To appear in *Journal of Systems and Software*

PAT: A Pattern Classification Approach to Automatic Reference Oracles for the Testing of Mesh Simplification Programs

W. K. Chan [a], S. C. Cheung [b], Jeffrey C. F. Ho [c], and T. H. Tse [d]

[a] *City University of Hong Kong, Tat Chee Avenue, Hong Kong*

[b] *The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong*

[c] *University College London, Gower Street, London*

[d] *The University of Hong Kong, Pokfulam, Hong Kong*

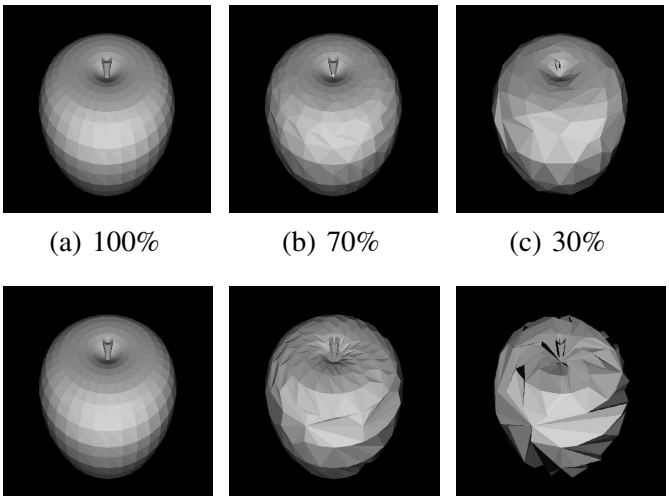
Abstract

Graphics applications often need to manipulate numerous graphical objects stored as polygonal models. Mesh simplification is an approach to vary the levels of visual details as appropriate, thereby improving on the overall performance of the applications. Different mesh simplification algorithms may cater for different needs, producing diversified types of simplified polygonal model as a result. Testing mesh simplification implementations is essential to assure the quality of the graphics applications. However, it is very difficult to determine the oracles (or expected outcomes) of mesh simplification for the verification of test results.

A reference model is an implementation closely related to the program under test. Is it possible to use such reference models as pseudo-oracles for testing mesh simplification programs? If so, how effective are they?

This paper presents a fault-based pattern classification methodology, called PAT, to address the questions. In PAT, we train the C4.5 classifier using black-box features of samples from a reference model and its fault-based versions, in order to test samples from the subject program. We evaluate PAT using four implementations of mesh simplification algorithms as reference models applied to 44 open-source three-dimensional polygonal models. Empirical results reveal that the use of a reference model as a pseudo-oracle is effective for testing the implementations of resembling mesh simplification algorithms. However, the results also show a tradeoff: When compared with a simple reference model, the use of a resembling but sophisticated reference model is more effective and accurate but less robust.

Key words: Test oracles, software testing, mesh simplification, graphics rendering, pattern classification reference models.



(a) 100% (b) 70% (c) 30%

(d) 100% (e) 70% (f) 30%

Fig. 1. Mesh simplification of polygonal models of a properly rendered apple (top row), and a badly rendered apple (bottom row)

1 Introduction

Computer graphics components are essential parts of real-life multimedia applications, such as medical imaging (Ahmed et al., 2002) interactive advertisement, and graphics-based entertainment. Many of these components use polygonal models (Luebke, 2001; Luebke et al., 2003) to visualize graphics. For interactive graphics-based software, such as the examples above, it is important to be responsive to the environment. Slow rendering of graphics is undesirable.

⋆⋆ This research is supported in part by a grant of City University of Hong Kong (project no. CityU 7002324) and grants of the Research Grants Council of Hong Kong (project nos. 111107, RPC07/08.EG24, and 714504). A preliminary version of this paper was presented at the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006) (Chan et al., 2006b). Part of the research was done when Ho was with The University of Hong Kong, Pokfulam, Hong Kong.

⋆⋆⋆ All correspondence should be addressed to Dr. W. K. Chan, Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. Email: wkchan@cs.cityu.edu.hk.

Mesh simplification (Cignoni et al., 1998; Luebke, 2001; Luebke et al., 2003) is a mainstream technique to address this problem. It aims at transforming a given three-dimensional (3D) polygonal model to one that has fewer polygons but resembles the original shape and appearance as much as possible. It therefore varies the level of sophistication of graphical outputs in order to save the computation time of details that need not to be seen clearly, such as when certain objects are supposedly viewed from a distance. Figure 1 shows two simple illustrations of mesh simplification, in which two apples are modeled by different numbers of polygons; the number of polygons to model the same apple decreases gradually from left to right, yielding a progressively coarser image as a result. Notice that a faulty implementation of mesh simplification may, by coincidence, render a better-looking graphic (such as Figure 1(e)) than one using a more aggressive simplification percentage (such as Figure 1(c)) and, hence, trick the careless testers to accept it as desirable outcome.

In many of these applications, graphical objects are stored in data structure such as polygonal models. Applications make use of various graphical functions, such as in the form of in-house libraries, to manipulate the data structures and render target graphical objects. Since the graphical outputs of such application are tightly-coupled with the libraries, the correctness of mesh simplification functions critically affects the correctness and the non-functional qualities of the applications.

The testing of mesh simplification implementations is an important part of the quality assurance process of graphics rendering software. This paper focuses on the issue regarding the effectiveness of failure identification, which will be explained below. We first introduce some terminology.

A *formal test oracle*, or *oracle* for short, is a mechanism against which testers can check the output of a program and decide whether it is correct. When an oracle is not available, other means of determining whether the test result is correct are known as *pseudo-oracles*. When an oracle or pseudo-oracle establishes that the output of the program is incorrect, such an output is known as a *failure*.

For visual approximation software such as mesh simplification implementations, however, comparing the actual graphical output of a test case with the expected result is challenging. On one hand, an automatic pixel-by-pixel verification is unreliable owing to the approximate nature of the rendering software. On the other hand, while software developers may serve as a manual test oracle, human judgment is often laborious, subjective, and error-prone.

When a formal oracle is hard to avail or costly to apply, the situation is well known as the *test oracle problem*. In short, there is a test oracle problem in the testing of mesh simplification implementations. To ease the test oracle problem, testers may like to find a pseudo-oracle that can serve as a good statistical approximation.

Researchers have applied classification techniques to address the above problem. A classification approach to program testing usually involves two steps: training a classifier to distinguish failures from successful cases on a selected subset of results, and then applying the trained classifier to identify failures in the main set of results. As we will discuss in Sections 2 and 3.1.2, many existing techniques use the same program for both the training and testing phases. On the other hand, it is unreliable to assume that a program is of sufficiently high quality for training the classifier before that program is actually tested.

We observe that different mesh simplification programs have been developed because no mesh simplification algorithm excels at simplifying all graphical models in all situations (Luebke and Erikson, 1997; Luebke, 2001; Luebke et al., 2003). For instance, the functional and non-functional requirements (such as performance and storage requirements) of individual programs may differ, so that different implementation considerations are adopted for various polygon simplification techniques.

Another observation is that a software development project usually produces a series of versions of the same program. Such a program is often the result of continuous improvements over various versions. Software developers of mesh simplification components may refer to other published or accessible mesh simplification techniques to compare and judge whether their own programs run anomalously. To ease our discussions, an existing implementation closely related to the program under development will be called a *reference model*.

It motivates us to study whether it is feasible to use a reference model as a means for checking automatically the correctness of the test outputs of another program. This paper presents a fault-based pattern classification methodology called PAT. The methodology extracts black-box features from the outputs of a reference model and its fault-based versions as the training dataset for identifying failures of the program under test.

We evaluate PAT using four implementations of mesh simplification algorithms as reference models, and apply them to render 44 open-source three-dimensional polygonal models. We generate training samples from each of the subject programs to train the C4.5 classifier, which is applied to test samples from the other three. In other words, one program will be used as a reference model to train the classifier and to identify failures of the other programs.

We then verify the performance of PAT. The result indicates that it is generally effective, accurate, and robust to use an implementation of a resembling algorithm as a reference model for training the C4.5 classifier to be a pseudo-oracle for the program under test. However, the empirical result also indicates a tradeoff: When compared with a simple reference model, the use of a resembling but sophisticated reference model is more effective and accurate but relatively less robust.

