a coarser mesh and a set of wavelet coefficients to recover the mesh at the next higher resolution level. To give a general sense of a FWT and IWT, we provide an example in Figure 2.7. Figure 2.7(a) illustrates applying a two-level FWT to a cow mesh. After each level of the FWT, the resolution level of the mesh decreases by one, and a set of wavelet coefficients is produced. So, after two levels of the FWT, two sets of wavelet coefficients are produced, denoted by W_1 and W_2. Then, Figure 2.7(b) illustrates applying a two-level IWT to the result of the two-level FWT to compute the mesh at an intermediate resolution and, finally, recover the original mesh at full resolution. In each level of the IWT, a set of wavelet coefficients is used to recover the mesh at the next higher resolution level.

A WT modifies both the geometry and topology of a mesh. The IWT refines the mesh and introduces detail

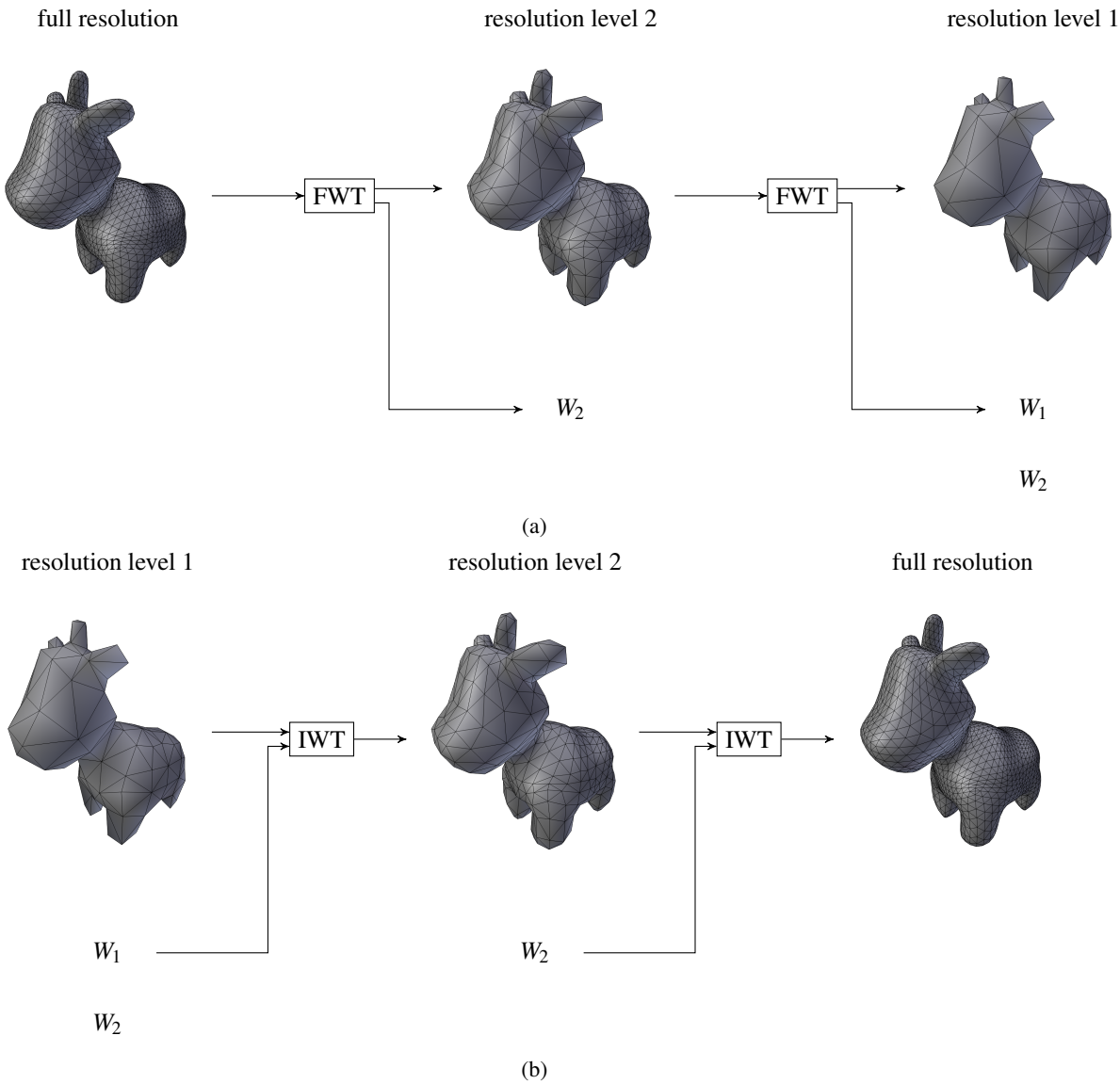


Figure 2.7: Example of a WT. (a) Applying two levels of the FWT to a cow mesh decreases the resolution level of the mesh by two and obtains two sets of wavelet coefficients denoted by W_1 and W_2, where W_1 encodes the details at resolution level 1, and W_2 encodes the details at resolution level 2. (b) Applying two levels of the IWT to the result of the 2-level FWT increases the resolution level of the mesh and recovers the original mesh at full resolution.

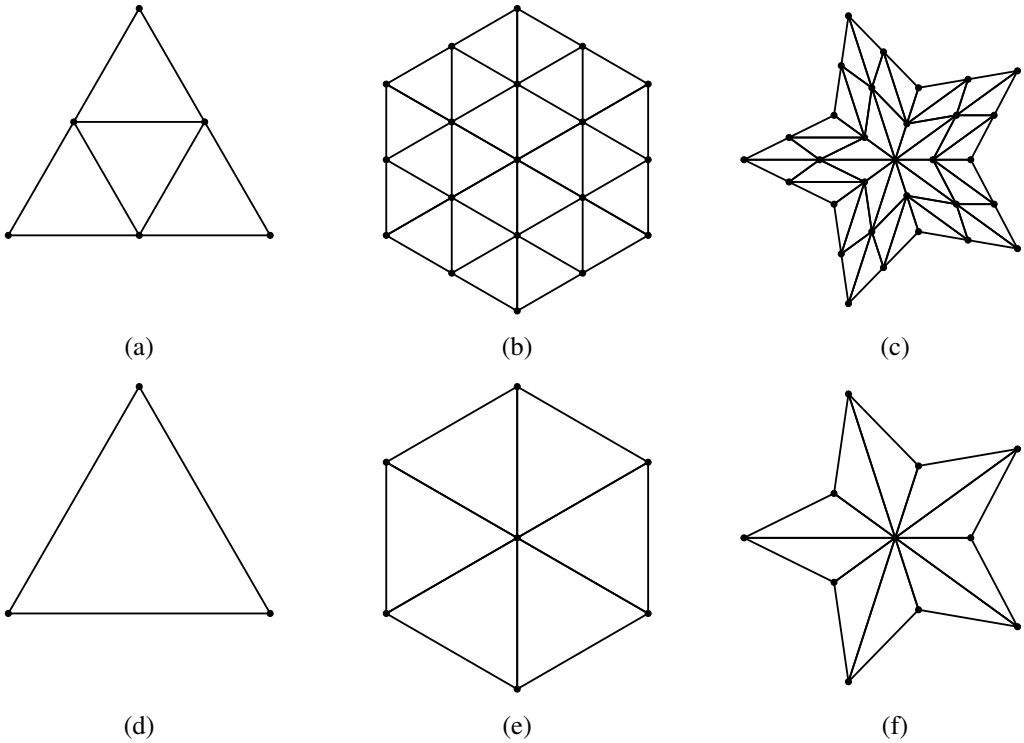
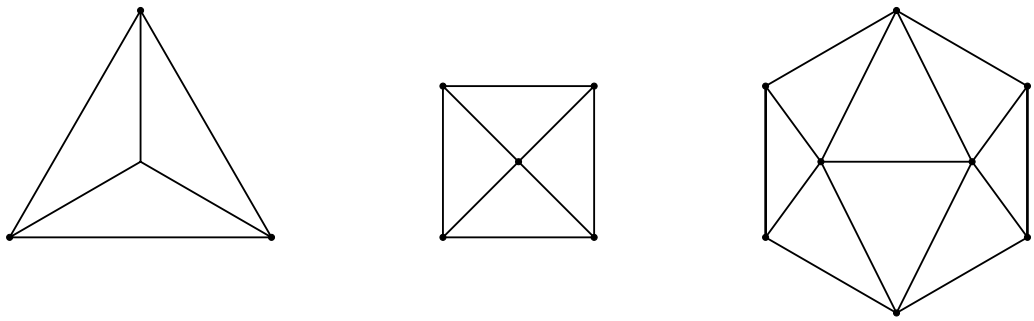


Figure 2.8: Example of meshes with PTQ subdivision connectivity.

(i.e., wavelet coefficients), and the FWT removes detail and produces an approximation. In an implementation, the geometric and topologic modifications are completed in two stages. For the IWT, a refinement step first employs a topologic refinement rule (e.g., PTQ) to introduce new vertices, thereby refining the mesh. Then, the wavelet coefficients are used to determine the positions of the vertices in the refined mesh. For the FWT, the vertices of the mesh are used to determine a set of wavelet coefficients. Then, the vertices corresponding to detail captured by the wavelet coefficients are discarded in a coarsening step to yield the coarse mesh. The FWT, however, has an additional requirement for the input mesh, namely, subdivision connectivity. According to Lounsbery et al. [17], the theoretical basis of the WTs considered herein, subdivision connectivity guarantees the existence of a multiresolution analysis for a mesh. In other words, subdivision connectivity determines which vertices are used to construct the coarse mesh and which vertices are essentially turned into wavelet coefficients. Although some methods [9, 22] extend the WT to arbitrary meshes, they internally convert the input to a mesh with subdivision connectivity. A mesh M is said to have subdivision connectivity if there exists another mesh M' such that applying the topologic refinement rule for subdivision some number of times to M' yields a mesh with the same topology as M. A mesh that has subdivision connectivity with respect to the PTQ topologic refinement rule is said to have PTQ subdivision connectivity. To facilitate understanding, we present some meshes with and without PTQ subdivision connectivity in Figures 2.8 and 2.9. The meshes shown in Figures 2.8(a), (b), and (c) can be yielded from the coarser meshes shown in Figures 2.8(d), (e), and (f) by PTQ, respectively. For the meshes shown in Figure 2.9, we cannot find coarser meshes from which PTQ can yield them, so they do not have PTQ subdivision connectivity.

WTs usually have a relationship to subdivision. A subdivision scheme can be viewed as an IWT with all wavelet coefficients set to zero. Then, inverting the operations of the subdivision scheme obtains the corresponding FWT computation. Such a WT that is derived directly from a subdivision scheme is called a lazy WT. Nonetheless, a lazy WT may not have desirable properties in practice. Thus, modifications to a lazy WT are required to improve its properties (e.g., stability or greater smoothness). In the next section, we introduce the lifting scheme which is commonly used to implement WTs.



(a) (b) (c)

Figure 2.9: Example of meshes without PTQ subdivision connectivity.

2.4.1 Lifting Scheme

The lifting scheme, proposed by Sweldens [25], is a framework for the design, analysis, and implementation of a WT. The lifting scheme guarantees that the implemented WT can be computed in linear time with respect to the number of vertices, and the inverse transform can be yielded trivially. In computation, the lifting scheme first requires the input data (i.e., the vertices of a mesh) to be partitioned into several disjoint sets. Thus, the lifted WT needs to first classify the vertices into disjoint sets. In the single-level IWT, the vertices before topological refinement constitute the set of old vertices, and the vertices introduced by topologic refinement constitute the set of new vertices. Depending on the topologic refinement rule being used, the new vertex set can be further split into multiple sets based on the type of the new vertices. In the single-level FWT, an additional step, subdivision connectivity detection, is required to ensure the WT is well defined (e.g., the vertices can be correctly partitioned). As described earlier, for a mesh M with subdivision connectivity, there exists a control mesh M' such that topologic refinement of M' yields a mesh with the same topology as M. The vertices that exist in both M and M' constitute the set of old vertices, and the vertices that exist in M but not in M' constitute the set of new vertices. Also, the set of new vertices can be further split depending on the topologic refinement rule being used. In the following section, we will illustrate how to detect subdivision connectivity and classify vertices in detail.

After vertex classification, the lifting scheme applies several cascaded lifting steps and scaling steps to the sets of vertices to compute the final result. A lifting step adds (or subtracts) a filtered version of one or more sets to another set. Specifically, a lifting step updates a vertex in a set by adding (or subtracting) a linear combination of vertices in other sets to the vertex. The vertices involved in the computation are usually specified by a mask. A scaling step scales (i.e., multiply or divide a scalar) each of the vertices in a set. By employing the lifting scheme, the computations in a single-level WT are realized as a sequence of lifting steps and scaling steps. In each step, the positions of the vertices in one set are updated, and the updated vertices will be used to modify the vertices in other sets in the following steps. In addition, a lifted WT is trivially invertible by simply reversing the order of the lifting and scaling and inverting their operators. That is, changing addition to subtraction and changing multiplication to division and then reversing the steps in a transform yields its inverse transform. Suppose that a single-level lifted IWT is implemented as a refinement step, a lifting step, a scaling step, and a lifting step. The operations can be illustrated by the diagram shown in Figure 2.10(a), where the sign of a box indicates the operator used in a lifting or scaling step. Then, the inverse of the IWT, the FWT, is defined by the diagram shown in Figure 2.10(b), which includes reversed lifting and scaling steps and a coarsening step. The coarsening step will be illustrated in the following section. Furthermore, repeating the single-level lifted WT yields the multi-level WT. Besides, a multi-level FWT requires the input mesh with the same levels of subdivision connectivity so that the computation steps are well defined.

2.4.2 Vertex Partition and Coarsening

Since vertex partition and mesh coarsening are non-trivial steps, before proceeding further to the lifting framework, we introduce the algorithms of vertex partition and mesh coarsening first. We start by the subdivision connectivity detection algorithm. Based on the detection result, we can partition the vertices as well as coarsen the mesh. Hor-

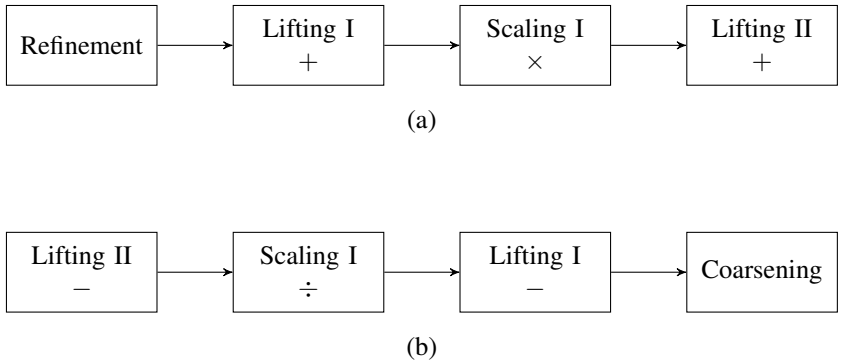


Figure 2.10: Diagrams of the lifted WT steps. (a) The steps of the IWT. (b) The steps of the FWT. The sign of a box indicates the used operator in a lifting or scaling step.

mann [11] and Taubin [26] proposed two subdivision connectivity detection algorithms. Hormann's method, however, has a serious limitation. It cannot applied to a mesh in which all of the vertices are regular. Thus, in what follows, we introduce Taubin's **covering mesh** method, which is used in our lifted WT implementation.

The covering mesh method can detect subdivision connectivity with respect to any topologic refinement rules. In this section, PTQ detection is the primary case in illustrating this method. Since the geometry of a mesh is not relevant to the covering mesh method, we consider only topology. Generally, the covering mesh method identifies subdivision connectivity by two steps: 1) reversing quadrisection on each triangle to produce a coarse triangle known as a **tile** and 2) filtering the tiles. Reversing quadrisection is realized as merging each triangle with its three adjacent triangles to produce a tile for the triangle. The process of reversing quadrisection on a triangle f can be shown in Figure 2.11, where T is the obtained tile. Vertices v_1, v_2, and v_3 are the vertices that compose f's adjacent triangles and are not in f, and they become the **corners** of T. The vertices of f are now the **covered vertices** of T. Reversing quadrisection is skipped on triangles without three adjacent neighbours. The next step is filtering the obtained tiles. Filtering the tiles is realized as grouping the tiles into sets by their connectivity. Two tiles are said to be connected if they share two corners. Then, we check if one of the sets is equivalent to the original mesh, which is realized as checking if all the covered vertices and corners have a one-to-one mapping to the vertices of the original mesh. If there exists a set of tiles that passes this check, the original mesh has subdivision connectivity. Otherwise, the original mesh does not have subdivision connectivity.