The contributions of the preliminary version (Chan et al., 2006b) of this paper are as follows: It proposes the idea of constructing a pseudo-oracle via a reference model of the program under test. It evaluates the proposal to compare the effectiveness of using a resembling reference model and a dissimilar reference model. The evaluation metrics are the percentage of mutants killed and the percentage of test cases being classified correctly. Finally, it presents an initial guideline, which recommends using the implementations of resembling algorithms to address the test oracle problem for mesh simplification programs.

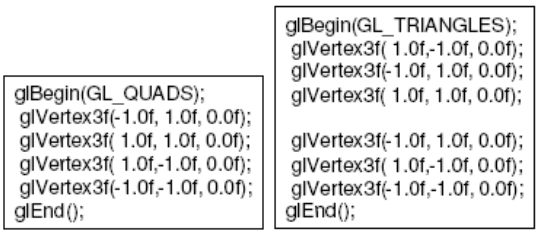
The extended contributions of this paper are threefold: Firstly, it formalizes the methodology. Secondly, it significantly extends the analysis in Chan et al. (2006b) to evaluate the methodology in three dimensions of performance (namely effectiveness, accuracy, and robustness). These three dimensions have statistical rigors and strong co-relations among them. They form a suite of metrics that properly evaluates a binary classification scheme. The results show that the proposed methodology is promising in identifying failures of mesh simplification programs effectively. It also presents a tradeoff between effectiveness and robustness. Last, but not the least, based on the empirical results, it provides a further handy guideline to testers: when they have no idea of the relationships between reference models and the programs to be tested, they may use simple mesh simplification reference models to achieve robustness.

The rest of the paper is organized as follows: Section 2 reviews related approaches to testing software with graphical interfaces. In Section 3, we present a pattern classification methodology to tackle the test oracle problem above. We evaluate our methodology by experimentation on four mesh simplification programs in Section 4. The evaluation results are discussed in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

This section reviews related work on the test oracle problem for the testing of software with graphical interfaces.

Berstel et al. (2005) design a formal specification language VEG to describe Graphical User Interfaces (GUIs). Although they propose to use a model checker to verify a VEG specification, their approach deals only with verification and validation before implementation and does not handle the identification of failures in an implementation. D'Aubbourge et al. (1998) propose a software environment to include formal operations in the design process of user interface systems. Like the work of Berstel et al. (2005), the verification of a GUI specification may then be arranged. Memon et al. (2000) propose to identify non-conformance between a test specification and the resulting execution sequence of events for each test case. They assume that a test specification of internal object interactions is available.



(a) Method 1 (b) Method 2

Fig. 2. Different lists of graphics rendering commands to render the same object

This type of approach is intuitive and popular in the conformance testing of telecommunication protocols. Sun et al. (2004) propose a similar approach for test harnesses. Memon et al. (2003) further evaluate different types of oracle for GUIs. They suggest the use of simple oracles for large test sets and complex test oracles for small test sets.

Other researchers and practitioners also propose similar approaches to test programs having outputs in computer graphics. For example, gDEBugger [1] checks the conformance of the list of commands issued by an application to the underlying graphics-rendering Application Programming Interface (API) of OpenGL (Segal and Akeley, 2004). However, many different sets of commands can be rendering the same graphics image. For example, Figure 2 shows two pieces of code, each drawing the same square in its own way. Checking the equivalence of lists of graphics rendering commands is an open challenge. Bierbaum et al. (2003) also point out that not all graphical applications make use of GUI widgets for graphics output.

Bierbaum et al. (2003) present an architecture for automated testing of virtual reality application interfaces. It first records the states of the input devices in a usage scenario of an application. Users may further specify checkpoints of the scenario as the expected intermediate states of test cases. In the playback stage of a test case, the architecture retrieves the recorded checkpoints and verifies the corresponding states of the test case against the checkpoints. Takahashi (2001) proposes to compare objects of the same program when they are scaled proportionally. For example, they propose to check whether the angles are identical and the lengths of edges are proportional. This may be considered as a type of metamorphic testing (Chen et al., 1998, 2002). Mayer (2005) proposes to use explicit statistical formulas such as mean and distributions to determine whether the output exhibits the same characteristics.

The test oracle problem has also been studied in other contexts. Ostrand et al. (1998) propose an integrated environment for checking the test results of test scripts, so that testers can easily review and modify their test scripts. Dillon and

[1] Available at http://www.gremedy.com/.

Ramakrishna (1996) discuss a technique to reduce the search space of test oracles constructed from a specification. Baresi et al. (2002) propose to use program assertion (Meyer, 1992) to check the intermediate states of programs.

More specifically, there are techniques for applying pattern classifications to alleviate the test oracle problems. Last et al. (2003) and Vanmali et al. (2002) propose to apply a data mining approach to augment the incomplete specification of legacy systems. They train classifiers to learn the casual input-output relationships of a legacy system. Podgurski et al. (2003) classify failure cases into categories. However, they do not study how to distinguish correct and failure behaviors in programs. Their research group (Francis et al., 2004) further proposes classification tree approaches to refine the results obtained from classifiers. Bowring et al. (2004) use a progressive machine learning approach to train a classifier on different software behaviors. They apply their technique in the regression testing of a consecutive sequence of minor revisions of the same program.

Chan et al. (2006a) propose to use supervised machine learning approaches to recognize output anomalies relevant to temporal relationships between media objects in multimedia applications. The work studies the effectiveness of the following technique: (1) comparing the actual outputs against expected outputs for some test cases of a test suite, and (2) using a machine learning approach to identify failures automatically through the remaining test cases. Compared to the present work, our previous work does not study the failures within a media object, such as failures in graphical outputs. More distinctly, PAT replaces step (1) above by an *automatic* step, which uses the outputs of reference models to delineate desirable behaviors and the outputs of the mutants of the reference models to delineate undesirable behaviors.

We observe that Bowring et al. (2004) use mutants of a program to serve as a sequence of program versions and extract white-box features from them to train classifiers. As we will discuss in the next section, a passed test case may only be coincidentally correct and, hence, it is intuitively less reliable to use white-box features (such as method counts or some program internal behavior) to distinguish a failure from a correct one. PAT proposes to use existing programs with different algorithms as reference models to train classifiers. We use only black-box features extracted from test cases and outputs. The work of Bowring et al. and ours augment each other.

3 Our Methodology

This section will be organized as follows: We will firstly present preliminary discussions in Section 3.1, which include a description of pattern classification (Section 3.1.1), our understanding of reference models (Section 3.1.2), and the reason for choosing black-box features (Section 3.1.3). We will then present PAT

in Section 3.2.

3.1 Preliminaries

3.1.1 Pattern Classification Technique

A pattern classification technique (Duda et al., 2000) normally consists of a training phase and a testing phase. The training phase guides a classifier to categorize given samples into different classes based on selected *classification features* or simply *features* of the samples with respect to a reference model. In the testing phase, the trained classifier assigns a test case to a trained class. For the purpose of identifying failures, we restrict our attention to the study of two classes in this paper, namely one categorized as *passed* and the other as *failed*.

3.1.2 Reference Model

This section explains our concept and choice of a reference model for mesh simplification programs.

We consider mesh simplification algorithms (Luebke et al., 2003) to be a set of requirements that software developers implement while fulfilling various tradeoffs including rendering speed, space, robustness issues, and so on. A mesh implication program is an implementation that fulfills these requirements.

In theory, a set of high level requirements, such as mesh simplification in general, can be defined via a specification S. The specification may be further designed (say as two algorithms) and, hence, implemented as two distinct programs P and P'. Each of these programs is a refinement of the specification, written as $P \sqsubseteq S$ and $P' \sqsubseteq S$. Thus, we define a *reference model* (with respect to the program P) as an existing program P' such that both $P \sqsubseteq S$ and $P' \sqsubseteq S$. In practice, graphical software developers may use their experience to determine whether a program is a (*resembling*) reference model of another program. [2]

One way to produce a reference model is to generate program mutants and use the original program as the reference model. Existing classification techniques in software testing (Bowring et al., 2004; Chan et al., 2006a) uses program mutants (generated by mutation operators) to serve as a sequence of program versions and extract method counts or temporal data from test case executions as features to train classifiers. They do not explore black-box features for classifier training. While program mutants have been verified to be useful in test evaluation experiments under a laboratory setting (Andrews et al., 2005) (for example, they are compared with "the 'realistic' faults hand-seeded by experienced engineers"), their practical

[2] We note that such a practical way to identify a reference model has been advocated in the evaluation of software architectures (Bass et al., 2003).

use in software testing is still limited. We will come back this point at the end of the subsection.