Based on the result, we can classify the vertices of the mesh. Clearly, applying PTQ to the satisfied set of tiles yield the original mesh, and the covered vertices of the set of tiles are vertices introduced by PTQ, so they constitute the set of new vertices. The other vertices, all the corners of the set of tiles, constitute the set of old vertices. Figure 2.12 shows an example of detecting subdivision connectivity and classifying vertices of a mesh. Figure 2.12(b) shows the satisfied set of tiles obtained from applying Taubin's method to the mesh shown in Figure 2.12(a). The white dots are the covered vertices of all the tiles, and the black dots are corners of all the tiles. Thus, the vertices of the mesh are classified as shown in Figure 2.12(c). The white dots represent vertices that constitute the set of new vertices, as they were identified as covered vertices in the detection, and the black dots represent vertices that constitute the set of old vertices.

Also, mesh coarsening can be well defined based on the result of subdivision connectivity detection. For a mesh with vertices being classified, the coarsening step is done by removing the new vertices and the edges that connect two new vertices and then linking the broken edges previously joined by a new vertex to reconstruct coarse faces. Figure 2.13 illustrates coarsening the classified mesh shown in Figure 2.12(c), where empty dots and dashed lines in the left mesh refer to vertices and edges to be removed. The remaining old vertices and edges and faces that link them constitute the coarsened mesh.

2.4.3 Lifted Loop Subdivision

Back to the lifting framework, to facilitate understanding, we present a lifted version of Loop subdivision as follows. Consider a closed triangle mesh, of which the vertices are denoted by an old vertex set V_{old}. After the topologic

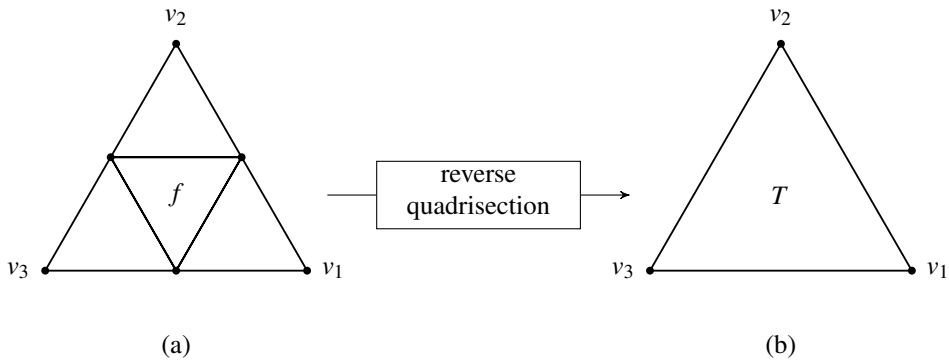


Figure 2.11: Example of reversing quadrisecting on a triangle f to build its tile T. Vertices v_1, v_2, and v_3 are corners of T, and vertices of f are corners of T.

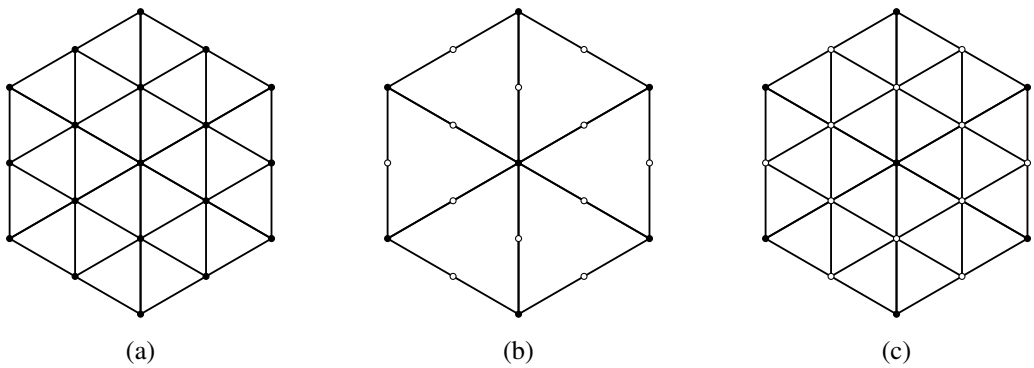


Figure 2.12: Example of detecting subdivision connectivity. (a) The mesh. (b) The set of tiles that passes the vertex mapping check. The black and white dots are corners and covered vertices of the tiles. (c) Vertex classification based on the result of subdivision connectivity detection. White dots represent vertices that constitute the set of new vertices, and black dots represent vertices that constitute the set of old vertices.

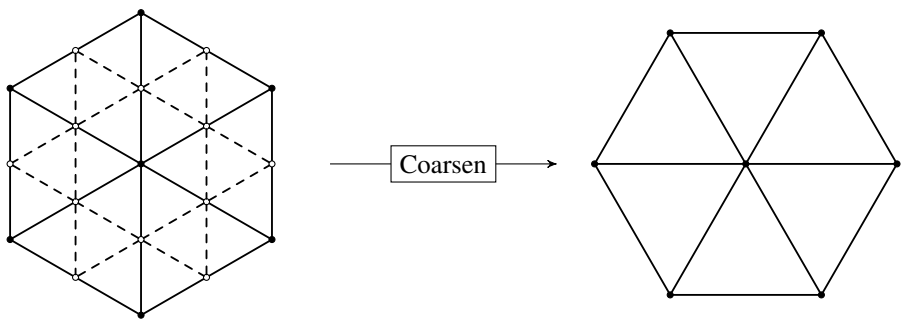


Figure 2.13: Example of coarsening the mesh shown in Figure 2.8(b). Empty dots and dashed lines in the right mesh refer to vertices and edges to be removed.

refinement, PTQ introduces new vertices denoted by a new vertex set V_{new} to the mesh. The vertices in the mesh now can be viewed as in two disjoint sets, the old and new vertex set, which satisfy the partition requirement of the lifting scheme. The lifted Loop subdivision consists of two lifting steps and one scaling step. These steps are sequentially performed, and a single step updates either the new vertex set or the old vertex set. The computations of the three steps are defined as follows:

1. **Lifting I**. The first lifting step adds a filtered version of the old vertex set to the new vertex set. Let v be a vertex in V_{new} and v_1, v_2, v_3, and v_4 be vertices in V_{old}. The topologic relations between v_1, v_2, v_3, and v_4 and v are specified by the mask shown in Figure 2.14(a). Then, the position of v is given by
$$v = \frac{3}{8}(v_1 + v_2) + \frac{1}{8}(v_3 + v_4).$$

2. **Scaling**. The scaling step multiplies the old vertex set by a scalar. Let v be a vertex in V_{old} with valence n. Then, the updated value v' of v is given by
$$v' = \beta_n v,$$
where
$$\beta_n = \frac{8}{5}\left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2.$$

3. **Lifting II**. The second lifting step adds a filtered version of the new vertex set to the old vertex set. Let v be a vertex in V_{old}, and v_1, v_2 ..., and v_n be the n 1-ring neighbours of v, which are also in V_{new}. Their topologic relations are specified by the mask shown in Figure 2.14(b). Then, the updated value v' of v is given by
$$v' = v + \delta_n \sum_{i=1}^{n} e_i,$$
where
$$\delta_n = \frac{1}{n}\left[1 - \frac{8}{5}\left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2\right].$$

The reader can verify that the lifted Loop subdivision produces the same result as the classical Loop subdivision. The advantage is that the lifted Loop subdivision is reversible, which serves as a starting point for designing a lifted WT. Also, additional lifting or scaling steps can be introduced to improve the mathematical properties of the transform. In the following section, we present the lifted Loop WT which is derived from the lifted Loop subdivision.

2.4.4 Lifted Loop WT

Bertram [3] introduced an additional lifting step to the lifted Loop subdivision to define the Lifted Loop WT for a closed mesh. Then, Wang et al. [30] introduce more lifting steps to extend the lifted Loop WT to meshes with boundaries. As described earlier, the lifting scheme guarantees the WT is trivially invertible. We start by introducing the lifted Loop FWT. Then the IWT can be yielded by reversing the operations of the FWT.

The computations in the single-level FWT consists of six lifting steps and a scaling step. After incorporating the coarsening step, the operations of lifted Loop FWT can be illustrated by the diagram shown in Figure 2.15, where the sign of a box indicates the operator used in a lifting or scaling step. According to Wang et al. [30], for the FWT, the vertices of a mesh are split into two sets: an old vertex set V_{old} and a new vertex set V_{new}. The vertices that have been introduced by PTQ belong to V_{new}, and the other vertices belong to V_{old}. Given the partition of the vertices, the computations in FWT are defined as below:

1. **Lifting I**. The first lifting step subtracts a filtered version of the boundary vertices in V_{new} from those in V_{old}. Let v be a boundary vertex in V_{old}, and v_1 and v_2 be boundary vertices from V_{new}. The relative positions of v_1 and v_2 to v are specified by the mask shown in Figure 2.16(a). Then, the updated value v' of v is given by
$$v' = v - \frac{1}{4}(v_1 + v_2).$$

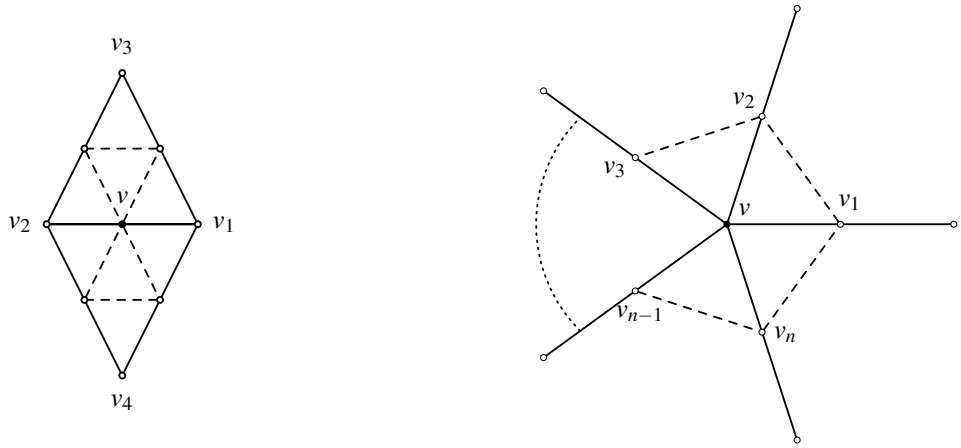


Figure 2.14: Masks of the lifted Loop subdivision for a closed triangle mesh.

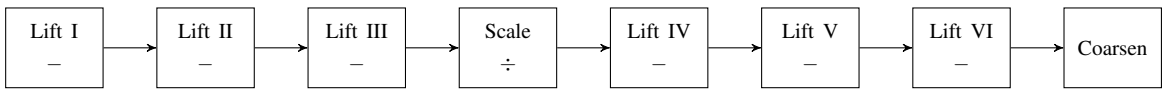


Figure 2.15: The FWT for a one-level lifted Loop WT.

2. **Lifting II**. The second lifting step subtracts a filtered version of the boundary vertices in V_{old} from those in V_{new}. Let v be a boundary vertex in V_{new}, and v_1 and v_2 be vertices from V_{old}. The relative positions of v_1 and v_2 to v are specified by the mask shown in Figure 2.16(b). Then, the updated value v' of v is given by

$$v' = v - \frac{1}{2}(v_1 + v_2).$$

3. **Lifting III**. The third lifting step subtracts a filtered version of the interior vertices in V_{new} from those in V_{old}. Let v be a vertex in V_{old}, and v_1, v_2 ..., and v_n be the n 1-ring neighbours of v, which are also in V_{new}. The relative positions of v_1, v_2 ..., and v_n to v are specified by the mask shown in Figure 2.16(c). Then, the updated value v' of v is given by

$$v' = v - \delta_n \sum_{j=1}^{n} v_j,$$

where

$$\delta_n = \frac{1}{n}\left[1 - \frac{8}{5}\left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2\right].$$

4. **Scaling**. The scaling step divides the interior vertices in V_{old} by a scalar. Let v be an interior vertex of valence n in V_{old}. The updated value of v' of v is given by

$$v' = \frac{v}{\beta_n},$$

where

$$\beta_n = \frac{8}{5}\left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2.$$

5. **Lifting IV**. The fourth lifting step subtracts a filtered version of the vertices in V_{old} from the interior vertices in V_{new}. Let v be an interior vertex in V_{new}, and v_1, v_2, v_3, and v_4 be vertices from V_{old}. The relative positions of v_1,

v_2, v_3, and v_4 to v are specified by the mask shown in Figure 2.16(d). Then, the updated value v' of v is given by

$$v' = v - \left[\frac{3}{8}(v_1+v_2) + \frac{1}{8}(v_3+v_4)\right].$$

6. **Lifting V**. The fifth lifting step subtracts a filtered version of the boundary vertices in V_{new} from those in V_{old} Let v_1, v_2, v_3, and v_4 be boundary vertices in V_{old}, and v be a boundary vertex from V_{new}. The relative positions of v_1, v_2, v_3, and v_4 to v are specified by the mask shown in Figure 2.16(e). Then, the updated values of v'_1, v'_2, v'_3, and v'_4 of v_1, v_2, v_3, and v_4 are given by

$$v'_i = v_i - \eta_i v \quad \forall i = 1,2,3,4,$$

where

$$\eta_1 = \eta_4 = -0.525336 \text{ and } \eta_2 = \eta_3 = 0.189068.$$

7. **Lifting VI**. The last lifting step subtracts a filtered version of the interior vertices in V_{new} from the vertices in V_{old}. Let v_1, v_2, v_3, and v_4 be vertices in V_{old}, and v be an interior vertex from V_{new}. The relative positions of v_1, v_2, v_3, and v_4 to v are specified by the mask shown in Figure 2.16(f). Let α_i, β_i, γ_i, and δ_i and ω_i be the coefficients and weight associated with v_i of valence n_i, where $i \in \{1,2,3,4\}$. Coefficients α_i, β_i, γ_i, and δ_i are defined as

$$\begin{aligned}
\alpha_i &= \frac{3}{8} + \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n_i}\right)^2, \\
\beta_i &= \frac{8}{5}\left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n_i}\right)^2, \\
\gamma_i &= \frac{1}{n_i}\left[\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n_i}\right)^2\right], \text{ and} \\
\delta_i &= \frac{1}{n_i}\left[1 - \frac{8}{5}\left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n_i}\right)^2\right].
\end{aligned}$$

The four weights ω_i ($i = 1,2,3,4$) are calculated by solving the following linear equation:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}. \tag{2.1}$$

The elements of the left side 4×4 matrix in Equation 2.1 are given by

$$
\begin{aligned}
a_{11} &= \alpha_1^2 + \gamma_2^2 + \gamma_3^2 + \gamma_4^2 + \frac{1}{256}(n_0 - 3) + \frac{5}{32}n_0, \\
a_{12} &= a_{21} = \alpha_1\gamma_1 + \gamma_2\alpha_2 + \gamma_3^2 + \gamma_4^2 + \frac{21}{64}, \\
a_{13} &= a_{31} = \alpha_1\gamma_1 + \gamma_2^2 + \gamma_3\alpha_3 + \frac{85}{256}, \\
a_{14} &= a_{41} = \alpha_1\gamma_1 + \gamma_2^2 + \gamma_4\alpha_4 + \frac{85}{256}, \\
a_{22} &= \gamma_1^2 + \alpha_2^2 + \gamma_3^2 + \gamma_4^2 + \frac{1}{256}(n_1 - 3) + \frac{5}{32}n_1, \\
a_{23} &= a_{32} = \gamma_0^2 + \alpha_1\gamma_1 + \gamma_2\alpha_2 + \frac{85}{256}, \\
a_{24} &= a_{42} = \gamma_0^2 + \alpha_1\gamma_1 + \gamma_3\alpha_3 + \frac{85}{256}, \\
a_{33} &= \gamma_0^2 + \gamma_1^2 + \alpha_2^2 + \frac{1}{256}(n_2 - 2) + \frac{5}{32}n_2, \\
a_{34} &= a_{43} = \gamma_0^2 + \gamma_1^2 + \frac{1}{64}, \text{ and} \\
a_{44} &= \gamma_0^2 + \gamma_1^2 + \alpha_3^2 + \frac{1}{256}(n_3 - 2) + \frac{5}{32}n_3.
\end{aligned}
$$

The elements of the right side 4×1matrix in Equation 2.1 are given by

$$b_1 = \alpha_0\delta_0 + \gamma_1\delta_1 + \frac{3}{8}, \; b_2 = \gamma_0\delta_0 + \alpha_1\delta_1 + \frac{3}{8}, \text{ and } \; b_3 = b_4 = \gamma_0\delta_0 + \gamma_1\delta_1 + \frac{1}{8}.$$

Then, the updated value v_i' of v_i is given by

$$v_i' = v_i - \omega_i v \quad \forall i = 1,2,3,4.$$

After the computations, a coarsening step, as described earlier, is employed in order to remove the vertices in V_{new} from the input mesh. These new vertices then become wavelet coefficients, and the old vertices remain in the coarse mesh.

The single-level IWT can be easily defined by reversing the operations of the single-level FWT. First, PTQ is employed in order to refine the input mesh, and the vertices are also partitioned. The vertices introduced by PTQ belong to the new vertex set, and the other vertices belong to the old vertex set. The vertices have the same classification as in the FWT. In addition, wavelet coefficients are used as the initial positions for the new vertices. Then, the IWT applies the lifting/scaling steps in the reverse order to compute the refined mesh. The operations in the IWT are illustrated by the diagram in Figure 2.17. In each step, the computation is defined as inverting the operator (e.g., changing subtraction to addition or changing division to multiplication) of the corresponding step in the FWT. The masks shown in Figure 2.16 are also used in the corresponding steps in the IWT.

2.4.5 Lifted Butterfly WT

Having described the lifted Loop WT, we now introduce the lifted Butterfly WT, originally proposed by Sweldens et al. [24] for closed triangle meshes. The lifted Butterfly WT shares the same coarsening and refinement operations with the lifted Loop WT. Therefore, in what follows, we focus on introducing the lifting computations. We start by the single-level FWT. It consists of two lifting steps. The diagram shown in Figure 2.18 illustrates the operations in the FWT, where the sign of a box indicates the operator used in a lifting step. The Butterfly FWT separates vertices in the same way as in the Loop FWT. For the input mesh, the vertices that have been introduced by PTQ are classified as in a new vertex set V_{new}, and the other vertices are classified as in an old vertex set V_{old}. Given this vertex partition, the computations in the single-level FWT are defined as below:

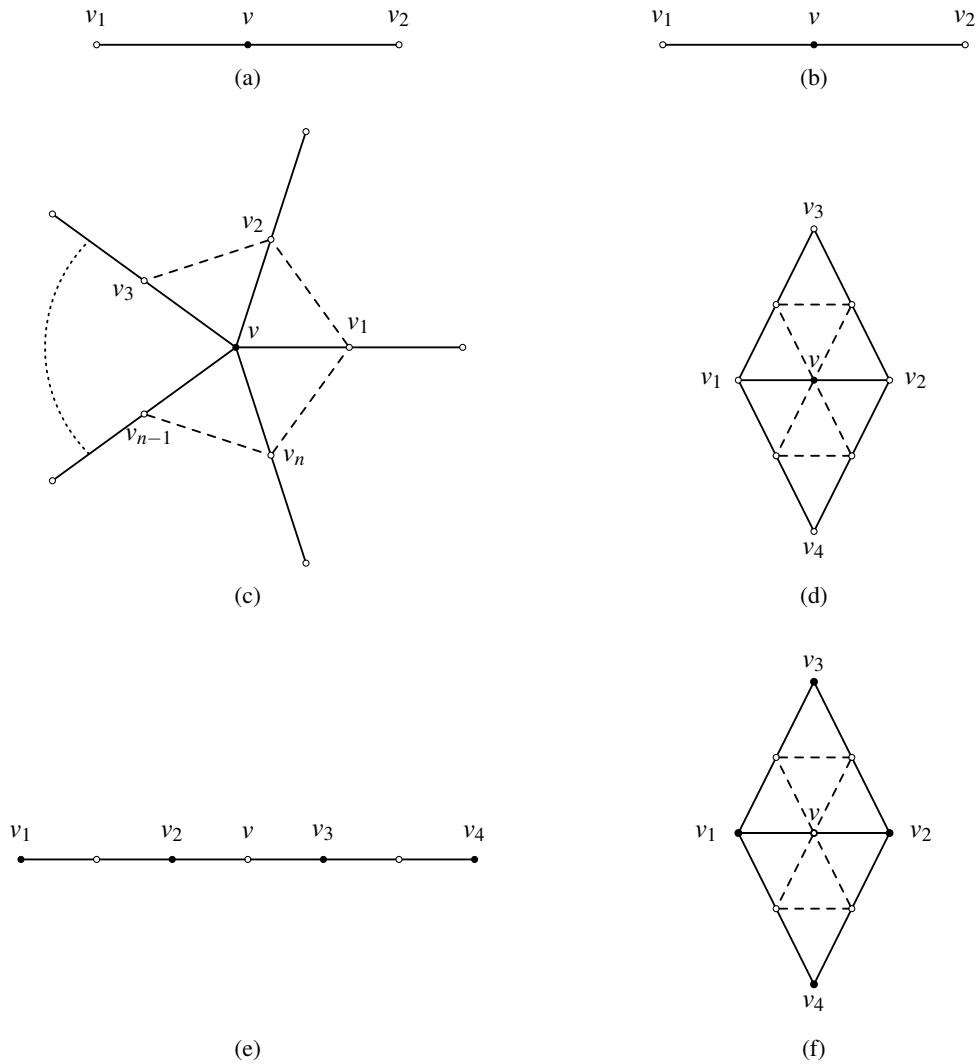


Figure 2.16: Masks for the lifted Loop WT.

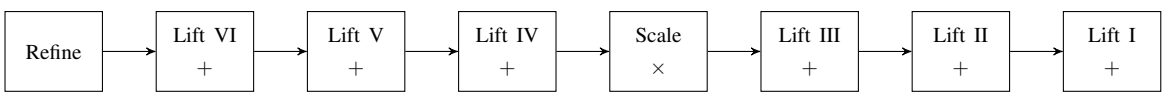


Figure 2.17: The IWT for a single-level Loop WT.

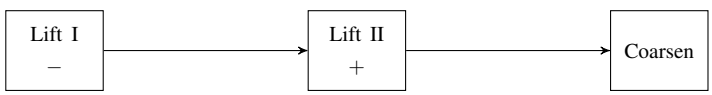


Figure 2.18: The FWT of a single-level Butterfly WT.

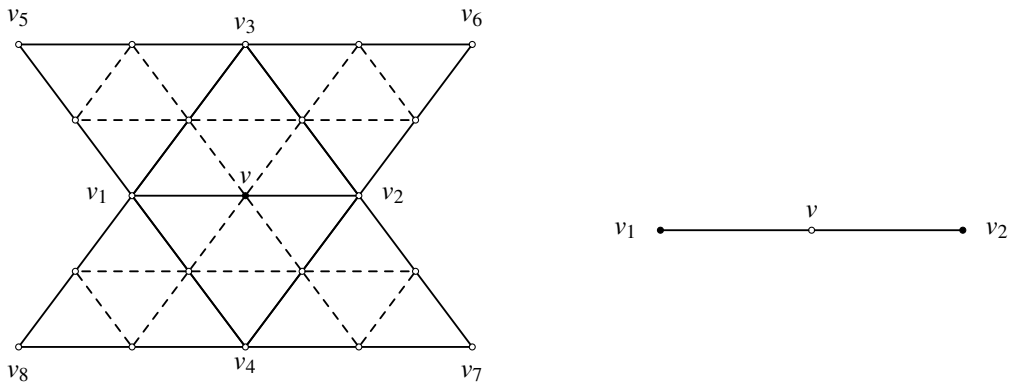


Figure 2.19: Masks used in the lifted Butterfly WT.

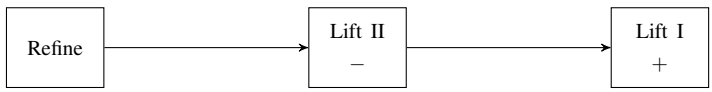


Figure 2.20: The IWT for a single-level lifted Butterfly WT.

1. **Lifting I**. The first lifting step subtracts a filtered version of the vertices in V_{old} from those in V_{new}. Let v be a vertex from V_{new}, and $\{v_i\}_{i=1}^{8}$ be vertices from V_{old}. The relative positions of $\{v_i\}_{i=1}^{8}$ to e are specified by the mask shown in Figure 2.19(a). Then, the updated value v' of v is given by

$$v' = v - \left[\frac{1}{2}(v_1+v_2) + \frac{1}{8}(v_3+v_4) - \frac{1}{16}(v_5+v_6+v_7+v_8)\right].$$

2. **Lifting II**. The second lifting step adds a filtered version of the vertices in V_{new} to those in V_{old}. Let v_1 and v_2 be vertices from V_{old}, and v be a vertex from V_{new}. The relative positions of v_1 and v_2 to v are specified by the mask shown in Figure 2.19(b). Then, the updated values v_1' and v_2' of v_1 and v_2 are given by

$$v_i' = v_i + s_i v \quad \forall i = 1,2, \tag{2.2}$$

where s_i is a weight of v_i. The calculation of s_i will be discussed later.

As with obtaining the IWT from the Loop FWT, the Butterfly IWT is defined as first employing PTQ and then reversing the two lifting steps and their operators. The operations in the single-level IWT are illustrated in Figure 2.20.

The multi-level Butterfly WT can be implemented by iteratively applying the single-level FWT or IWT. Next, we consider the calculation of the weight s_i used in Equation 2.2, which is affected by the number of levels in the FWT or IWT. From the originally proposed calculation [24], we conclude a closed formula for computing the weights, which works on a closed mesh. Let L be the number of levels in the WT, and let j denote the current level in the WT, where $j \in \{1,2,\ldots,L\}$ and $j = 1$ is the coarsest level and $j = L$ is the finest level. Then, the weight s_i for vertex v_i can be computed using the formula:

$$s_i = \frac{4^{L-j}-1}{2\left[1+\frac{n}{6}(4^{L-j-1}-1)\right]},$$

where n is the valence of v_i.

Chapter 3

Wavelet Transform Toolkit

3.1 Introduction

This chapter introduces the Wavelet Transform Toolkit (WTT), the software developed by the author. The WTT consists of a C++ header-only library and three application programs. The library provides application programming interface (API) for computing multi-level WTs on a 3-D mesh. The API is sufficiently generic that allows users to add definitions of custom WTs. For convenient use, the library provides several built-in functions to assist in defining a WT. In the case that the Loop and Butterfly WTs are of interest, the library offers some wrapper functions that users can call directly to perform the WTs for a triangle mesh. The application programs demonstrate WT-based applications, including compression and denoising.

The remainder of this chapter is organized as follows. We start by explaining how to build and install the WTT. This is then followed by an introduction to some basic concepts related to the WTT. After that, we give an overview of and describe its APIs. Lastly, descriptions of the demonstration application programs are given.

3.2 Software Installation

The WTT is written in C++ and utilizes many C++17 features. Therefore, in order to build the WTT, a compiler with C++17 support is needed. The use of GCC 7.2.0 or higher and Clang 5.0.0 or higher are recommended as C++ compilers. Since the WTT has the Boost Library [5] and the Computational Geometry Algorithm Library (CGAL) [7] as dependencies, these dependencies should be installed prior to building the WTT. The following library versions have been verified to work with the WTT:

- CGAL 4.2.0 and above (CGAL 5.0 is recommended).
- Boost 1.58.0 and above (Boost 1.71.0 is recommended).

The WTT supports a build process based on CMake [13]. In what follows, let `$SOURCE_DIR` denote the top-level directory of the WTT software distribution, `$BUILD_DIR` denote a directory where build files are created, and `$INSTALL_DIR` denote a directory where the software is to be installed. To build and install the WTT, perform the following steps (in order):

1. Generate the native build files by running the command:

```
cmake -H$SOURCE_DIR -B$BUILD_DIR -DCMAKE_INSTALL_PREFIX=$INSTALL_DIR
```

2. Build the software by running the command:

```
cmake --build $BUILD_DIR --clean-first
```

3. Install the software by running the command:

```
cmake --build $BUILD_DIR --target install
```

3.3 Concepts

Before proceeding further, we introduce some basic concepts related to the WTT. Readers are expected to be familiar with the basics of C++. The WTT is developed based on CGAL and uses the template class `CGAL::Polyhedron_3` as the type of a mesh. The type `Polyhedron_3` utilizes the halfedge data structure [19, 31] and provides member functions to access the vertices, edges, and faces of a mesh. A vertex is accessed through the `Vertex_handle` and `Vertex_const_handle` handle classes. Users can use the handles to query and modify the position (i.e., x, y, and z coordinates) of a vertex and access its adjacent vertices.

In addition to the position, we introduce additional properties (i.e., the level, type, ID, and border properties) to a vertex. The level property, an integer starting from 0, represents the resolution of a vertex. The type property, an integer, generally represents the type of a vertex produced by a topologic refinement rule. The ID property, an integer, is the unique identifier of a vertex. Lastly, the border property, a boolean, indicates if a vertex is on the boundary.

3.4 Library

The library provides the capability to define and compute multi-level WTs on a 3-D mesh. Since this library is header-only, users only need to include the corresponding headers to use the library. The library provides a generic API, which is composed of several template classes, to allow users to add definitions to customize a WT. The library decomposes computing a WT into several operations. Users could add their own definitions to the operations (e.g., vertex classification, topologic modifications, lifting computations,) for customization. For convenience of use, some predefined operations are provided to assist in defining a WT. If PTQ is employed in a WT, users can use built-in functions for classifying vertices and coarsening and refining the mesh. By defining the lifting computations, one can customize the WT. Furthermore, the library offers built-in functions that implement the Loop and Butterfly WTs for direct use. In what follows, we illustrate the usage of the library in detail.

3.4.1 Usage of API

We start by describing the generic API for defining and computing a multi-level WT. In order to define a custom WT, users can provide definitions of the special classes. Based on the functionality, the classes are divided into computational classes and operational classes. Generally, defining a custom WT is realized as instantiating the operational classes first and then using them to instantiate the computational classes. The objects of the instantiated computational classes can be used as normal function calls to compute the FWT and IWT. The computational classes are 1) `Wavelet_analyze` and 2) `Wavelet_synthesize`, which implement the functionality for computing the multi-level FWT and IWT, respectively. They accept the operational classes as template parameters for instantiation. The operational classes are 1) `Wavelet_mesh_operations` which specifies operations to vertices, and 2) `Wavelet_analysis_ops` and 3) `Wavelet_synthesis_ops`, which abstract the operations of the FWT and IWT. User-defined functions can be used to instantiate the three classes, and these functions finally propagate to the computational classes to compute a custom WT. Next, we consider a code example to illustrate how to define a custom WT as well as use some of the built-in functions.

A complete example of defining and computing a WT is shown in Listing 3.1. In lines 1 to 20, we include the required headers and define several handy type aliases. As described earlier, in order to define a WT, the operational classes, `Wavelet_mesh_operations`, `Wavelet_analysis_ops`, and `Wavelet_synthesis_ops`, need to be instantiated first. The three template classes define the interfaces of the operations in the FWT and IWT. User-defined functions, which realize the operations, are passed in via the template parameters and constructors of the operational classes and wrapped in their member functions. The operational classes are used as adaptors that the computational class can use to access the user-defined operations for computing a custom WT through their member functions. Illustrating the instantiations of `Wavelet_mesh_operations`, `Wavelet_analysis_ops`, and `Wavelet_synthesis_ops` with a description of the code is given in what follows.