Another way is to select a particular version of a program among a series of slightly modified versions in a software development project and treat this version as the reference model. Nevertheless, a version of a program under development is often faulty, making it unreliable as a reference model for correctness verification.

A third way is use a workable program that implements the same functionality to be the reference model of the program under test. We will choose the third approach in our method to test mesh simplification software. Developers of mesh simplification programs may modify existing programs to suit their specific needs. This may require extensive modifications such as replacing the original algorithms by entirely different approaches.

We propose to use the existing programs as reference models. We use such a program as a kind of source and employ a fault-based approach (e.g., mutation analysis) to construct faulty versions of the source to train a classifier. We make use of their behaviors as automatic oracles for verifying the test results of new mesh simplification programs. We note that a trained classifier is reusable for testing multiple programs by researchers and practitioners. It may potentially leverage the cost of using a dynamic fault-based approach and make such an approach more acceptable by the industry. Our procedure will be described in Section 3.2.

3.1.3 Black-box features

We propose to extract black-box features from a reference model and use these features to train a classifier as a pseudo-oracle for the testing of mesh simplification programs. However, there is a problem of coincidental correctness when executing faulty programs over test cases (Richardson and Thompson, 1993; Hierons, 2006). For example, an activated fault may affect the program state during execution, but the failure of the state may not be propagated to the output to show a visible failure. In the presence of coincidental correctness during the execution of a passed test case, a white-box feature such as the method counts of program execution will unavoidably include the failed behavior, which will in turn affect the quality of the white-box feature. We note also that coincidental correctness is an internal property of faulty programs. By its problem nature, such a property is *never* observable externally. Hence, black-box features do not suffer from such a limitation. On the other hand, as the implementation of a mesh simplification program may differ a lot from that of a reference model, their outputs may also be different. We will evaluate the impact of such differences on our approach.

3.2 Formal Model of PAT

We present a formal model of PAT in this section. Let C be a classifier to be trained to test a program P using a reference model R. Let $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ be a set of input files representing 3D polygon models. Executing R over $\mathcal{M}$ will produce a set of outputs $\{R(m_1), R(m_2), \ldots, R(m_k)\}$. A fault-based technique, such as the mutation analysis technique in Andrews et al. (2005), can be used to generate a set of faulty versions of the program R. Suppose the mutants of R are denoted by $\{\overline{R_1}, \overline{R_2}, \ldots, \overline{R_u}\}$. Executing each $\overline{R_j}$ of these mutants over $\mathcal{M}$ will produce a corresponding set of outputs $\{\overline{R_j}(m_1), \overline{R_j}(m_2), \ldots, \overline{R_j}(m_k)\}$.

Let $(f_1, f_2, \ldots, f_v)$ be a list of classification feature extraction functions that extracts features from input polygonal models and program outputs. Given an input model m_i, the program R, and the output $R(m_i)$, the above list of functions will extract a list of features $(f_1(m_i, R, R(m_i)), f_2(m_i, R, R(m_i)), \ldots, f_v(m_i, R, R(m_i)))$, known as a *vector of extracted features*.

Similarly, for each mutant $\overline{R_j}$, the list of functions will produce a corresponding vector of extracted features $(f_1(m_i, \overline{R_j}, \overline{R_j}(m_i)), f_2(m_i, \overline{R_j}, \overline{R_j}(m_i)), \ldots, f_v(m_i, \overline{R_j}, \overline{R_j}(m_i)))$. If the vector of extracted features produced from mutant $\overline{R_j}$ is identical to that produced from the original R, PAT will not use it to verify the subject program P. We refer to the remaining vectors as *non-equivalent mutation vectors*. PAT labels the vector from the original R as *passed* and each non-equivalent mutation vector as *failed*. PAT uses all these labeled vectors to train the classifier C.

To test a program P, PAT constructs a set of vectors of extracted features for P using the above scheme but replacing R by P. PAT then passes each of these vectors of P to the trained classifier C and let the classifier label the vector. A vector labeled as *passed* will be considered as a test case that reveals no failure, while a vector labeled as failed will be considered as a test case that reveals a failure.

PAT advocates that R should be a resembling reference model, which is determined by the mesh simplification experts. In our experiments in Section 4.4 that test the feasibility and effectiveness of PAT, we also construct abnormal versions of P to be used as mutants. We again use C to classify the non-equivalent mutation vectors produced by these mutants of P. Also, we use other reference models that do *not* resemble the program under test as benchmarks for the evaluation of PAT.

4 Experimentation on PAT

This section presents an experimentation on PAT. Section 4.1 firstly describes our experimental method to extract black-box pattern classification features. Section 4.2 then describes the subject programs to be used for evaluation. Next, we will describe how sample data are selected for the experiments. Finally, we will

describe the experimental procedure to evaluate PAT.

4.1 Classification Feature Selection

Mesh simplification aims at retaining the skeleton form of an original polygonal model and removing unnecessary details, as illustrated in Figure 1, to save processing time. Since the actual shape of a simplified polygonal model differs from that of the original, lighting effects such as shadowing cannot be adopted without re-rendering. These necessary changes inspire us to propose to extract classification features based on the strengths of image frequencies and the lighting effects in the experimentation on PAT. To avoid any undue bias for particular images, we use the generic spatial frequency spectrum to extract the image frequencies.

Classification feature 1: Change of ratios of major and minor image frequencies. We extract the amount of major and minor image frequencies that remain in a simplified polygonal model for a given percentage of simplification using standard fast Fourier transform (Gonzalez and Woods, 2002; Nixon and Aguado, 2002). The level of simplification is normally defined by a simplification percentage, as illustrated in the labels in Figure 1.

Since rendering an image is relatively computationally expensive, we adopt the advice of Memon et al. (2003) that "complex oracles are better at detecting faults than the simplest ones." We propose to extract a number of frequency attributes from the images, which we will refer to as *image frequencies*. We use them to synthesize classification features, which serve as the basis for training the classifier for recognition as a test oracle. We extract classification features based on different orientations. To ease our presentation, we will use the notation "$Image_{r\%}$" to represent an image simplified to $r\%$ from the original (which will be denoted as $Image_{100\%}$).

We observe that a mesh simplification program may simplify a given polygonal model to different levels (such as $Image_{90\%}$, $Image_{80\%}$, ..., and $Image_{10\%}$). We firstly sort the frequencies obtained from the fast Fourier transform above, and determine a sequence of ratios of major to minor image frequencies for the simplification of the same polygonal model to various levels (see below). We then fit the sequence of normalized ratios of major to minor image frequencies (against the level of simplification) using the regression curve-fitting technique. The coefficients of the fitted curves represent the values of the corresponding classification feature.

For a given polygonal model, ratios are calculated for the original image as well as simplified images at 10% intervals starting from 100%. (That is, $Image_{100\%}$, $Image_{90\%}$, ..., and $Image_{10\%}$.) The curve fitting program applied in

our experiments is ImageJ [3] , which uses a simplex method based on Press et al. (1992). The details for determining a ratio of major to minor image frequencies is as follows.