Listing 3.1: Example of defining and computing a WT.

```
1  #include <wtlib/ptq_impl/classify_vertices.hpp>
2  #include <wtlib/ptq_impl/subdivision_modifier.hpp>
```

```cpp
#include <wtlib/wavelet_mesh_operations.hpp>
#include <wtlib/wavelet_operations.hpp>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/Simple_cartesian.h>
#include <map>

using K = CGAL::Simple_cartesian<double>;
using Mesh = CGAL::Polyhedron_3<K>;
using Vector3 = Mesh::Traits::Vector_3;
using Vertex_handle = Mesh::Vertex_handle;
using Vertex_const_handle = Mesh::Vertex_const_handle;

struct Vertex_info {
    int  level;
    int  type;
    int  id;
    bool border;
};

int main() {
  std::map<Vertex_const_handle, Vertex_info> properties;
  auto get_level = [&properties](Vertex_const_handle v) -> int {
    return properties.at(v).level;
  };
  auto set_level = [&properties](Vertex_handle v, int level) -> void {
    properties[v].level = level;
  };

  auto get_type = [&properties](Vertex_const_handle v) -> int {
    return properties.at(v).type;
  };
  auto set_type = [&properties](Vertex_handle v, int type) -> void {
    properties[v].type = type;
  };

  auto get_id = [&properties](Vertex_const_handle v) -> int {
    return properties.at(v).id;
  };
  auto set_id = [&properties](Vertex_handle v, int id) -> void {
    properties[v].id = id;
  };

  auto get_border = [&properties](Vertex_const_handle v) -> bool {
    return properties.at(v).border;
  };
  auto set_border = [&properties](Vertex_handle v, bool border) -> void {
    properties[v].border = border;
  };

  using Mesh_ops = wtlib::Wavelet_mesh_operations<Mesh,
                                                  decltype(get_id),
                                                  decltype(set_id),
                                                  decltype(get_level),
                                                  decltype(set_level),
                                                  decltype(get_type),
                                                  decltype(set_type),
                                                  decltype(get_border),
```

```
60                                                  decltype(set_border)>;
61
62    Mesh_ops mesh_ops(get_id, set_id,
63                      get_level, set_level,
64                      get_type, set_type,
65                      get_border, set_border);
66
67    auto get_num_types = [](Mesh& mesh, const Mesh_ops& mesh_ops) -> int {
68      /**
69        * Return the number of types of vertices, which depends on the topologic
70        * refinement rule. If PTQ is used, this function return 2, as below.
71        */
72      return 2;
73    };
74
75    auto classify = [](Mesh& mesh, const Mesh_ops& mesh_ops, int num_levels,
76                       std::vector<Vertex_handle>& vertices,
77                       std::vector<Vertex_handle*>& bands) -> bool {
78      /**
79      *  Detect subdivision connectivity and classify vertices. If PTQ is used in
80      *  the WT, the implementation below can be used.
81      */
82      return wtlib::ptq_impl::
83                PTQ_classify_vertices<Mesh, Mesh_ops>::
84                  classify(mesh, mesh_ops, num_levels, vertices, bands, true);
85    };
86
87    auto fwt_init = [](Mesh& mesh, const Mesh_ops& mesh_ops, int num_levels) -> void
        {
88      // Do any necessary initialization before lifting.
89    };
90
91    auto fwt_cleanup = [](Mesh& mesh, const Mesh_ops& mesh_ops) -> void {}
92      // Cleanup resources acquired in initialization after the FWT is done.
93    };
94
95    auto fwt_lift = [](Mesh& mesh, const Mesh_ops& mesh_ops,
96                       Vertex_handle** first,
97                       Vertex_handle** last) -> void {
98      // Do lifting calculation.
99    };
100
101   auto coarsen = [](Mesh& mesh, const Mesh_ops& mesh_ops, int num_levels) -> void {
102     /**
103       * Coarsen the mesh after lifting, depending on the topologic refinement
104       * rule. If PTQ is used, the implementation below can be used.
105       */
106     wtlib::ptq_impl::PTQ_subdivision_modifier<Mesh, Mesh_ops>::coarsen(mesh,
          mesh_ops, num_levels);
107   };
108
109   using FWT_ops = wtlib::Wavelet_analysis_ops<Mesh, Mesh_ops,
110                                               decltype(get_num_types),
111                                               decltype(classify),
112                                               decltype(fwt_init),
113                                               decltype(fwt_cleanup),
114                                               decltype(fwt_lift),
```

```
115                                                 decltype(coarsen)>;
116
117   FWT_ops fwt_ops(get_num_types, classify, fwt_init,
118                   fwt_cleanup, fwt_lift, coarsen);
119
120   auto get_mesh_size = [](Mesh& mesh,
121                           const Mesh_ops& mesh_ops,
122                           int num_levels) -> int {
123     /**
124      * Predicate the number of vertices in the final mesh after num_levels
125      * levels IWT. If PTQ is used, the implementation below could be used.
126      */
127      return wtlib::ptq_impl::
128               PTQ_subdivision_modifier<Mesh, Mesh_ops>::
129                 get_mesh_size(mesh, mesh_ops, num_levels);
130   };
131
132   auto iwt_init = [](Mesh& mesh, const Mesh_ops& mesh_ops, int num_levels) -> void
       {
133     // Do any necessary initialization before lifting.
134   };
135
136   auto iwt_cleanup = [](Mesh& mesh, const Mesh_ops& mesh_ops) -> void {
137     // Cleanup resources acquired in initialization after the IWT is done.
138   };
139
140   auto refine = [](Mesh& mesh, const Mesh_ops& mesh_ops,
141                    int num_levels,
142                    std::vector<Vertex_handle>& vertices,
143                    std::vector<Vertex_handle*>& bands) -> void {
144     /**
145       * Refine the mesh by a topologic refinement rule before lifting. If PTQ is
146       * used, the implementation below can be used.
147       */
148     wtlib::ptq_impl::
149       PTQ_subdivision_modifier<Mesh, Mesh_ops>::
150         refine(mesh, mesh_ops, num_levels, vertices, bands);
151   };
152
153   auto iwt_lift = [](Mesh& mesh, const Mesh_ops& mesh_ops,
154                      Vertex_handle** first,
155                      Vertex_handle** last) -> void {
156     // Do lifting calculation.
157   };
158
159   using IWT_ops = wtlib::Wavelet_synthesis_ops<Mesh,
160                                                Mesh_ops,
161                                                decltype(get_num_types),
162                                                decltype(get_mesh_size),
163                                                decltype(iwt_init),
164                                                decltype(iwt_cleanup),
165                                                decltype(refine),
166                                                decltype(iwt_lift)>;
167
168   IWT_ops iwt_ops(get_num_types,
169                   get_mesh_size,
170                   iwt_init,
```

```
171                     iwt_cleanup,
172                     refine,
173                     iwt_lift);
174
175     using FWT = wtlib::Wavelet_analyze<Mesh_ops, FWT_ops>;
176     using IWT = wtlib::Wavelet_synthesize<Mesh_ops, IWT_ops>;
177     FWT fwt(mesh_ops, fwt_ops);
178     IWT iwt(mesh_ops, iwt_ops);
179
180     Mesh mesh;
181     if (!std::cin >> mesh) {
182       std::cerr << "Fail to read mesh.\n";
183       std::exit(1);
184     }
185     std::vector<std::vector<Vector3>> coefs;
186     int num_levels = 4;
187
188     if (!fwt(mesh, coefs, num_levels)) {
189       std::cerr << "Mesh does not have "
190                 << num_levels
191                 << " levels subdivision connectivity.\n";
192       std::exit(1);
193     }
194
195     iwt(mesh, coefs, num_levels);
196     return 0;
197   }
```

Usage of `Wavelet_mesh_operations`

Proceeding to the `main` function, lines 23 to 65 instantiate the `Wavelet_mesh_operations` class as `Mesh_ops` and define an object named as `mesh_ops`. Class `Wavelet_mesh_operations` specifies the operations of accessing the level, type, ID, and border properties of a vertex. First, line 23 defines a vertex property map, where the properties of each vertex are to be stored. Then, lines 24 to 50 define several lambda functions for getting/setting each of the properties. The implementation of the property access is not limited to lambda functions, which could be functions or functional objects as long as their interfaces match with that of the corresponding lambda functions. After that, line 52 instantiates `Wavelet_mesh_operations` as `Mesh_ops` by the declared types of the lambdas. Line 62 constructs the `Mesh_ops` object `mesh_ops` by these lambda functions, and each of these lambda functions is wrapped in a corresponding member function of `mesh_ops`.

Usage of `Wavelet_analysis_ops`

Next, lines 67 to 118 instantiate the `Wavelet_analysis_ops` class as `FWT_ops` and define an object named as `fwt_ops`. Class `Wavelet_analysis_ops` abstracts the operations of the FWT. lines 67 to 107 define several lambda functions that implement the required operations. Descriptions of the lambda functions are given in what follows.

The lambda function defined in line 67 provides the number of vertex types in the WT. Generally, the number of types is determined by the number of types of vertices introduced in topologic refinement plus one. Suppose that PTQ is used in the WT. In this case, the function should return 2.

Line 75 defines a lambda function where detecting subdivision connectivity and classifying vertices should be implemented. Since this operation is not trivial, the built-in function for the PTQ case is used. For a custom implementation, the following requirements should be satisfied in order to embed the implementation into the WTT library correctly. The return value is a boolean flag indicating if `levels` levels of subdivision connectivity was successfully detected in the mesh. If successful, the level, type, and ID properties should be assigned to each vertex. Handles of all the vertices should be placed in the array `vertices` and sorted by level, type, and ID in ascending order. Array `bands`

should be populated continuously by pointers to the starting vertex handle of each type of each level in `vertices` and the pointer to the end of `vertices`.

The lambda functions defined in lines 87 and 91 are used to perform any necessary initialization and cleanup before and after the multi-level FWT computations. Additional resources can be allocated and released in this pair of functions.

Line 95 defines a lambda function where the lifting steps of the single-level FWT should be populated. This function is called iteratively to perform the multi-level FWT. For a single level WT, the vertices that participate in the computations are determined by `*first` and `*last`. Users could design custom lifting computations for these vertices. The vertices below the current level (i.e., the old vertices) can be accessed continuously through the handles in between `*first` and `*(first+1)`, and the vertices at the current level (i.e., the new vertices) can be accessed continuously through the handles in between `*(first+1)` and `*last`. Incrementing `*(first+1)` and dereferencing it obtains the pointer to the handle of the starting vertex of the next type among the new vertices, if there is more than one type.

Line 101 defines a lambda function where the coarsening should be implemented. This function is also called iteratively to perform the multi-level FWT. Since this operation is not trivial, the built-in implementation for the PTQ case is used. In a custom implementation, the vertices whose levels equal to `level` are expected to be removed from the mesh.

After all the required operations are defined, line 109 instantiates `Wavelet_analysis_ops` as `FWT_ops` by the declared types of the lambdas. line 177 constructs the `FWT_ops` object `fwt_ops` by these lambda functions, and each of these lambda functions is wrapped in a corresponding member function of `fwt_ops`.

Usage of `Wavelet_synthesis_ops`

Proceeding further with the code example, lines 120 to 173 instantiate the `Wavelet_synthesis_ops` class as `IWT_ops` and define an object named as `iwt_ops`. The class `Wavelet_synthesis_ops` abstracts the operations of the IWT. lines 120 to 157 define several lambda functions that implement the required operations for the WT. Descriptions of the lambda functions are given in what follows.

The lambda function defined in line 120 is used to determine the final number of vertices in the mesh after `num_levels` levels of refinement. The returned result is used by the library to allocate adequate memory for internal buffers. The final number can be calculated based on the topologic refinement rule used in the WT, since a topologic refinement rule determines the relationship between the numbers of vertices, edges, and faces before and after the refinement. In the case of PTQ, the built-in function could be employed as shown in line 129.

The lambda functions defined in lines 132 and 136 are used to perform any necessary initialization and cleanup before and after the multi-level IWT computations. Additional resources can be allocated and released in this pair of functions.

Line 140 defines a lambda function where the topologic refinement should be implemented. This function is called iteratively to perform multi-level topologic refinement on the mesh. Since this operation is not trivial, the built-in function for the PTQ case is used. For a custom implementation, the following requirements should be satisfied in order to be incorporated into the WTT library correctly. First, a well-defined topologic refinement rule needs to be implemented. For the vertices introduced by the rule, the parameter `level` should be assigned to their level property. The type and ID of these vertices should also be assigned, depending on the rule. Then, the handles of the introduced vertices should be appended to the array `vertices` and sorted by type and ID in ascending order. Pointers to the end of the new vertex handles of each type in `vertices` should be appended to `bands`.

Line 153 defines a lambda function where the lifting steps of the single-level IWT should be implemented. This function is called iteratively to perform the multi-level IWT. The parameters have the same meaning as that in the `fwt_lift`. Users can use the parameters to navigate to the vertices for which a custom lifting step is designed.

After all the required operations are defined, line 159 instantiates `Wavelet_synthesis_ops` as `IWT_ops` by the declared types of the lambdas. Since the IWT and FWT usually have the same number of vertex types, the `get_num_types` lambda function being used in instantiating `Wavelet_analysis_ops` can be reused here. Then, line 168 constructs the `IWT_ops` object `iwt_ops` by these lambda functions, and each of these lambda functions is wrapped in a corresponding member function of `iwt_ops`.

Usage of `Wavelet_analyze` and `Wavelet_synthesize`

Having defined all the operational classes, we can use them to define the computational classes, `Wavelet_analyze` and `Wavelet_synthesize`. lines 175 and 176 instantiate the computational classes as `FWT` and `IWT` by the defined `Mesh_ops`, `FWT_ops` and `IWT_ops`. lines 177 and 178 constructs the objects `fwt` and `iwt` by `mesh_ops`, `fwt_ops`, and `iwt_ops`. The objects `fwt` and `iwt` can later be used via the function call operator to compute the multi-level FWT and IWT.

Next, lines 180 and 181 defines the input mesh and reads data from standard input to construct the input mesh. line 185 defines a container where the wavelet coefficients obtained from the multi-level FWT are stored. The layout of the wavelet coefficients is given as follows. Since a wavelet coefficient is associated with a vertex, the position of the wavelet coefficient in `coefs` is determined by the level, type, and ID of the original vertex. The wavelet coefficients are sorted by the level and type of the original vertices in ascending order, and each inner array of `coefs` stores the wavelet coefficients with the same type and level. In an inner array of `coefs`, the wavelet coefficients are sorted by the ID of original vertices in ascending order.

Then, line 186 defines the number of levels of the WT as 4. line 188 compute the 4 levels of FWT for the input mesh. Since the FWT could fail, we add a block following line 188 to handle the failure case. For a successful result, the input mesh is coarsened 4 times, and `coefs` is filled with the wavelet coefficients. After that, line 195 computes a 4-level IWT based on the mesh and wavelet coefficients. The mesh is refined 4 times, and the wavelet coefficients are sequentially loaded from the beginning of `coefs` for computations. For a well-defined WT, the original mesh is recovered after `iwt` returns.

3.4.2 Loop and Butterfly WTs Built-in Functions

In case of the Loop and Butterfly WTs, the WTT library provides functions that can be used directly to compute those WTs on a mesh. A function performs either the FWT or IWT in the Loop or Butterfly case. Next, we illustrate the built-in WT functions, starting with the Loop case.

Loop WT Functions

The built-in functions for computing the Loop WT are: 1) `loop_analyze` and 2) `loop_synthesize`. Function `loop_analyze` computes the multi-level Loop FWT on a triangle mesh, and function `loop_synthesize` computes the multi-level Loop IWT. Both the functions have a template parameter that specifies the type of the mesh. The accepted type of the mesh should be inherited from `CGAL::Polyhedron_3`. An example program is shown in Listing 3.2 to demonstrate the usage of the two functions.

Listing 3.2: Example program of computing the Loop FWT and IWT on a triangle mesh.

```
1  #include <wtlib/loop_wavelet_transform.hpp>
2  #include <CGAL/IO/Polyhedron_iostream.h>
3  #include <CGAL/Simple_cartesian.h>
4  #include <CGAL/Polyhedron_3.h>
5
6  using Kernel = CGAL::Simple_cartesian<double>;
7  using Vector3 = Kernel::Vector_3;
8  using Mesh = CGAL::Polyhedron_3<Kernel>;
9
10 int main() {
11   Mesh mesh;
12   if (!std::cin >> mesh) {
13     std::cerr << "Fail to read mesh.\n"
14     return 1;
15   }
16   if (!mesh.is_pure_triangle()) {
17     std::cerr << "The input mesh is not a pure triangle.\n";
18     return 1;
19   }
```

```
20
21    std::vector<std::vector<Vector3>> coefs;
22
23    bool res = wtlib::loop_analyze<Mesh>(mesh, coefs, 3);
24    if (res) {
25      wtlib::loop_synthesize<Mesh>(mesh, coefs, 3);
26    } else {
27      std::cerr << "The input mesh does not have 3 levels of subdivision
            connectivity.\n";
28      return 1;
29    }
30    return 0;
31  }
```

The program loads a mesh from standard input and computes a three level FWT and IWT. A description of the program follows. After including the required headers, some handy type aliases are defined by lines 6 to 8. At the beginning of the main function, lines 11 to 19 constructs a mesh by the data read from standard input and checks if the input mesh is a pure triangle mesh. Then, a multi-dimensional array `coefs` is created by line 21 to buffer the wavelet coefficients obtained from the FWT. After that, line 23 computes a three level FWT and stores the returned status. The mesh is coarsened three times, and the `coefs` is filled with wavelet coefficients. Each of the inner arrays of `coefs` contains the wavelet coefficients at the same level, in ascending order. If the returned status is **`true`**, line 25 computes a three level IWT to recover the original mesh. Otherwise, the program prints an error message to standard error, indicating the input mesh does not have three levels of subdivision connectivity.

Butterfly WT Functions

The built-in functions for the Butterfly WT are: 1) `butterfly_analyze` and 2) `butterfly_synthesize`. Function `butterfly_analyze` computes the multi-level Butterfly FWT on a closed triangle mesh, and `butterfly_synthesize` computes the multi-level Butterfly IWT. Except for the internal computations, the two functions have the same interfaces as `loop_analyze` and `loop_synthesize`, respectively. Besides, since the Butterfly WT is defined on closed triangle meshes, before calling the two functions, we have to check if the input mesh is closed. Otherwise, the behavior is undefined. In Listing 3.3, we present a program to demonstrate the usage of the Butterfly WT functions. The program also computes three levels the FWT and IWT on a triangle mesh read from standard input. The program is the same as that in Listing 3.2, except that lines 21 to 24 are inserted to check if the input mesh is closed.

Listing 3.3: Example program of computing the Butterfly FWT and IWT on a closed triangle mesh.

```
1  #include <wtlib/butterfly_wavelet_transform.hpp>
2  #include <CGAL/IO/Polyhedron_iostream.h>
3  #include <CGAL/Simple_cartesian.h>
4  #include <CGAL/Polyhedron_3.h>
5
6  using Kernel = CGAL::Simple_cartesian<double>;
7  using Vector3 = Kernel::Vector_3;
8  using Mesh = CGAL::Polyhedron_3<Kernel>;
9
10 int main() {
11   Mesh mesh;
12   if (!std::cin >> mesh) {
13     std::cerr << "Fail to read mesh.\n"
14     return 1;
15   }
16   if (!mesh.is_pure_triangle()) {
17     std::cerr << "The input mesh is not a pure triangle.\n";
18     return 1;
19   }
20
```

```
21    if (!mesh.is_closed()) {
22      std::cerr << "The Butterfly wavelet transform only supports closed mesh.\n"
23      return 1;
24    }
25
26    std::vector<std::vector<Vector3>> coefs;
27
28    bool res = wtlib::butterfly_analyze<Mesh>(mesh, coefs, 3);
29    if (res) {
30      wtlib::butterfly_synthesize<Mesh>(mesh, coefs, 3);
31    } else {
32      std::cerr << "The input mesh does not have 3 levels of subdivision
            connectivity.\n";
33      return 1;
34    }
35    return 0;
36  }
```

3.5 Programs

In addition to the library, the WTT includes some wavelet-based application programs that use the library. These application programs perform FWT and IWT computation, and wavelet denoising and compression. In what follows, we describe each application program in turn.

3.5.1 The `wtt_fwt` and `wtt_iwt` Programs

The WTT provides two application programs for computing multi-level Loop and Butterfly WTs: `wtt_fwt` and `wtt_iwt`. The program `wtt_fwt` computes the FWT on a triangle mesh and outputs the coarsened mesh and wavelet coefficients. The program `wtt_iwt` computes the IWT on a triangle mesh with wavelet coefficients and outputs the refined mesh. Detailed descriptions of the two programs are given as below.

The `wtt_fwt` Program

This program reads a triangle mesh from a file or standard input, computes the L-level Loop or Butterfly FWT of the mesh, and outputs the coarsened mesh and wavelet coefficients to files or standard output. Both the input and output mesh are in OFF format [1]. The input mesh is required to have at least L levels of subdivision connectivity. Otherwise, the program exits with an error. In the wavelet coefficient output, each wavelet coefficient (i.e., three real numbers separated by spaces) occupies a line, and a blank line separates wavelet coefficients of two consecutive levels. The synopsis and command-line options of this program are given below.

Synopsis

```
wtt_fwt -l $levels -m $scheme
                    [-c $path] [-i $path] [-o $path]
```

Options

`-l $levels`
This required option sets the number of levels for the FWT to `$levels`. Argument `$levels` accepts a non-negative integer.

`-m $scheme`
This required option sets the type of WT to `$scheme`. Argument `$scheme` accepts `Butterfly` or `Loop`.

`-c $path`
This option sets the file path for the output wavelet coefficients to `$path`. If no file is specified, standard output is used.

`-i $path`
This option sets the file path for the input mesh to `$path`. If no file is specified, standard input is used.

`-o $path`
This option sets the file path for the output mesh to `$path`. If no file is specified, standard output is used.

The `wtt_iwt` Program

This program reads a triangle mesh and wavelet coefficients from files or standard input, computes the L-level Butterfly or Loop IWT, and outputs the refined mesh to a file or standard output. Both the input and output mesh are in OFF format. The requirements on the wavelet coefficient input are as follows. Each wavelet coefficient should occupy a single line and be composed of three real numbers separated by spaces. A blank line should separate wavelet coefficients in two consecutive levels. Otherwise, the behavior is undefined. It is recommended to use the wavelet coefficients obtained from `wtt_fwt`, since the order of wavelet coefficients is managed internally. If the input mesh and wavelet coefficients are from `wtt_fwt`, the original mesh is guaranteed to be recoverable. The synopsis and command-line options of this program are given below.

Synopsis

```
wtt_iwt -l <levels> -m $scheme [-A]
                       [-c $path] [-i $path] [-o $path]
```

Options

`-l $levels`
This required option sets the number of levels for the IWT to `$levels`. Argument `$levels` accepts a non-negative integer.

`-m $scheme`
This required option sets the type of WT to `$scheme`. Argument `$scheme` accepts `Butterfly` or `Loop`.

`-A`
This option enables auto-adjusting wavelet coefficients. Without this option, the program checks if the number of wavelet coefficients matches the required number. If the check fails, the program exits with an error. With this option, the program pads missing wavelet coefficients with zeros or truncates the extra wavelet coefficients.

`-c $path`
This option sets the file path for the input wavelet coefficients to `$path`. If no file is specified, standard input is used.

`-i $path`
This option sets the file path for the input mesh to `$path`. If no file is specified, standard input is used.

`-o $path`
This option sets the file path for the output mesh to `$path`. If no file is specified, standard output is used.

Examples

A few examples of using the above application programs are provided in what follows.

EXAMPLE 3.1 Given a triangle mesh file `bunny.off`, shipped with WTT, users can compute the 3-level Butterfly FWT and dump the coarsened mesh to file `bunny_fwt.off` and wavelet coefficient to file `bunny.coefs`, with the command:

```
wtt_fwt -l 3 -m Butterfly -c bunny.coefs \
                    -i bunny.off -o bunny_fwt.off
```

■

EXAMPLE 3.2 Given a triangle mesh file `bunny_fwt.off` and wavelet coefficient file `bunny.coefs` obtained from Example 3.1, users can compute the 3-level Butterfly IWT and output the refined mesh to file `bunny_iwt.off`, with the command:

```
wtt_iwt -l 3 -m Butterfly -c bunny.coefs \
                        -i bunny_fwt.off -o bunny_iwt.off
```

■

EXAMPLE 3.3 Given a triangle mesh file `bunny.off`, users can compute the 2-level Loop IWT with all-zero wavelet coefficients and output the refined mesh to file `bunny_iwt_zero.off`, with the command:

```
wtt_iwt -l 3 -m Butterfly -c /dev/null -A \
                        -i bunny_fwt.off -o bunny_iwt.off
```

■

3.5.2 Wavelet Denoising and Compression

In this section, we introduce another two application programs: `wtt_filter` and `wtt_demo`, which are developed for wavelet denoising and compression. Descriptions of the two programs are given below.

The `wtt_filter` Program

This program demonstrates Butterfly and Loop wavelet compression on a triangle mesh. It loads a mesh in OFF format, applies multi-level Butterfly or Loop FWT to the mesh, filters the wavelet coefficients, and performs the same levels of the IWT to construct the output mesh. Filtering is realized as setting some of the wavelet coefficients to zero. Different filtering schemes can be selected via the command-line options. The synopsis and command-line options of this program are given below.

Synopsis

```
wtt_filter -l $levels -m $scheme [-i $path] [-o $path]
           (-L $level | -c $percentage | -t $threshold)
```

Options

`-l $levels`
This required option sets the number of levels of for the FWT and IWT to `$levels`. Argument `$levels` accepts a non-negative integer.

`-m $scheme`
This required option sets the type of WT to `$scheme`. Argument `$scheme` accepts a string literal `Butterfly` or `Loop`.

`-i $path`
This option sets the file path for the input mesh to `$path`. If no file is specified, standard input is used.

`-o $path`
This option sets the file path for the output mesh to `$path`. If no file is specified, standard output is used.

`-L $level`
This option sets the filtering scheme to lowpass, which sets the wavelet coefficients whose level is higher than the given `$level` to zero.

`-c $percentage`
This option sets the filtering scheme to compression. A percentage `$percentage` of wavelet coefficients with larger magnitudes are preserved, and the other wavelet coefficients are set to zero. Argument `$percentage` accepts a real number within the range 0 to 100.

`-t $threshold`
This option sets the filtering scheme to hard-thresholding. The wavelet coefficients whose magnitudes are smaller than the threshold `$threshold` are set to zero. Argument `$threshold` accepts a real number.

Example of Usage

An example of using the `wtt_filter` program for compression is given in what follows.

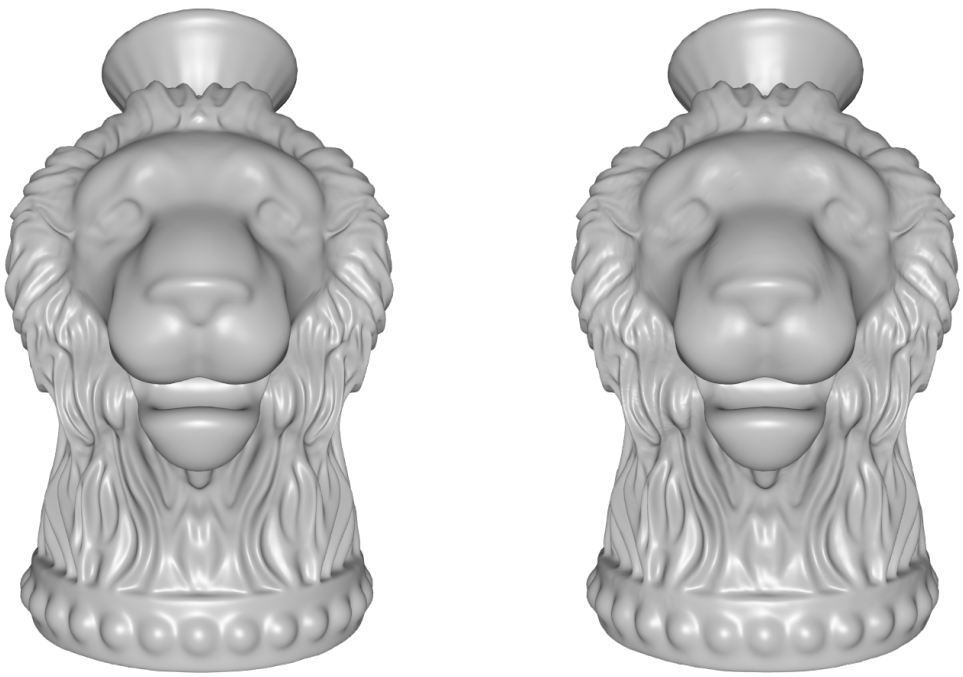
EXAMPLE 3.4 Given a triangle mesh file `vase.off`, one can compress the wavelet coefficients to 5% with a 3-level Butterfly WT by using the command:

```
wtt_wavelet_filter -l 3 -m Butterfly -c 5 \
                   -i vase.off -o vase_from_compression.off
```

In this example, 183860 out of 193536 wavelet coefficients are set to zero. The input and output meshes of the example are shown in Figure 3.1(a) and (b), respectively. ■

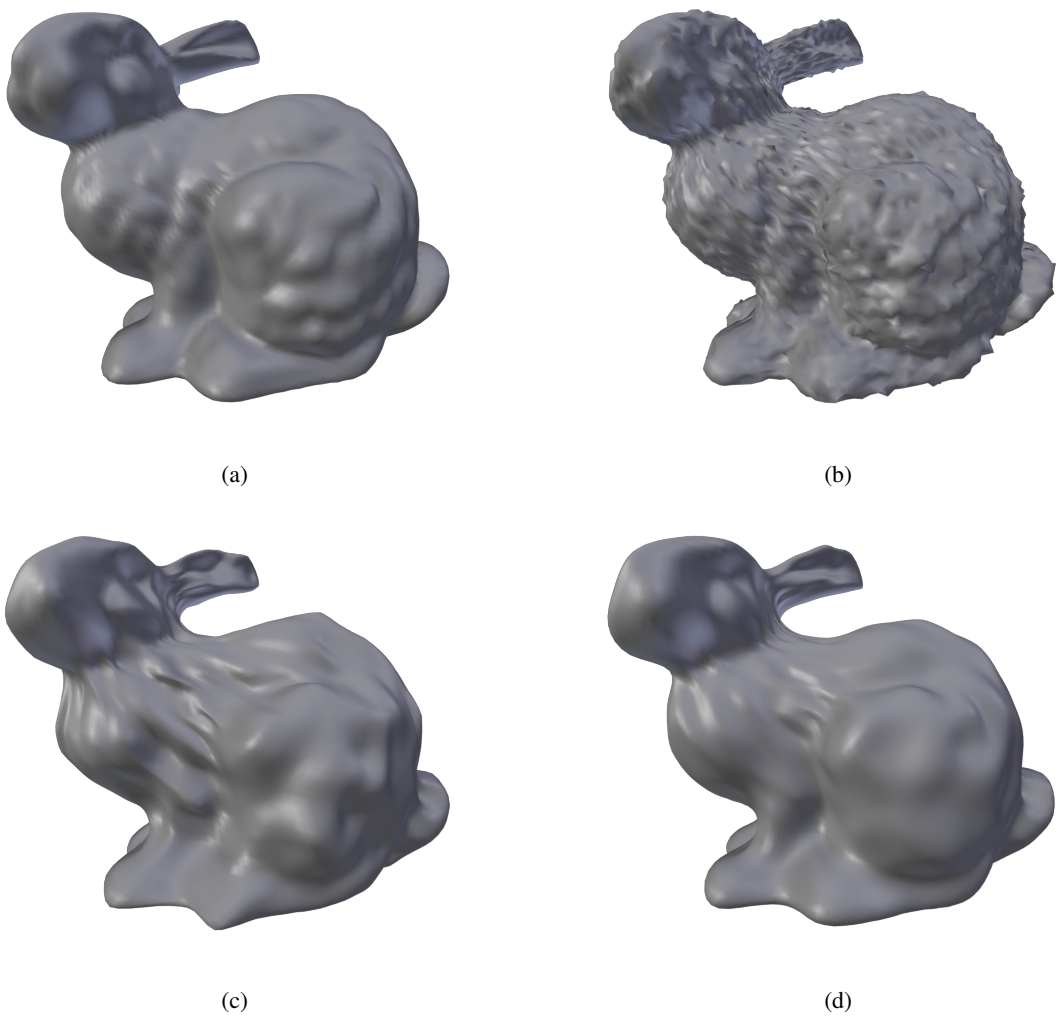
The `wtt_demo` Script

This script demonstrates Butterfly and Loop wavelet denoising by running the `wtt_filter` program on two example meshes: a dragon and a bunny. The original mesh, the noisy mesh, and the denoised mesh by Butterfly and Loop are rendered by MeshLab [20]. Since this script relies on MeshLab for rendering, users should have MeshLab installed beforehand. Figures 3.2 and 3.3 show some screenshots captured from running `wtt_demo`. The original mesh and obtained denoised mesh are shown in Figures 3.2 and 3.3, for the cases of the bunny and dragon meshes respectively.