Ratio of major to minor image frequencies. We first extract the amount of signals of the original polygonal model $Image_{100\%}$ that remains in a simplified model $Image_{r\%}$. We deconvolve $Image_{100\%}$ with $Image_{r\%}$ (Gonzalez and Woods, 2002; Nixon and Aguado, 2002). The result forms a filter, $Filter_{100\%-r\%}$, representing the transformation from $Image_{100\%}$ to $Image_{r\%}$. Informally, a more simplified polygonal model will have a higher overall value in the resultant filter. This is because when fewer polygons suffice to model an image, smaller amounts of image frequencies of the original model remain in the simplified version. The stronger the strength, the more it will contribute to the image. Signals with major contributions are low frequency signals contributing major image frequencies of $Image_{100\%}$. Signals with minor contributions are high frequency signals contributing minor image frequencies of $Image_{100\%}$. Thus, we sort the image frequencies according to signal strengths to facilitate the determination of a threshold T for us to extract major and minor contributions passing through a filter $Filter_{100\%-r\%}$. We may set T as the mean value of all contributions of all signals of the original image $Image_{100\%}$. Other choices of T may include the mean $\pm$ various multiples of the standard deviation. After deciding on the threshold T, all signals of $Image_{100\%}$ are checked against it. By marking signals with contributions above and below the threshold T, a mask in the frequency domain is produced.

This mask is used to split $Filter_{100\%-r\%}$ into two parts. One part is responsible for keeping signals with major contribution in the output (that is, how large the portion of major image frequencies remains). It contains the signals whose strengths are at least the same as T. The other part is responsible for keeping the minor counterpart (that is, how large the portion of minor image frequencies remains). It includes all the remaining signals. We recall that, as a polygonal model is being simplified, a smaller amount of minor image frequencies from the original model will be retained. Major image frequencies are also reduced but usually to a lesser extent. We compute the sum of values of the parts responsible for the major image frequencies, as well as that for the minor image frequencies: The ratio of the two sums is $sum_{minor} \div sum_{major}$.

The sets of coefficients as the classification feature set. Seven different thresholds are used. They include various combinations of the mean and standard deviations of the signal contribution for the image at 100%, namely $M-3\sigma$, $M-2\sigma$, $M-\sigma$, M, $M+\sigma$, $M+2\sigma$, and $M+3\sigma$, where M is the mean and σ is the standard deviation. [4] For each threshold value, we construct one set of the above coefficients.

[3] Available at http://rsb.info.nih.gov/ij/.

[4] A polygonal model that renders useful graphics contains many signals. Since the number of signals is large, the use of 3 standard deviations will cover most of the available data.

From a statistical point of view, when the number of frequencies is large (> 20, which is the case in our experiments), the distribution of frequencies in an image can be regarded as a normal distribution. By covering up to 3 standard deviations, the effect of over 99% of the frequencies in an image are considered in our experiments.

Classification feature 2: Lighting effect. The second set of classification features is concerned with the general lighting of the rendered polygonal models. This is to remedy the use of ratio in the first classification feature, which eliminates the effect of any changes (in the numerator and denominator) that happen to be proportional.

For every image, the average value of the maximum and minimum pixel brightness is computed and used a classification feature. This feature set would alert the classifier of any polygonal model rendered with abnormal brightness or darkness.

4.2 Subject Programs: Reference Models

Our subject programs consist of four different programs, each with a unique mesh simplification algorithm implemented. They are all written in Java. These algorithms are:

(i) *Shortest*. One of the basic mesh simplification algorithms. It always picks the shortest edge of a mesh to collapse.
(ii) *Melax* (1998). Another algorithm using the edge collapsing approach. The heuristic cost estimate for collapsing an edge is the product of the length and curvature of the edge. A vertex connected to edges with the lowest cost estimates is first removed.
(iii) *Quadric* (Garland and Heckbert, 1997). An algorithm that contracts pairs of vertices rather than edges, so that unconnected regions can also be joined. It approximates contraction errors by quadric matrices.
(iv) *QuadricTri* (Garland and Heckbert, 1997). A variant of the *Quadric* algorithm. It also takes into account the sizes of triangles around vertices during contraction. If the area of a triangle is large, the error of eliminating the vertex will be large.

Figure 4 shows the rendering results at 10% simplification level by the four programs using an original spider model with 9,286 triangles.

Quadric and *QuadricTri* are two subject programs with resembling algorithms, while other combinations are dissimilar.[5] These two properties will be used as benchmarks for each other. They help us compare the circumstances in which software developers adapt an existing algorithm to implement their own versions.

[5] This categorization is confirmed by members of the graphics research group at The University of Hong Kong.

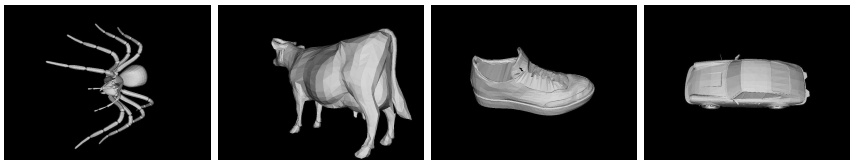
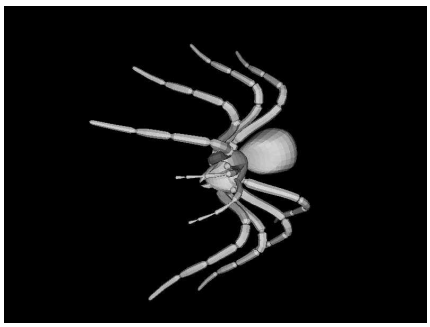
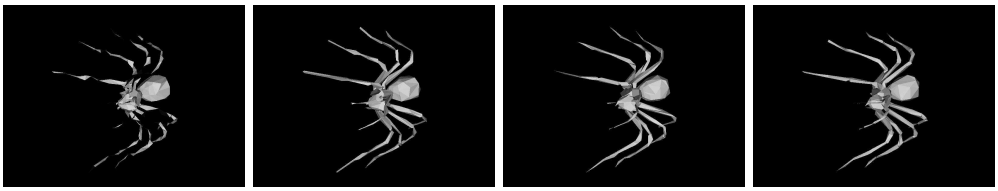


Fig. 3. Sample rendered graphics without any simplification



(a) Original



(b) via *Shortest* (c) via *Melax* (d) via *Quadric* (e) via *QuadricTri*

Fig. 4. Graphics of spider model: original and simplifications to 10% level via respective techniques

Every program accepts two inputs: a file storing the input of a 3D polygonal model in standard .PLY format and an integer (from 0 to 100) indicating the percentage of polygons in the model that should result from the simplification process. If the value of the integer input is zero, only the background will be shown. We use the black color as the background in our experiments. Each program scales the polygonal model to within a bounding cube $(-1, -1, -1)$ to $(1, 1, 1)$, centered at $(0, 0, 0)$. The operations to scale and relocate models in 3D space are common in many graphics applications. The programs output images with a resolution of 800×600 showing simplified versions of the polygonal model. Examples of rendered graphics according to the input files without simplification are shown in Figure 3.

4.3 Test Case Selection

PAT defines two classes for pattern classification, namely *passed* and *failed*.

Shortest	*Melax*	*Quadric*	*QuadricTri*
350	401	1,122	1,187

Table 1
Numbers of mutants of subject programs

In the evaluation experiments, to collect training samples of the *passed* class, we execute the set of 44 open-source 3D polygonal models [6] over every reference model. In order to better utilize the polygonal models, each is further rotated in 22 different orientations. They correspond to rotating a model along the x-axis every 22.5 degrees and along the y-axis every 45 degrees. Thus, each original polygonal model generates 22 inputs representing rotated models with various orientations, and each input produces 11 images at various simplification levels ($Image_{100\%}, Image_{90\%}, \ldots, Image_{10\%}, Image_{0\%}$). In other words, $22 \times 11 = 242$ images are produced from every original polygonal model.

To collect training samples for the *failed* class, program mutants are generated from the reference model using a mutation tool known as *mu*Java [7] (Ma et al., 2005). We use all the mutants generated from the conventional mutation operators of the mutation tool. We have taken a few measures to vet the mutants thus generated in our experiments. Normally, our subject programs take no more than 30 seconds to generate an image from the input of our 44 selected 3D polygonal models. Mutants that take more than 3 minutes to execute an input are considered to have failed to terminate. They have not been used in our experiments. Mutants that produce non-background contents for rendering graphical models at 0% are also removed because they are very obvious failures. If any program assertions inside the implementation are violated during the execution of any inputs, the corresponding mutants are excluded. If a mutant is found equivalent to the original implementation, it is removed. We further consolidate mutations that produce the same output images for the same corresponding inputs into one representative mutant (which is randomly selected amongst others). There are a total of 3,060 remaining mutants, as shown in Table 1. They are all used in the experiments.

Based on these mutants, we have tried our best effort to collect the classification features from more than 440,000 program executions. Since the programs are deterministic, we used more than 10 desktop computers to run samples for more than 2 months to collect all the data and extract the features described above. Because of the strenuous effort involved, we did not conduct similar case studies on other classifiers and features.

[6] Available at http://www.melax.com/polychop/lod_demo.zip. According to the above source, they are a "big demo" to convince skeptical visitors that the implemented simplification techniques are working. To give readers a sense of complexity, the numbers of polygons in these models range from 1,700 to 17,000.

[7] Available at http://www.isse.gmu.edu/∼ofut/mujava/. Version 1 is used.

We use the same number of training samples for each class bearing in mind that imbalanced data in the training of a classifier may result in biases or failures of classification algorithms (Weiss, 2004).

4.4 Experimental Procedure

This section describes the experimental procedure for evaluating PAT. We divide the set of publicly available 3D polygonal models (see Section 4.3) into two sets. Set_1 contains 13 models and Set_2 contains the rest. As we will explain below, the value of 13 is immaterial to PAT. We simply divide the models into two groups to ensure that the classifier would never come cross some of them in the training stage.

Each of the four subject programs is treated in turn as the reference model, which we will call R_A. In each case, the three remaining subject programs are treated as implementations under test, which we will refer to as P_B. Ten iterations are carried out for the preparation of different sets of training and testing examples. We have chosen 10 as the number of iterations in the experiments because it is sufficiently large to collect statistically valid data and yet not excessive in resource requirements — we have to run the experiments continuously on multiple machines for two months.

Classifier. A small pilot study with several major categories of classification algorithms has been carried out using sample data. The results indicate that the classifier C4.5 (Quinlan, 1993) together with the Adaboost M1 boosting method (Freund and Schapire, 1996) gives the most favorable performance in terms of effectiveness. (See the next section for a discussion of the metric.) C4.5 by itself is also a classical, representative, and prominent machine learning algorithm. Hence, we conduct our main experimental case study using this classifier. To select the classification result fairly among sample data, we use multiple (five) independent decision trees to form the basis, each with different random seeds. The classifier is a combination of these five decision trees. Predication is decided by casting equally weighted votes based on the five decision trees. We have searched the literature and find no consensus on a particular number of decision trees in a method. We have set this parameter to the value five simply for practical reasons.

Training stage. One-tenth of the 3D polygonal models are picked randomly from Set_2.[8] They are input to the original implementation of P_B for every 10% from 100% to 0% to check the accuracy of the trained classifier on *passed* examples from P_B.

[8] We note that in data mining benchmarking research, researchers may use 10%–90% of all data as the training set.

	Failed Test Case	**Passed Test Case**
Labeled as *failed*	true positive	false positive
Labeled as *passed*	false negative	true negative

Table 2
True and false positives and negatives

We use the following procedure to train a classifier: N 3D polygonal models from Set_1, where $1 \leq N \leq 5$, are randomly selected and executed with mutants of R_A for every 10% from 100% to 0% to produce training examples of *failed* outputs. These N polygonal models, together with the polygonal models from the remaining nine-tenth of Set_2, are then input to the original implementation of R_A to produce training examples of *passed* outputs. Classification features are extracted from the outputs of the *passed* and *failed* classes. Classifiers are trained with the values of these extracted classification features.

Testing (or evaluation) stage. The 3D polygonal models unused in the training stage are input to the original implementation as well as the mutants of P_B for every 10% from 100% to 0%. We note that all the polygonal models used in the testing stage are unseen (not even in different orientations) in the training stage. A mutant is marked as *killed* if more than *50%* of its outputs are classified as failed. The criterion of *50%* is chosen because there are two classes in the classification scheme, namely *passed* and *failed*, and we do not want the classifier to be biased in one direction or the other.

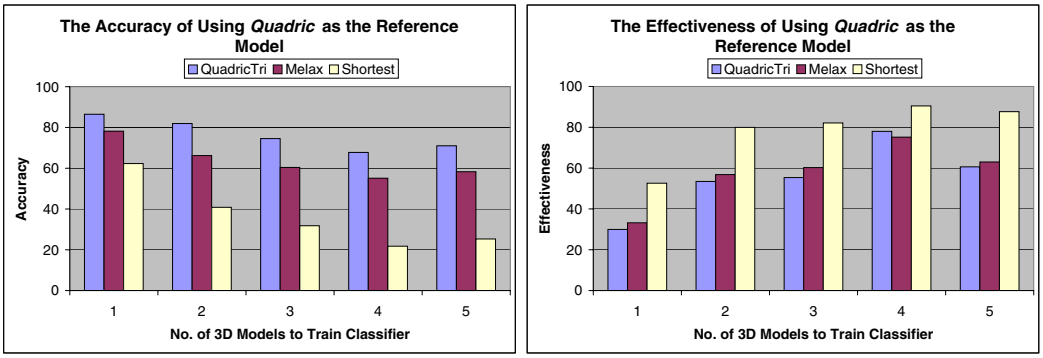
5 Experimental Evaluation Results

In this section, we analyze our experimental evaluation results on PAT.

5.1 Goodness Metrics

To facilitate discussions of our findings, we apply a few goodness measures. They are defined according to how the classifier labels test cases in comparison with the expected classification. There are four possible combinations as depicted in Table 2. A failed test case is known as *true positive* if it is labeled by the classifier as failed, and known as *false negative* otherwise. Similarly, a passed test case is known as *true negative* if labeled by the classifier as passed, and *false positive* otherwise.

Based on the above terminology, we define three goodness metrics for evaluating our experimental results on PAT.



(a) Accuracy (b) Effectiveness

Fig. 5. Accuracy and effectiveness of verification via *Quadric*

(i) *Accuracy* (Kohavi and Provost, 1998) is the percentage of test cases that are properly classified. It is also known as *error rate* in machine learning. It is a statistical measure to check the rate of correct predictions made by a model over a dataset (Kohavi and Provost, 1998).

$$\text{Accuracy} = \frac{\text{no. of true positives} + \text{no. of true negatives}}{\text{total no. of test cases}} \times 100\%$$

It is plotted in the graphs (a) of Figures 5, 6, 7, and 8.

(ii) *Effectiveness* is the percentage of failed test cases that are properly classified. It is called *sensitivity* (Hosmer and Lemeshow, 2000; Kohavi and Provost, 1998) in statistics. It measures how well a model correctly identifies a condition. We use the term effectiveness in this paper to align with the usual terminology in software testing research.

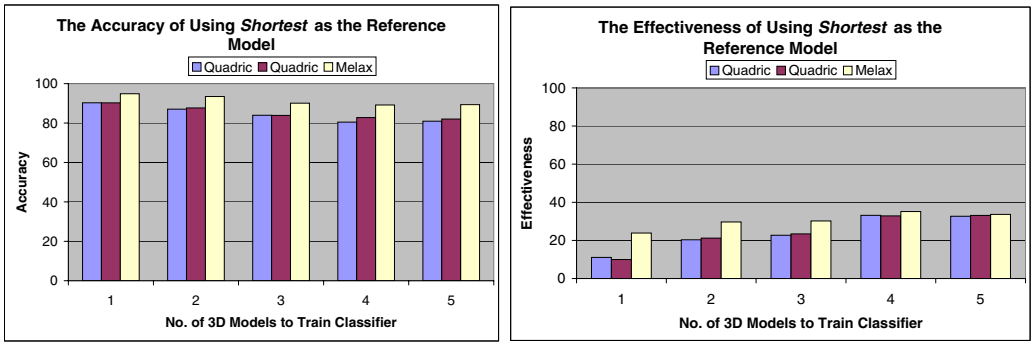
$$\text{Effectiveness} = \frac{\text{no. of true positives}}{\text{no. of true positives} + \text{no. of false negatives}} \times 100\%$$

It is plotted in the graphs (b) of Figures 5, 6, 7, and 8.