(a) (b)

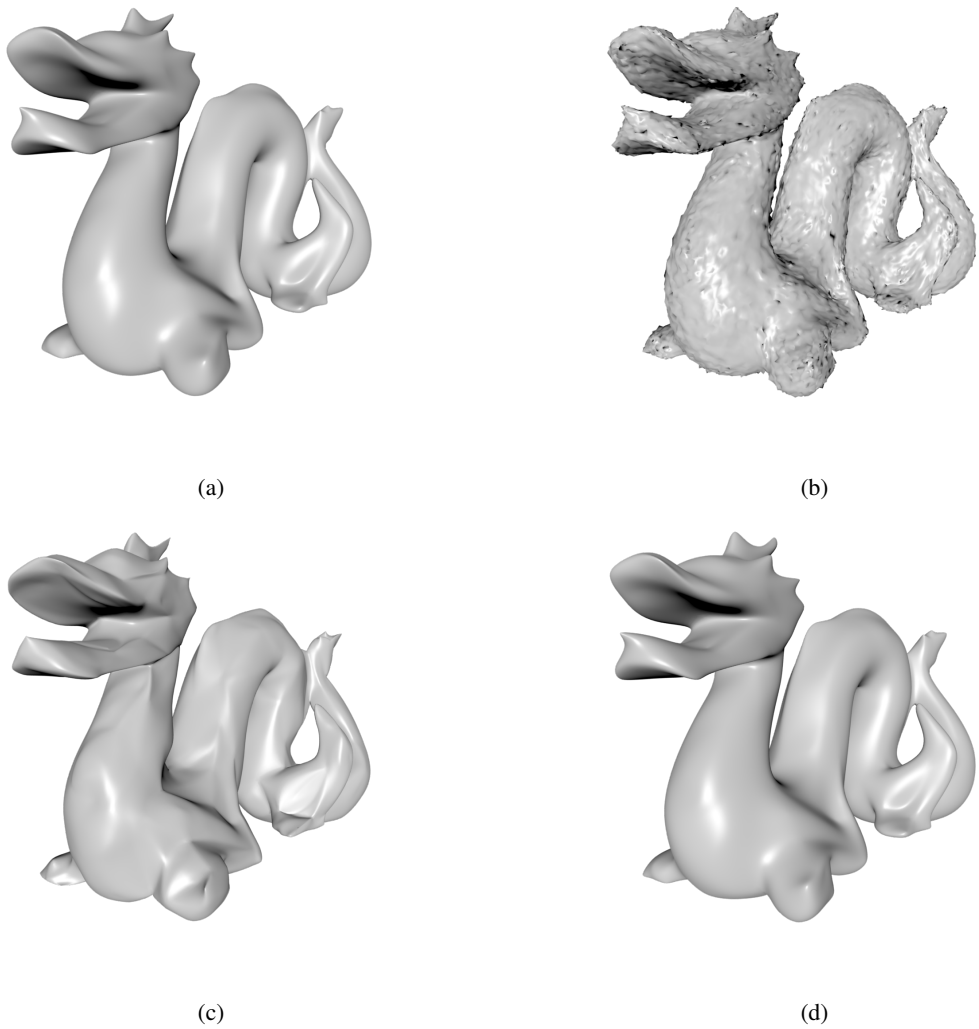
Figure 3.1: Example of 3-level Butterfly wavelet compression with compression rate 5%. (a) The original mesh (`vase.off`). (c) The output mesh (`vase_from_compression.off`)



(a) (b)

(c) (d)

Figure 3.2: Screenshots obtained from running `wtl_demo`. The (a) original, (b) noisy, (c) the Butterfly denoised, and (d) the Loop denoised `bunny`.



(a) (b)

(c) (d)

Figure 3.3: Screenshots obtained from running `wtl_demo`. The (a) original, (b) noisy, (c) the Butterfly denoised, and (d) the Loop denoised `dragon`.

Chapter 4

Results and Analysis

4.1 Introduction

In this chapter, we present some results obtained with the applications in the WTT. The application results considered Butterfly and Loop wavelet denoising and compression on 3-D triangle meshes. Also, the run-time performance of the WTT library is evaluated by measuring execution time and memory consumption, and the performance bottlenecks are analyzed. This chapter starts with a description of the test datasets used in our experiments.

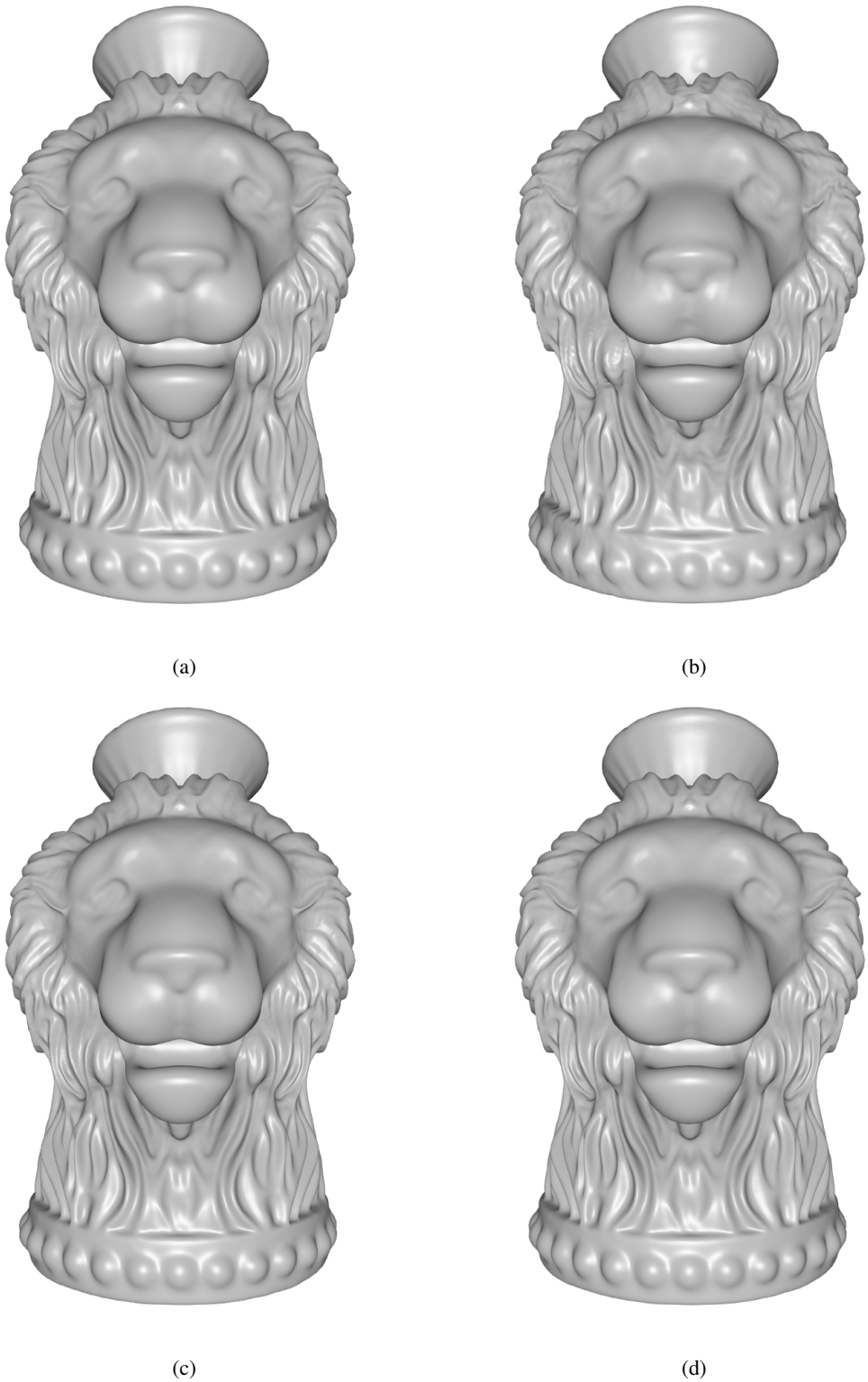
4.2 Datasets

In Table 4.1, some basic information about each of the trianlge meshes used in our experiments is given, including the number of vertices and faces, whether the mesh is closed, and the source of the mesh. These meshes have been widely used in the research literature. Since the FWT requires subdivision connectivity, all the test meshes have been subdivided or remeshed as needed in order to ensure at least one level of subdivision connectivity.

4.3 Experimental Results

In this section, we present the experimental results for Butterfly and Loop wavelet compression and denoising. They are both implemented by computing the FWT, adjusting the wavelet coefficients, and computing the IWT. Wavelet compression preserves a given percentage of wavelet coefficients with larger magnitudes and sets the remaining coefficients to zero. Wavelet denoising sets wavelet coefficients at one or more resolution levels to zero.

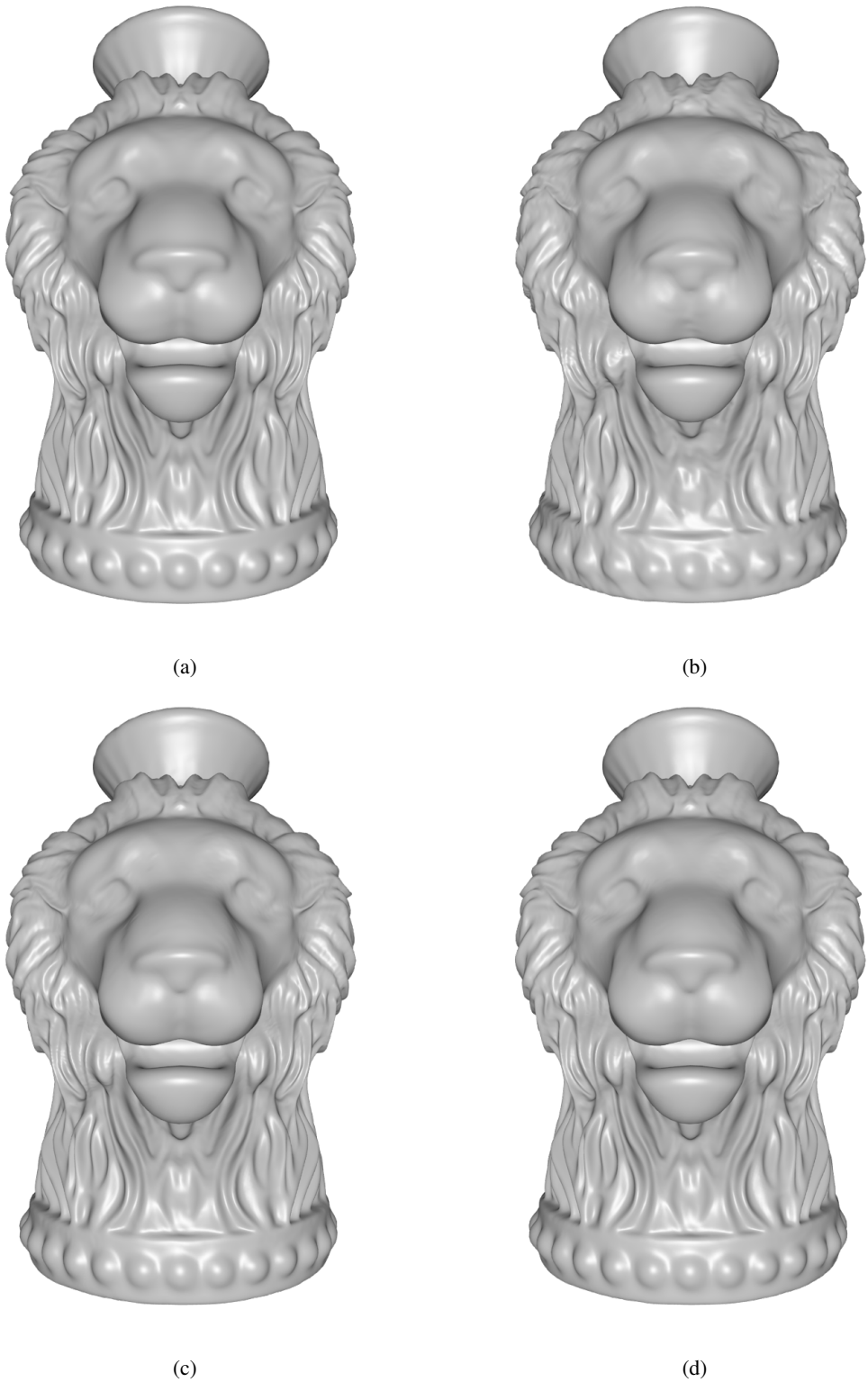
Examples of Loop and Butterfly wavelet compression are respectively shown in Figure 4.1 and 4.2. In the experiment, the 2-level Loop and Butterfly FWT were computed on the mesh `lion-vase` first. Then, the 2-level Loop and Butterfly IWT were computed with the wavelet coefficients whose magnitudes are in the top 1%, 5%, and 10% and other zero wavelet coefficients, respectively. The sub-figures of Figures 4.1 and 4.2 present (a) the original mesh, and the Loop and Butterfly reconstructed meshes with a wavelet coefficient compression rate of (b) 1%, (c) 5%, and (d) 10%. Examples of wavelet denoising are provided in Figures 4.3 and 4.4. We first introduce noise to two meshes, a bunny and a dragon, as shown in Figures 4.3(a) and (b) and 4.4(a) and (b). Then we apply 3-level Loop and Butterfly wavelet denoising to the noisy meshes. For the bunny, the wavelet coefficients above level 1 are discarded (i.e., set to zero) in order to remove noise. The denoised bunny obtained in the Butterfly and Loop cases are shown in Figures 4.3(c) and (d). For the dragon, all the wavelet coefficients are discarded. The denoised dragon obtained in the two cases are shown in Figures 4.4 (c) and (d).



(a) (b)

(c) (d)

Figure 4.1: Example of 2-level Loop wavelet compression. (a) The original mesh. (b) Reconstructed mesh by using 1843 wavelet coefficients whose magnitudes are in the top 1%. (c) Reconstructed mesh by using 9216 wavelet coefficients whose magnitudes are in the top 5%. (d) Reconstructed mesh by using 18432 wavelet coefficients whose magnitudes are in the top 10%.



(a) (b)

(c) (d)

Figure 4.2: Example of 2-level Butterfly wavelet compression. (a) The original mesh. (b) Reconstructed mesh by using 1843 wavelet coefficients whose magnitudes are in the top 1%. (c) Reconstructed mesh by using 9216 wavelet coefficients whose magnitudes are in the top 5%. (d) Reconstructed mesh by using 18432 wavelet coefficients whose magnitudes are in the top 10%.