(iii) *Robustness* is the percentage of passed test cases that are properly classified. It is also known as *specificity* (Hosmer and Lemeshow, 2000; Kohavi and Provost, 1998) in statistics. It measures how well a model correctly identifies the negative cases. In statistical terms, a high specificity has a low Type 1 (false positive) error rate. We use the term robustness in this paper to reflect the testers' aspiration to minimize the errors due to these false positive test cases.

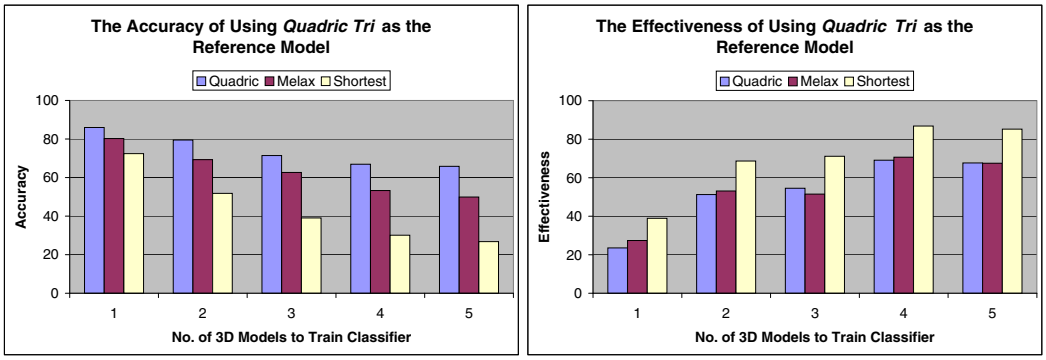
$$\text{Robustness} = \frac{\text{no. of true negatives}}{\text{no. of true negatives} + \text{no. of false positives}} \times 100\%$$

In our experimental case study of PAT, we study two classes (namely, passed and failed) in the classification scheme. Accuracy and precision are a pair of dimensions to study a binary classification. In machine learning, sensitivity and specificity are standard ways to represent the precision dimension. We thus use accuracy, effectiveness, and robustness as the three suggested measures in our experiments.



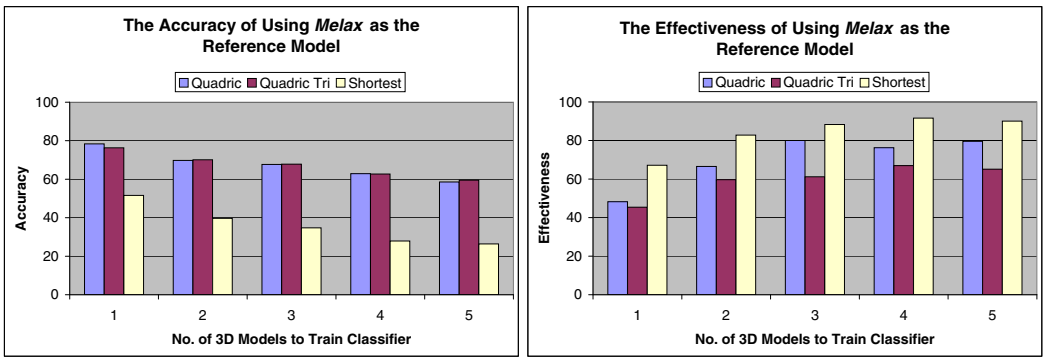
(a) Accuracy (b) Effectiveness

Fig. 6. Accuracy and effectiveness of verification via *Shortest*



(a) Accuracy (b) Effectiveness

Fig. 7. Accuracy and effectiveness of verification via *Quadric Tri*



(a) Accuracy (b) Effectiveness

Fig. 8. Accuracy and effectiveness of verification via *Melax*

5.2 Goodness of Automatic Test Oracles via Reference Models

In this section, we present the results of our case study. Both accuracy and effectiveness are immediately related to the quality of a testing technique in

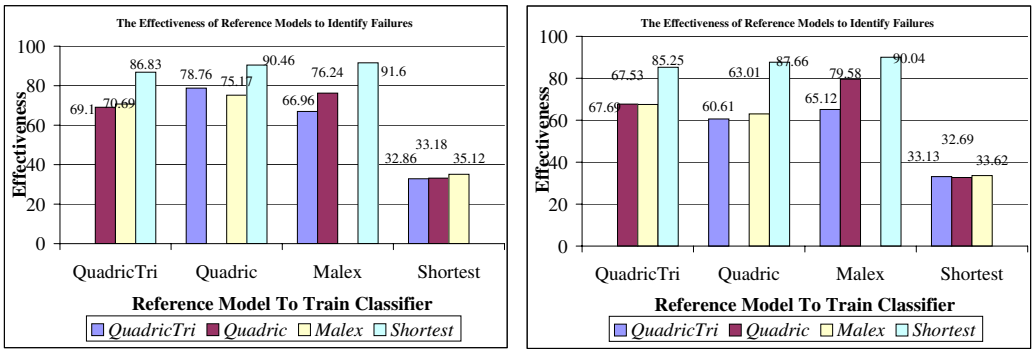
revealing failures. They will, therefore, be paired up for discussion first. Robustness will then be discussed.

Figure 5 shows the accuracy and effectiveness of verification using *Quadric* as the reference model for identifying failures of other subject programs. The horizontal axes show the number of 3D graphics inputs (N) used to train the classifier. Each histogram in the figure consists of three bars, representing the verification results for *QuadricTri*, *Melax*, and *Shortest*, respectively. Figure 6 shows the case of using *Shortest*, a basic but dissimilar reference model, for identifying failures of other subject programs. Figures 7 and 8 show the cases of using *QuadricTri* and *Melax*, respectively, as the reference models. Compared to *Shortest*, *QuadricTri* and *Melax*are relatively sophisticated. The main observation is: In the absence of any knowledge on how the given mesh simplification may behave, the use of a simple reference model will achieve accuracy but at the expense of effectiveness.

The plots in Figures 5(b), 6(b), 7(b), and 8(b) show that, in general, the effectiveness of verification gradually increases with the number of 3D polygonal models used to train the C4.5 classifier. It agrees with the usual expectation that the more data used for training, the better will be the classifier in learning a desirable behavior.

There are a total of 44 3D polygonal models used in the experiments. The use of 4 or 5 models for training means that around 10% of the data are used to train the C4.5 classifier. Using a large volume of data to train a classifier will reduce the chance that the latter will be biased (Beiden et al., 2003). On the other hand, researchers in software testing tend to use a much smaller training set. Bowring et al. (2004), for instance, use only 3% of the inputs to train a classifier to give encouraging preliminary results in their software testing experiments. This is understandable since it would be unproductive to evaluate the outcomes of software testing using a large percentage of the available data as the training set. In the sequel, to strike a balance between the two different views in the two communities, we will focus on the results that involve 4 or 5 3D polygonal models in the training phase. They are shown in the rightmost two sets of bars in each plot in Figures 5, 6, 7, and 8

The plots of testing results of *QuadricTri* in Figure 5(b) show that, on average, an effectiveness of 70% is achieved using a resembling and relatively sophisticated reference model (*Quadric*). The results of *Quadric* in Figures 7(b) and 8(b) are also similar. On the other hand, Figure 6(b) shows that the use of a simple reference model (*Shortest*) is significantly less effective. We also find that the test results of the other combinations follow this observation. It may indicate that using relatively sophisticated reference model can be effective to identify failures. However, Figures 5(a), 7(a), and 8(a) also indicate that it is inaccurate to use a relatively sophisticated reference model to check the implementation of a simpler system.



(a) Trained with 4 polygonal models out of 44

(b) Trained with 5 polygonal models out of 44

Fig. 9. Effectiveness of verification via various reference models.

One may wonder whether the high scores in the goodness measures are due primarily to (i) the similarity with the reference model or (ii) the sophistication of the reference model. To look into this interesting question, we need to examine the results of *all* reference models to train the C4.5 classifier.

Figure 9 depicts comparisons of the effectiveness of verification using various reference models. The horizontal axes show the reference models used to train the classifier. The three bars in the histogram of each reference model show the respective effectiveness of identifying failures of the other three subject programs. We observe that the set of bars for the *Shortest* reference model is significantly lower than the rest, while the other three sets of bars look quite similar. It indicates that the level of sophistication appears to be a factor which dominates over similarity. It further shows that *Melax* is also effective, in comparison with *Quadric* or *QuadricTri*, as a reference model for identifying failures. This being the case, one may wonder why one should not use a dissimilar but sophisticated program as a reference model to train a classifier. We need to further examine the issue through goodness metrics other than effectiveness.

A closer look of the accuracy and robustness measures may give some hints. Let us consider one scenario. Comparing the results in Table 3 with their counterparts in Figure 5(a), the accuracy for the classification of *Shortest* using *Melax* as the reference model is the lowest. For the classification of *Quadric* and *QuadricTri* using *Melax* as the reference model, the accuracy measures (60.73% and 61.12%, respectively) are similar to each other. The mean is 60.9%. In the case of utilizing 4 and 5 3D polygonal models as shown in the rightmost two sets of bars in Figure 5(a), the mean accuracy measure for the use of *Quadric* as a reference model for the testing of *QuadricTri* is 69.4%. However, the mean accuracy measure for the use of *QuadricTri* for testing *Quadric* is 66.4%. This result indicates that a resembling combination gives a slightly higher accuracy than a dissimilar combination. Since the difference is only marginal, let us examine the full picture of the issue from the robustness perspective. The analysis is as follows:

	QuadricTri	*Quadric*	*Shortest*
Trained with 4 Polygonal Models	62.71%	62.87%	27.93%
Trained with 5 Polygonal Models	59.53%	58.58%	26.40%

Table 3
Accuracy of verification using *Melax* as reference model

	QuadricTri	*Quadric*	*Malex*	*Shortest*
QuadricTri		90	43	0
Quadric	100		70	0
Malex	40	43		0
Shortest	100	100	100	

(a) Trained with 4 polygonal models out of 44

	QuadricTri	*Quadric*	*Malex*	*Shortest*
QuadricTri		87	30	0
Quadric	100		30	0
Malex	70	40		0
Shortest	100	100	100	

(b) Trained with 5 polygonal models out of 44

Fig. 10. Robustness of verification via various reference models

	QuadricTri	*Quadric*	*Malex*	*Shortest*
QuadricTri		69.10		
Quadric	78.76			
Malex				
Shortest	32.86	33.18	35.12	

(a) Trained with 4 polygonal models out of 44

	QuadricTri	*Quadric*	*Malex*	*Shortest*
QuadricTri		67.69		
Quadric	60.61			
Malex				
Shortest	33.13	32.69	33.62	

(b) Trained with 5 polygonal models out of 44

Fig. 11. Effectiveness of verification for robust combinations.

While we use the classification technique to alleviate the test oracle problem, we also need to maintain the reliability of the mechanism to avoid false positive results. Readers may recall that a false positive case refers to the misclassification of a passed test case as a failure. As a result of such misclassifications, testers would spend unnecessary effort in reviewing test cases that should not warrant any manual involvement in the first place. In a typical situation, most test cases do not reveal failures. Hence, even if a small number passed test cases are misclassified, the percentage of false positives (and, hence, the percentage of wasted manual effort) will be large. It is crucial, therefore, for a technique to achieve a low false positive rate. This is the purpose and usefulness of the robustness measure.

Figure 10 depicts the results of the robustness measure. The four rows represent the reference models used to train the classifier. The four cells in each row represent the percentage of correct verification results for the subject programs indicated in the caption of the column. In Figure 10(a), for example, when the classifier is trained by *QuadricTri* to test the program *Quadric*, the robustness measure is 90%.

We use the Pareto principle (Shulmeyer and McCabe, 1998), also known as the 80-20 rule, as the criterion to determine whether a particular combination in Figure 10

can be described as "robust". [9] For instance, the use of *QuadricTri* as a reference model for testing the program *Quadric* can be described as "robust" because the robustness measure is 90%. The "non-robust" combinations are blacked out in Figure 11.

Since effectiveness is a key metric, we further superimpose the effectiveness of verification via various reference models from Figure 9 onto the robust combinations in Figure 11. The resulting *effectiveness* measures of the *robust* combinations are depicted in the (non-blacked out) cells in Figure 11. We observe the following: The values in the cells for resembling reference models, namely in the four cells for *Quadric* and *QuadricTri*, are significantly higher than the rest in all cases Furthermore, there is no (highlighted) entry for the rows for *Melax*, meaning that it is not robust to use *Melax* as a reference model. The use of a dissimilar reference model will possibly produce many false positives and may, therefore, waste the testers' valuable time trying to resolve whether a test case indeed reveals a failure. Combined with our earlier observation that the use of a resembling reference model gives a higher accuracy than a dissimilar reference model, our result shows that the use of a resembling reference model to train the C4.5 classifier is an accurate, effective, and robust means of obtaining a test oracle.

5.3 Further Discussions

In summary, our empirical results show that, if testers know a resembling reference model to the current program under test, they should use it as to training a classifier to serve as an automatic oracle. This reinforces a common practice by mesh simplification developers to use resembling reference models to manually check the outputs of their implementations.

On the other hand, if the testers are unaware of a resembling reference model, they should consider the use of a less sophisticated reference model. This will give a test report with a low number of false positive cases. The tradeoff in such case is to achieve accuracy at the expense of effectiveness.

To train or test a classifier, our previous work (Chan et al., 2006a) uses the test cases of a program under test and the dynamic behaviors shown by the executions of these test cases to train a classifier. The present work considers the problem from an orthogonal perspective. It extracts features from the outputs of a resembling reference model of the program under test. Potential future work is to integrate the two approaches and determine whether the errors in one approach complement those of the other effectively.

[9] See, for example, the empirical study (Gittens et al., 2005) of the effect of the Pareto principle on the distribution of software faults.

The results of the present case study also reconfirm in part the techniques in Chan et al. (2006a) and Bowring et al. (2004). The use of a subject program to train a classifier can be considered as a special case of using a resembling reference model to train a classifier. Our result reconfirms that their approaches are effective and robust. Our result may also reveal a common weakness in their work, namely, that they are less accurate than the use of a simple dissimilar reference model to identify failures. It is interesting to study other innovative applications of machine learning to directly improve on these techniques.

5.4 Threats to Validity

In this section, we discuss potential threats to the validity of our experimental case study.

The outputs of mesh simplification software are graphics files. We have used a popular feature extraction technique to obtain generic classification patterns to train the C4.5 classifier as an automatic pseudo-oracle. One may, of course, use other feature extraction techniques or the entire images for training and testing. We are also aware from the machine learning community that the selection of useful features plays a central role in the effectiveness of a classifier. Our experiments will serve as an initial benchmark for subsequent research studies.

All the subject programs use openGL to render graphics. They do not represent other types of rendering API. The latter may produce distinct behaviors and, hence, distinct sets of mutants. This might affect the accuracy, effectiveness, and robustness measures. It would be interesting to find out the extent that other rendering algorithms may impact the results.

Our experiments were conducted on a set of 44 open-source 3D polygonal models. They include models that graphically show a chair, a spider, a teapot, a tennis shoe, a weathervane, a street lamp, a sandal, a cow, a Porsche sports car, and an airplane. They have been studied intensively and details are available from the Web to help us render the graphics using the four mesh simplification algorithms described earlier. There are also other ways to represent the inputs for graphical rendering. The relationship between the use of polygonal input and other types of input in the context of software testing remains to be explored. On the other hand, the size of the present study is fairly extensive. We have already done our best to conduct the experiments, which uses 10 machines to execute the programs continuously for two months. We believe that it realistically represents the testing of mesh simplification programs in real-life situations.

We have only conducted experiments on four implementations of mesh simplification algorithms. We have assumed that these implementations are of high quality. The generalization of the proposal warrants more research. Also, our work is built on top of the C4.5 classifier. Although it is an extremely important

and classical tool, there are other classifiers that may be used. From the pattern recognition community, we are aware that the use of a large training dataset may override the effects due to different classifiers. The use of a reference model allows testers to generate training datasets of different sizes. However, the effect of the sizes of datasets on the performance of the proposal remains to be further studied.