Table 4.1: The test meshes and their characteristics

Name	Vertices	Faces	Closed	Source
torus	12288	24576	yes	[4]
bunny	16386	32768	yes	[21]
bulb	28162	56320	yes	[8]
dragon	32000	64000	yes	[2]
torusknot	40960	81920	yes	[4]
kid	49154	98304	yes	[12]
gargoyle	65538	131072	yes	[21]
horse	65538	131072	yes	[21]
tyra	98306	196608	yes	[21]
hand	188241	376320	no	[27]
vase	196610	393216	yes	[12]
armardillo	262146	524288	yes	[21]
cow	393218	786432	yes	[21]
venus	1048578	2097152	yes	[21]

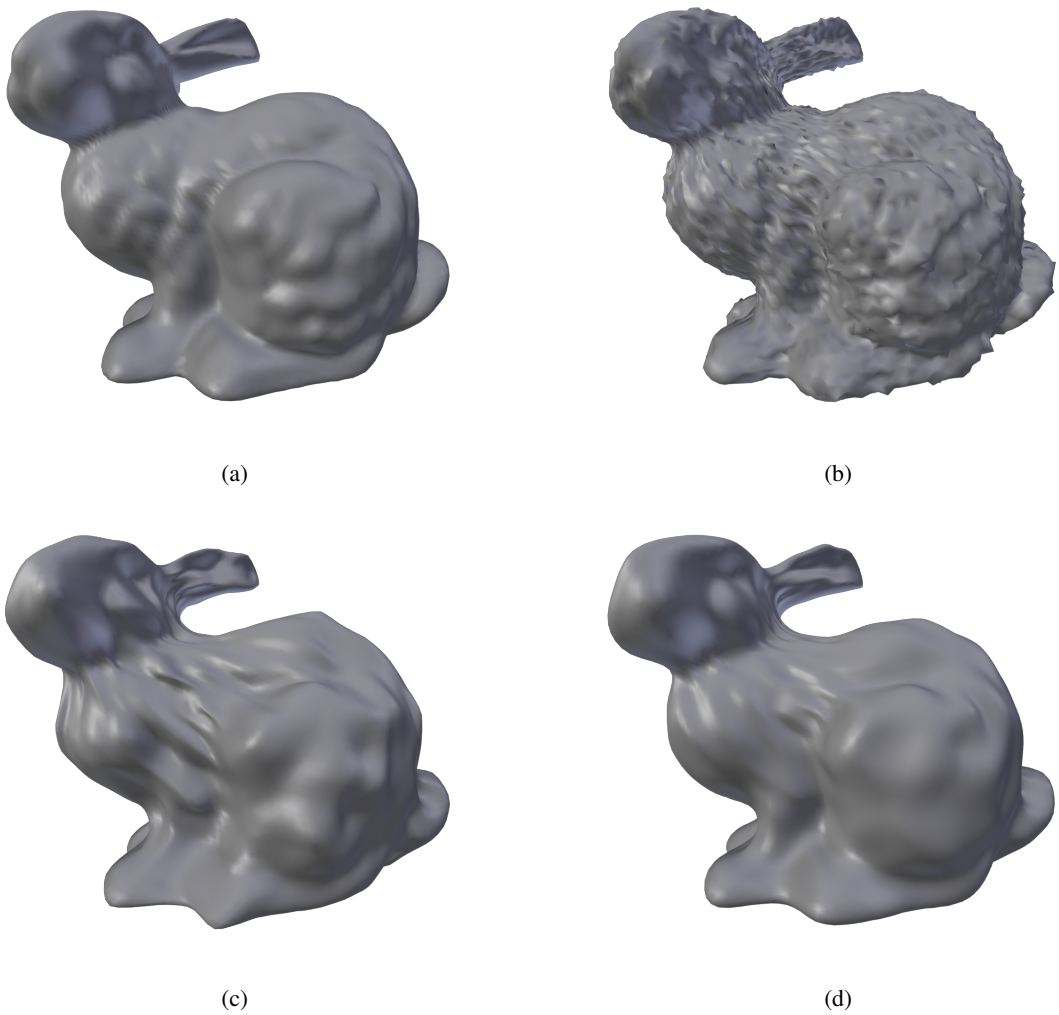
4.4 Run-Time Performance

Next, we consider the run-time performance, which is evaluated by measuring the execution time and memory consumption of the functions for computing FWT and IWT for the Loop and Butterfly cases. Before proceeding further, we briefly introduce the hardware that was employed to conduct the experiments. The experimental results were collected on a computer with a 3.7 GHz Intel Core i7-8700k CPU and 32 GB of RAM. All the code was compiled with full optimization enabled. In what follows, we consider the execution time for computing the FWT and IWT.

4.4.1 Analysis of Execution Time

The analysis of execution time consists of two parts. First, we present the execution time of the FWT and IWT function calls on the 14 meshes, which covers the case of tens of thousands to millions of vertices. It provides a general relationship between the number of vertices and the execution time of the FWT and IWT. Second, we present the profiling of the functions that internally implement the FWT and IWT, which reveals the execution time percentage of each internal function and the bottlenecks of the FWT and IWT. Both the Loop and Butterfly FWT and IWT are evaluated in our experiments. For each of the 14 meshes in the datasets, we run the `wtt_fwt` and `wtt_iwt` programs 50 times to gather run-time data. In each run, the one-level FWT followed by the one-level IWT are computed. The `gprof` [10] tool was employed to perform the analysis. Although `gprof` is primarily used for profiling instead of measuring execution time, we still use the execution time results obtained from `gprof` to ensure the data consistency in the two parts. The median execution time of the Loop and Butterfly FWT and IWT are given in Table 4.2. Since the mesh `hand` is a mesh with boundaries, where the Butterfly WT is not supported, the execution time is not available.

To begin, we explore the relationship between the execution time and the number of vertices. We expect the execution time increases with the number of vertices, and two meshes with similar vertices should consume a similar time. Since the FWT includes a non-trivial vertex classification step, the execution time of the FWT is expected to be higher than the IWT. Furthermore, the classification step is implemented by Taubin's subdivision detection with a linear-time complexity [26] and sorting with a time complexity of $O(n \log n)$. Thus, the execution time of the FWT



(a) (b)

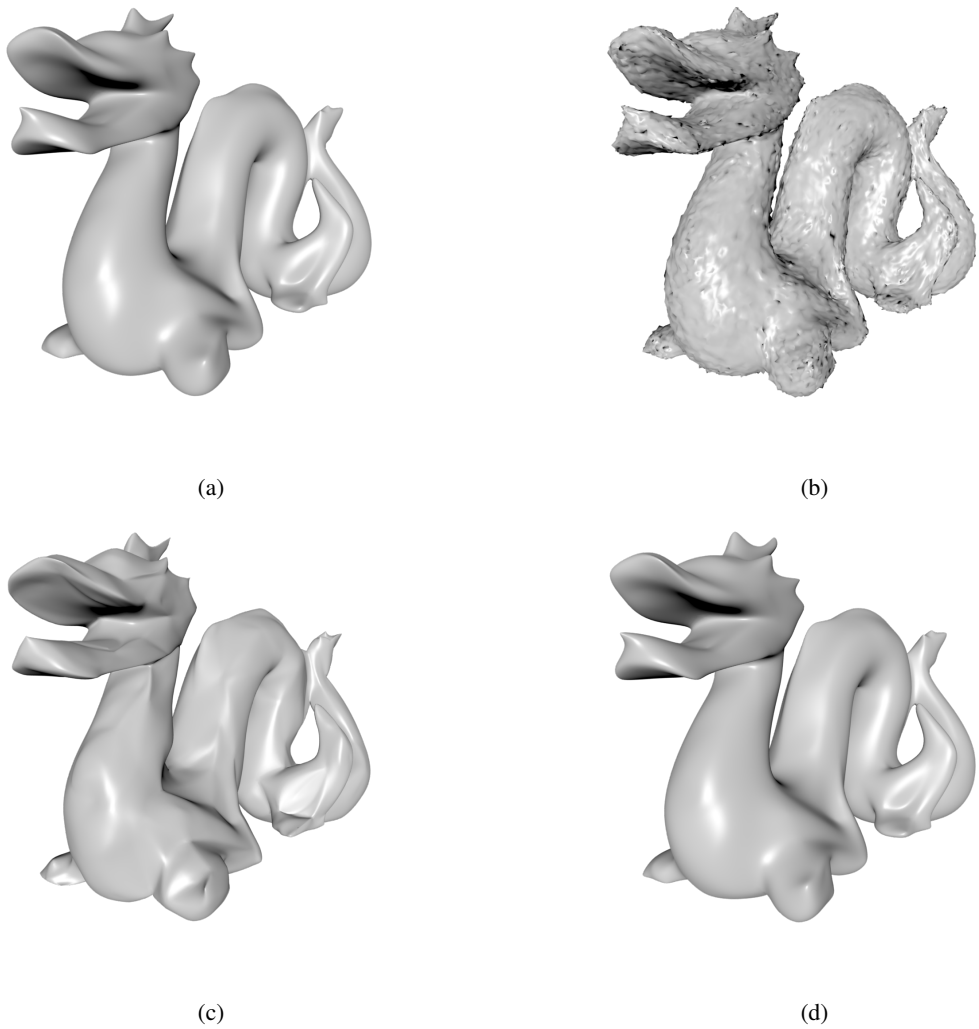
(c) (d)

Figure 4.3: Example of 3-level wavelet denoising on mesh `bunny`. The (a) original, (b) noisy, (c) the Butterfly denoised, and (d) the Loop denoised `bunny`.

is expected to be always higher than the IWT. In comparing the Loop and Butterfly, since the lifting steps of Loop consist of solving a linear system, the FWT and IWT of Loop are expected to consume more time than Butterfly. Our expectations are confirmed by the execution time results in Table 4.2. Observe that meshes `horse` and `gargoyle` with the same number of vertices have different execution times. The reason is that it is impossible to reproduce the same environment (e.g., CPU loads) to measure the execution time on different meshes. Also, since the sampling interval of `gprof` is 0.01 seconds, the difference within 10 milliseconds is unreliable. As their execution time difference is within 10 milliseconds, we consider they have the same execution time.

Next, we proceed to profile the functions that internally implement the Butterfly and Loop transform to analyze the bottlenecks. Tables 4.3, 4.4, 4.5, and 4.6 list the execution time of these functions and their percentages of the total FWT and IWT time. Observe that function `classify` occupies the majority of the execution time of the FWT in both Loop and Butterfly. The reason is not only sorting but also Taubin's algorithm. Although Taubin's algorithm has a linear time complexity with respect to the number of vertices, it requires complex data structures and operations to implement, which is time-consuming.

In what follows, we consider the execution time of function `lift`, which implements the lifting computations. The execution time should be linear to the number of vertices. The experimental results in Tables 4.3, 4.4, 4.5, and 4.6



(a) (b)

(c) (d)

Figure 4.4: Example of 3-level wavelet denoising on mesh `dragon`. The (a) original, (b) noisy, (c) the Butterfly denoised, and (d) the Loop denoised `dragon`.

Table 4.2: The execution time (in milliseconds) of Butterfly and Loop FWT and IWT.

Name	Vertices	Butterfly		Loop	
		FWT (milliseconds)	IWT (milliseconds)	FWT (milliseconds)	IWT (milliseconds)
torus	12288	13.40	8.00	14.40	8.00
bunny	24578	31.20	14.90	34.00	14.70
bulb	28162	52.40	23.60	58.00	30.00
dragon	32000	64.00	30.60	77.10	33.20
torusknot	40960	90.00	35.80	103.00	45.10
kid	49154	98.20	45.80	108.20	48.50
horse	65538	154.41	69.00	161.61	76.70
gargoyle	65538	156.71	70.00	167.51	74.80
tyra	98306	241.41	113.00	254.31	120.81
hand	188241	N/A*	N/A*	391.42	187.31
vase	196610	503.52	242.21	521.02	254.51
armardillo	262146	701.13	344.22	746.93	363.02
cow	393218	1080.00	509.12	1130.00	534.72
venus	1048578	3120.00	1420.00	3250.00	1500.0

* This result is unavailable as the Butterfly wavelet transform does not support meshes with boundaries.

Table 4.3: The execution time and percentages of functions that internally implement the Butterfly FWT.

Name	Vertices	get_num_types	classify	initialize	lift	coarsen	cleanup
torus	12288	0.00 (0.00%)	7.95 (59.36%)	0.00 (0.00%)	3.02 (22.44%)	1.62 (12.08%)	0.00 (0.00%)
bunny	24578	0.00 (0.00%)	17.84 (57.20%)	0.00 (0.00%)	7.32 (23.51%)	3.36 (10.83%)	0.00 (0.00%)
bulb	28162	0.00 (0.00%)	27.92 (53.28%)	0.00 (0.00%)	17.63 (33.72%)	5.64 (10.81%)	0.00 (0.00%)
dragon	32000	0.00 (0.00%)	36.25 (56.71%)	0.00 (0.00%)	19.82 (30.90%)	7.62 (11.86%)	0.00 (0.00%)
torusknot	40960	0.00 (0.00%)	49.69 (55.24%)	0.00 (0.00%)	26.10 (28.98%)	10.74 (11.90%)	0.00 (0.00%)
kid	49154	0.00 (0.00%)	55.50 (56.54%)	0.00 (0.00%)	33.24 (33.80%)	9.05 (9.15%)	0.00 (0.00%)
gargoyle	65538	0.00 (0.00%)	83.07 (53.08%)	0.00 (0.00%)	48.38 (30.90%)	18.84 (12.05%)	0.00 (0.00%)
horse	65538	0.00 (0.00%)	81.74 (52.96%)	0.00 (0.00%)	47.56 (30.90%)	16.61 (10.72%)	0.00 (0.00%)
tyra	98306	0.00 (0.00%)	131.87 (54.58%)	0.00 (0.00%)	79.75 (32.95%)	25.99 (10.81%)	0.00 (0.00%)
vase	196610	0.00 (0.00%)	264.94 (52.62%)	0.00 (0.00%)	163.89 (32.63%)	58.86 (11.75%)	0.00 (0.00%)
armardillo	262146	0.00 (0.00%)	379.56 (54.10%)	0.00 (0.00%)	222.93 (31.79%)	79.43 (11.28%)	0.00 (0.00%)
cow	393218	0.00 (0.00%)	590.00 (54.42%)	0.00 (0.00%)	340.00 (31.31%)	120.00 (11.24%)	0.00 (0.00%)
venus	1048578	0.00 (0.00%)	1780.00 (56.98%)	0.00 (0.00%)	930.00 (29.70%)	330.00 (10.53%)	0.00 (0.00%)

Table 4.4: The execution time and percentages of functions that internally implement the Butterfly IWT.

Name	Vertices	get_num_types	get_mesh_size	initialize	refine	lift	cleanup
torus	12288	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	2.19 (27.44%)	5.61 (70.23%)	0.00 (0.00%)
bunny	24578	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	4.96 (33.26%)	9.62 (64.44%)	0.00 (0.00%)
bulb	28162	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	5.83 (24.75%)	17.02 (72.03%)	0.00 (0.00%)
dragon	32000	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	9.24 (30.13%)	19.66 (64.14%)	0.00 (0.00%)
torusknot	40960	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	10.77 (30.11%)	24.43 (68.21%)	0.00 (0.00%)
kid	49154	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	13.09 (28.69%)	30.66 (66.93%)	0.00 (0.00%)
gargoyle	65538	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	22.73 (32.48%)	45.67 (65.35%)	0.00 (0.00%)
horse	65538	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	22.66 (32.86%)	44.37 (64.11%)	0.00 (0.00%)
tyra	98306	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	32.83 (28.98%)	78.48 (69.39%)	0.00 (0.00%)
vase	196610	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	73.26 (30.23%)	160.79 (66.47%)	0.00 (0.00%)
armardillo	262146	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	112.14 (32.68%)	223.57 (64.97%)	0.00 (0.00%)
cow	393218	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	155.86 (30.55%)	336.81 (66.19%)	0.00 (0.00%)
venus	1048578	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	460.00 (32.50%)	930.00 (65.21%)	0.00 (0.00%)

Table 4.5: The execution time and percentages of functions that internally implement the Loop FWT.

Name	Vertices	get_num_types	classify	initialize	lift	coarsen	cleanup
torus	12288	0.00 (0.00%)	6.59 (45.81%)	0.00 (0.00%)	4.60 (31.94%)	1.61 (11.13%)	0.00 (0.00%)
bunny	24578	0.00 (0.00%)	16.67 (48.98%)	0.00 (0.00%)	9.10 (26.47%)	4.23 (12.41%)	0.00 (0.00%)
bulb	28162	0.00 (0.00%)	25.97 (44.83%)	0.00 (0.00%)	22.40 (38.61%)	6.23 (10.71%)	0.00 (0.00%)
dragon	32000	0.00 (0.00%)	39.44 (51.16%)	0.00 (0.00%)	27.58 (35.82%)	9.46 (12.24%)	0.00 (0.00%)
torusknot	40960	0.00 (0.00%)	51.67 (50.12%)	0.00 (0.00%)	34.38 (33.42%)	11.34 (11.06%)	0.00 (0.00%)
kid	49154	0.00 (0.00%)	56.01 (51.80%)	0.00 (0.00%)	37.62 (34.78%)	13.80 (12.80%)	0.00 (0.00%)
gargoyle	65538	0.00 (0.00%)	88.61 (52.87%)	0.00 (0.00%)	57.40 (34.25%)	17.39 (10.38%)	0.00 (0.00%)
horse	65538	0.00 (0.00%)	80.97 (50.06%)	0.00 (0.00%)	57.60 (35.60%)	16.44 (10.19%)	0.00 (0.00%)
tyra	98306	0.00 (0.00%)	126.17 (49.56%)	0.00 (0.00%)	91.00 (35.74%)	27.53 (10.77%)	0.00 (0.00%)
hand	188241	0.00 (0.00%)	228.15 (58.31%)	0.00 (0.00%)	107.40 (27.37%)	39.66 (10.10%)	0.00 (0.00%)
vase	196610	0.00 (0.00%)	265.85 (51.01%)	0.00 (0.00%)	184.32 (35.39%)	57.36 (10.96%)	0.00 (0.00%)
armardillo	262146	0.00 (0.00%)	390.36 (52.28%)	0.00 (0.00%)	258.41 (34.56%)	78.55 (10.51%)	0.00 (0.00%)
cow	393218	0.00 (0.00%)	600.00 (52.92%)	0.00 (0.00%)	390.00 (33.88%)	120.00 (10.66%)	0.00 (0.00%)
venus	1048578	0.00 (0.00%)	1790.00 (55.29%)	0.00 (0.00%)	1060.00 (32.24%)	330.00 (10.20%)	0.00 (0.00%)

Table 4.6: The execution time and percentages of functions that internally implement the Loop IWT.

Name	Vertices	get_num_types	get_mesh_size	initialize	refine	lift	cleanup
torus	12288	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	2.80 (35.07%)	5.02 (62.62%)	0.00 (0.00%)
bunny	24578	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	5.20 (35.40%)	9.48 (64.80%)	0.00 (0.00%)
bulb	28162	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	8.80 (29.36%)	19.98 (66.55%)	0.00 (0.00%)
dragon	32000	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	7.30 (22.08%)	24.20 (73.02%)	0.00 (0.00%)
torusknot	40960	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	15.78 (35.05%)	29.28 (65.30%)	0.00 (0.00%)
kid	49154	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	15.05 (31.11%)	30.80 (63.55%)	0.00 (0.00%)
gargoyle	65538	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	22.50 (30.02%)	50.88 (67.88%)	0.00 (0.00%)
horse	65538	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	23.05 (30.05%)	52.02 (67.94%)	0.00 (0.00%)
tyra	98306	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	33.71 (27.90%)	84.40 (69.94%)	0.00 (0.00%)
hand	188241	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	76.20 (40.72%)	106.92 (56.89%)	0.00 (0.00%)
vase	196610	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	74.85 (29.42%)	175.82 (69.23%)	0.00 (0.00%)
armardillo	262146	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	108.08 (29.71%)	244.51 (67.43%)	0.00 (0.00%)
cow	393218	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	163.48 (30.53%)	363.73 (68.10%)	0.00 (0.00%)
venus	1048578	0.00 (0.00%)	0.00 (0.00%)	0.00 (0.00%)	480.00 (31.76%)	1000.00 (66.19%)	0.00 (0.00%)

Table 4.7: First-level (L1) and last-level (LL) cache misses in function lift of the Loop wavelet transform.

Name	Vertices	FWT		IWT	
		L1 Miss (rate)	LL Miss (rate)	L1 Miss (rate)	LL Miss (rate)
hand	188241	5991053 (0.19%)	4822251 (0.15%)	5956614 (0.19%)	4809613 (0.15%)
vase	196610	8793885 (0.27%)	8230943 (0.25%)	8608764 (0.26%)	8123280 (0.25%)

Table 4.8: Comparison of cache misses and execution times (in milliseconds) of function lift for hand and hand_rnd.

Name	FWT			IWT		
	L1 misses (rate)	LL misses (rate)	time	L1 misses (rate)	LL misses (rate)	time
hand	5991053 (0.19%)	4822251 (0.15%)	107.40	5956614 (0.19%)	4809613 (0.15%)	106.92
hand_rnd	7151901 (0.23%)	6806742 (0.22%)	146.70	7169598 (0.23%)	6817019 (0.22%)	144.70

demonstrate that lift exhibits approximately linear-time behavior for all of the meshes in our test set except hand. In this exceptional case, observe that the number of vertices in hand is close to that of vase, but the lifting computation time for hand is nearly half of that of vase. The reason could be that the cache misses are fewer in computing hand than in computing vase. To show that this is a factor, we employ the tool callgrind [28] to collect statistics on cache misses and the cache miss rate, including the first-level (L1) cache misses and last-level (LL) cache misses. The results are given in Table 4.7. The cache miss rate of hand is shown to be somewhat less than vase. Furthermore, we suspect the cache performance is affected by the layout of vertices (i.e., the order of vertices in memory). Although function classify performs sorting, it sorts the vertex handles instead of vertices themselves, so the original layout is unmodified. To confirm our suspicion, we randomized the vertices of hand and reran the experiments. The comparison of cache misses and execution times for the function lift for the original and randomized cases are shown in Table 4.8. From the results, we can see changing the original vertex layout affects cache misses and execution time significantly.

4.4.2 Analysis of Memory Usage

Having analyzed the time complexity of the library, we now consider the memory complexity, which is evaluated by measuring the memory required to compute the FWT and IWT. We primarily focus on the peak memory usage,

Table 4.9: Memory usages of each vertex, halfedge, and face of the (a) mesh and (b) MCDS.

(a) Mesh

Primitives	Sizes (bytes)
vertex (B_v)	80
halfedge (B_h)	56
face (B_f)	56

(b) MCDS

Primitives	Sizes (bytes)
vertex (B'_v)	40
halfedge (B'_h)	56
face (B'_f)	32

since it determines the memory requirement of running our software. For each of the test meshes, we run the `wtt_fwt` program to compute the FWT, then run the `wtt_iwt` program based on the result of the FWT to compute the IWT, and record the peak memory usage of both programs. In addition, we first estimate the peak memory usage by analyzing the size of the main data structures used in the implementation.

We start by the FWT. From our knowledge of the implementation, the main data structures used in the FWT are the input mesh and a mesh-connectivity data structure (MCDS) that mirrors the topology of the mesh for implementing vertex classification. The MCDS is designed to avoid breaking the original mesh's topology while identifying subdivision connectivity and classifying vertices. Thus, the main memory usage would be the sum of the sizes of the mesh and the MCDS. Both the mesh and the MCDS are extensions of `CGAL::Polyhedron_3`, which consist of vertices, halfedges, and faces. In a general case, we assume the sizes of a vertex, a halfedge, and a face, in the mesh, are B_v, B_h, and B_f bytes; the sizes of a vertex, a halfedge, and a face, in the MCDS are B'_v, B'_h, and B'_f bytes, respectively. For a triangle mesh with V vertices, H halfedges, and F triangle faces, where the MDCS also has V vertices, H halfedges, and F faces, the estimated peak memory usage can be denoted by

$$(B_vV + B_hH + B_fF) + (B'_vV + B'_hE + B'_fF).$$

According to the Euler's formula, the relationship between the number of halfedges and faces and the number of vertices can be given by

$$H \approx 6V \text{ and } F \approx 2V.$$

Thus, the peak memory usage represented with the number of vertices can be approximately given by

$$(B_v + 6B_h + 2B_f + B'_v + 6B'_h + 2B'_f)V. \tag{4.1}$$

Since the same implementation of the mesh and MCDS is used in the Loop and Butterfly FWT, the equation above applies for both cases. The actual sizes of each vertex, halfedge, and faces of the mesh and MCDS depend on the compiler/machine and the customization of `CGAL::Polyhedron_3`. On the experiment machine, these sizes are listed in Tables 4.9(a) and (b). By populating the statistics into Equation 4.1, the actual peak memory usage (in bytes) represented by the number of vertices is given by

$$(80 + 6 \times 56 + 2 \times 56 + 40 + 6 \times 56 + 2 \times 32)V = 968V.$$

For the IWT, we apply a similar analysis. The main data structure used in the IWT is the mesh itself only. A mesh will be refined in the IWT, which requests memory for the introduced vertices, halfedges, and faces. Thus, the peak memory usage would be the size of the mesh after refinement. Since, in our experiments, the input of the IWT is the output of the FWT for each of the test meshes, the original mesh is recovered. The peak memory usage is expected to be the size of the original mesh. We use the same notations as in the FWT for the sizes of a vertex, a halfedge, and a face of the mesh to analyze the peak memory usage in general. For a mesh with V vertices, H halfedges, and F triangles, by combining with the Euler's formula, the estimated peak memory usage (in bytes) is given by

$$B_vV + B_hH + B_fF \approx (B_v + 6B_h + 2B_f)V. \tag{4.2}$$

Table 4.10: The peak memory usage and bytes per vertex in the FWT and IWT.

Name	Vertices	FWT (bytes)	bytes/vertex	IWT (bytes)	bytes/vertex
torus	12288	13842552	1126.5	8362232	680.5
bunny	24578	27605512	1123.2	16644504	677.2
bulb	28162	31619624	1122.8	18945416	672.7
dragon	32000	35920024	1122.5	21410552	669.1
torusknot	40960	45955272	1122.0	27162856	663.2
spot	47618	53410344	1121.6	32222648	676.7
kid	49154	55130664	1121.6	33208760	675.6
horse	65538	73480744	1121.2	43727288	667.2
gargoyle	65538	73480744	1121.2	43727288	667.2
tyra	98306	110181048	1120.8	66337176	674.8
hand	188241	210852984	1120.1	127193984	675.7
vase	196610	220281576	1120.4	132598104	674.4
armardillo	262146	293686024	1120.3	174672232	666.3
cow	393218	440486872	1120.2	265112008	674.2
venus	1048578	1174498824	1120.1	698440072	666.1

Also, as the same representation of the mesh is used in both the Loop and Butterfly IWT, their peak memory usage can be calculated by the above equation. Then, by populating the actual statistics from Table 4.9(a) into Equation 4.2, the peak memory usage represented by the number of vertices on the experiment machine is given by

$$(80+6\times 56+2\times 56)V = 472V.$$

Given the analysis, the memory complexity of both the FWT and IWT, in theory, is $O(n)$, where n is number of vertices of the input mesh. In what follows, we consider the peak memory usage in practice. The peak memory usage was tracked by the tool `massif` [28], which counts system calls of memory allocation to measure memory usage. Table 4.10 lists the actual peak memory usage and relationship to the number of vertices in the FWT and IWT programs. The experimental results confirm our expectation that the peak memory usage is linear to the number of vertices. Also, since the peak memory usages of Loop and Butterfly are the same, Table 4.10 does not separate the different cases.

Next, we analyze the actual peak memory usage. From the results shown in Table 4.10, we can see computing the FWT and IWT require 13.2 MB to 1.12 GB and 7.9 MB to 666.1MB memory for meshes with 12288 to 1048578 vertices. Observed that the actual memory complexity of the FWT and IWT is linear with respect to the number of vertices as we have analyzed. By comparing the actual peak memory usage to the theoretical memory consumption analysis, we found the actual peak memory is 1.15 to 1.4 times higher than the estimated value. The reasons for the differences are explained as follows. First, some third-party libraries are used to implement the functionalities, which might allocate additional memory that is not counted in the memory analysis. Second, the memory allocated by some auxiliary data structures (e.g., `vector`, `queue`, and `set`) is not counted in the memory analysis as well.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this project, the Loop and Butterfly WTs for 3-D meshes and the lifting framework have been studied. As part of the work, the author has developed the WTT, which consists of a library for computing and implementing lifted WTs and some example application programs. The library can be easily used for computing the Loop and Butterfly WTs on a triangle mesh and implementing a custom WT. Moreover, Loop and Butterfly wavelet denoising and compression on triangle meshes have been demonstrated. The time and memory complexity of the library have been studied. The time complexity of the library was analyzed by considering the execution time, and our implementation is able to compute the FWT and IWT with time complexities $O(n \log n)$ and $O(n)$ with respect to the number of vertices, respectively. The memory complexity of the library was analyzed by measuring the peak memory usage, and our implementations have been demonstrated to have a memory complexity of $O(n)$ with respect to the number of vertices. Our results have shown that the library computes the Loop and Butterfly WT reasonably fast.

5.2 Future Work

Although our implementations have achieved the desired level of performance, there is still potential work worth exploring in the future. As mentioned in Section 4.4.2, the mesh-connectivity data structure used for identifying subdivision connectivity is simply an extension of `CGAL::Polyhedron_3`, which might not be efficient in terms of memory utilization. Some research on data structure improvement could be done to reduce the memory requirement.

In addition, as mentioned in Section 3.4.1, defining a lifted WT requires not only the lifting functions but also functions regarding topology (i.e., subdivision connectivity and topologic refinement and coarsening). Although the WTT is able to handle arbitrary 3-D meshes, we only provide the topology related functions for the PTQ case on triangle meshes. Users have to provide their own definitions for these functions. Nonetheless, the vast majority of work for defining a WT is implementing the topology related functions. Further researches on implementing these functions on other types of 3-D meshes (e.g., quadrilateral meshes) could reduce the work of users.

Bibliography

[1] Object File Format, November 2019. http://shape.cs.princeton.edu/benchmark/documentation/off_format.html.

[2] M. D. Adams. Course: Multiresolution signal and geometry processing with c++, November 2019. https://www.ece.uvic.ca/~mdadams/waveletbook.

[3] M. Bertram. Biorthogonal loop-subdivision wavelets. *Computing*, 72(1-2):29–39, April 2004.

[4] Blender Foundation. Blender, November 2019. https://www.blender.org/.

[5] Boost. The Boost Library, November 2019. http://www.boost.org.

[6] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, 1978.

[7] CGAL. The Computational Geometry Algorithms Library, November 2019. http://www.cgal.org.

[8] K. Crane. 3-D Model Repository, November 2019. https://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/.

[9] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 173–182. ACM, 1995.

[10] Free Sofware Foundation, Inc. The GNU Profiler, November 2019. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

[11] K. Hormann. An easy way of detecting subdivision connectivity in a triangle mesh. 2002.

[12] Wenzel Jakob, Marco Tarini, Daniele Panozzo, and Olga Sorkine-Hornung. Instant field-aligned meshes. *ACM Transactions on Graphics.*, 34(6), October 2015. https://github.com/wjakob/instant-meshes.

[13] Kitware Inc. Cmake, November 2019. https://cmake.org/.

[14] L. Kobbelt. $\sqrt{3}$ Subdivisionn. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 103–112, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[15] C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, University of Utah, UT, USA, 1987.

[16] M. Lounsbery. *Multiresolution Analysis for Surfaces of Arbitrary Topological Type*. PhD thesis, University of Washington, WA, USA, 1994.

[17] M. Lounsbery, T. D. DeRose, and J. Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Transactions on Graphics*, 16(1):34–73, January 1997.

[18] S. G. Mallet. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[19] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Inc., New York, NY, USA, 1987.

[20] MeshLab. The open source system for processing and editing 3D triangular meshes., November 2019. http://www.meshlab.net.

[21] E. Praun and H. Hoppe. Data files correspond to Spherical Parametrization and Remeshing, 2003. http://hhoppe.com/proj/spherepara m/.

[22] E. Praun and H. Hoppe. Spherical parametrization and remeshing. *ACM Transactions on Graphics*, 22(3):340–349, July 2003.

[23] Schröder, P. and Sweldens, W. Spherical wavelets: Efficiently representing functions on the sphere. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 161–172, New York, NY, USA, 1995. ACM.

[24] Schröder, P. and Sweldens, W. Spherical wavelets: Texture processing. In *Rendering Techniques '95*, pages 252–263, Vienna, 1995. Springer Vienna.

[25] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Applied and Computational Harmonic Analysis*, 3(2):186–200, 1996.

[26] G. Taubin. Detecting and reconstructing subdivision connectivity. *The Visual Computer*, 18, August 2001.

[27] TurboSquid Inc. Free3d, November 2019. https://free3d.com.

[28] Valgrind™ Developers. Valgrind Tool Suite, November 2019. http://valgrind.org/.

[29] H. Wang, K. Qin, and H. Sun. $\sqrt{3}$-subdivision-based biorthogonal wavelets. *IEEE Transactions on Visualization and Computer Graphics*, 13, October 2007.

[30] H. Wang and K. Tang. Biorthogonal wavelet construction for hybrid quad/triangle meshes. *The Visual Computer*, 25(4):349–366, Apr 2009.

[31] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, Jan 1985.