MuJava is used to generate program mutants of the reference models in our experiments. The tool may only produce particular types of program mutant that may affect the analysis of the experimental results. Andrews et al. (2005) find that the use of mutation operators can yield trustworthy results. Kapoor (2006) proves that the coupling hypothesis of mutation testing holds in many logical fault classes, and further establishes a fault class hierarchy for logical faults with Bowen (Kapoor and Bowen, 2007). On the other hand, developers may produce complex faults in a program that simple program mutants may not strongly couple with. One potential way to complement our methodology is to extract the faults from the repository of mesh simplification and simulate them as faulty versions of a reference model. We leave the evaluation of such a strategy to evaluate PAT as future work.

In the experiments, we regard *Quadric* and *QuadricTri* to be more similar to each other and less similar to the other two implementations. This is based on a general observation that *QuadricTri* is directly modified from *Quadric*and preserves the basic algorithmic skeleton of the latter, while more distinct philosophies are used in the development of the algorithms for *Melax* and *Shortest*. To alleviate any risk from our observation, we run these four subject programs on the 3D polygonal models to produce simplified graphics at 70%, 40%, and 10% simplification ratios. We then manually and visually reviewed the outputs to spot the differences to confirm our intuitive division of resembling and dissimilar subject programs.

6 Conclusion

Systems with rendering engines to produce computer graphics are an important class of software applications. They usually use polygonal models to represent the graphics. Mesh simplification is a vital technique to vary the levels of details of rendered graphics and, hence, improve the overall performance of the rendering process. As such, the quality of its implementation affects that of the graphics rendering application. Although reference models may be available, software developers implement their own solutions because of other functional and non-functional constraints. The testing of these implementations is essential.

To identify failures, testers need to compare the actual outputs from test executions with the expect results. For graphics software, however, a pixel-by-pixel comparison is ineffective and slow. Furthermore, determining the expected graphics at the pixel level is by itself a complex process that warrants testing.

In this paper, we have proposed PAT, a novel fault-based approach to alleviating the test oracle problem. It uses a resembling reference model to train a pattern classifier to identify failures of other implementations. The experimental results show that PAT is promising. It suggests that the reference model should preferably be a system resembling the program under test; otherwise, the reference model should be a generic system for achieving robustness.

We also envisage the study of a "similarity measure" to refine the results. Moreover, we plan to further improve on the effectiveness, accuracy, and robustness of the use of resembling or generic types of reference model. We will further explore the use of metamorphic approaches (Chen et al., 1998, 2002) for testing mesh simplification software.

References

Ahmed, M. N., Yamany, S. M., Mohamed, N., Farag, A. A., Moriarty, T. 2002. A modified fuzzy c-means algorithm for bias field estimation and segmentation of MRI data, IEEE Transactions on Medical Imaging 21 (3), 193–199.

Andrews, J. H., Briand, L. C., Labiche, Y. 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). ACM Press, New York, pp. 402–411.

Baresi, L., Denaro, G., Mainetti, L., Paolini, P. 2002. Assertions to better specify the Amazon bug. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002). ACM Press, New York, pp. 585–592.

Bass, L., Clements, P., Kazman, R. 2003. Software Architecture in Practice, Addison Wesley, Reading, Massachusetts.

Beiden, S. V., Maloof, M. A., Wagner, R. F. 2003. A general model for finite-sample effects in training and testing of competing classifiers, IEEE Transactions on Pattern Analysis and Machine Intelligence 25 (12), 1561–1569.

Berstel, J., Reghizzi, S. C., Roussel, G., San Pietro, P. 2005. A scalable formal method for design and automatic checking of user interfaces, ACM Transactions on Software Engineering and Methodology 14 (2), 124–167.

Bierbaum, A., Hartling, P., Cruz-Neira, C. 2003. Automated testing of virtual reality application interfaces. In: Proceedings of the Eurographics Workshop on Virtual Environments, ACM Press, New York, pp. 107–114.

Bowring, J. F., Rehg, J. M., Harrold, M. J. 2004. Active learning for automatic classification of software behavior, in Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), ACM SIGSOFT Software Engineering Notes 29 (4), 195–205.

Chan, W. K., Cheng, M. Y., Cheung, S. C., Tse, T. H. 2006. Automatic goal-oriented classification of failure behaviors for testing XML-based multimedia software applications: an experimental case study, Journal of Systems and Software 79 (5), 602–612.

Chan, W. K., Cheung, S. C., Ho, J. C. F., Tse, T. H. 2006. Reference models and automatic oracles for the testing of mesh simplification software for graphics rendering. In: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006), Vol. 1. IEEE Computer Society Press, Los Alamitos, California, pp. 429–438.

Chen, T. Y., Cheung, S. C., Yiu, S. M. 1998. Metamorphic testing: a new approach for generating next test cases, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

Chen, T. Y., Tse, T. H., Zhou, Z. Q. 2002. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing, in Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), ACM SIGSOFT Software Engineering Notes 27 (4), 191–195.

Cignoni, P., Rocchini, C., Impoco, G. 1998. A comparison of mesh simplification algorithms, Computers and Graphics 22 (1), 37–54.

d'Ausbourg, B., Seguin, C., Durrieu, G., Roch, P. 1998. Helping the automated validation process of user interfaces systems. In: Proceedings of the 20th International Conference on Software Engineering (ICSE '98). IEEE Computer Society Press, Los Alamitos, California, pp. 219–228.

Dillon, L. K., Ramakrishna, Y. S. 1996. Generating oracles from your favorite temporal logic specifications, in Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '96/FSE-4), ACM SIGSOFT Software Engineering Notes 21 (6), 106–117.

Duda, R. O., Hart, P. E., Stork, D. G. 2000. Pattern Classification, Wiley, New York.

Francis, P., Leon, D., Minch, M., Podgurski, A. 2004. Tree-based methods for classifying software failures. In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004). IEEE Computer Society Press, Los Alamitos, California, pp. 451–462.

Freund, Y., Schapire, R. E. 1996. Experiments with a new boosting algorithm. In: Proceedings of the 13th International Conference on Machine Learning. Morgan Kaufmann, San Francisco, California, pp. 148–156.

Garland, M., Heckbert, P. 1997. Surface simplification using quadric error metrics. In: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97). ACM Press, New York, pp. 209–216.

Gittens, M., Kim, Y., Godwin, D. 2005. The vital few versus the trivial many: examining the Pareto principle for software. In: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), Vol. 1. IEEE Computer Society Press, Los Alamitos, California, pp. 179–185.

Gonzalez, R. C., Woods, R. E. 2002. Digital Image Processing, Prentice Hall, Englewood Cliffs, New Jersey.

Hierons, R. M. 2006. Avoiding coincidental correctness in boundary value analysis, ACM Transactions on Software Engineering and Methodology 15 (3), 227–241.

Hosmer, D. W., Lemeshow, S. 2000. Applied Logistic Regression, second edition.

Chapter 5. Wiley, New York.
Kapoor, K. 2006. Formal analysis of coupling hypothesis for logical faults, Innovations in Systems and Software Engineering 2 (2), 80–87.
Kapoor, K., Bowen, J. P. 2007. Test conditions for fault classes in Boolean specifications. ACM Transactions on Software Engineering and Methodology 16 (3), article 10.
Kohavi, R., Provost, F. 1998. Glossary of terms. Machine Learning 30 (2/3), 271–274.
Last, M., Friedman, M., Kandel, A. 2003. The data mining approach to automated software testing. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003). ACM Press, New York, pp. 388–396.
Luebke, D. P. 2001. A developer's survey of polygonal simplification algorithms, IEEE Computer Graphics and Applications 21 (3), 24–35.
Luebke, D. P., Erikson, C. 1997. View-dependent simplification of arbitrary polygonal environments. In: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97). ACM Press, New York, pp. 199–208.
Luebke, D. P., Reddy, M., Cohen, J. D., Varshney, A., Watson, B., Huebner, R. 2003. Level of Detail for 3D Graphics, Morgan Kaufmann, San Francisco, California.
Ma, Y.-S., Offutt, A. J., Kwon, Y.-R. 2005. MuJava: an automated class mutation system, Software Testing, Verification and Reliability 15 (2), 97–133.
Mayer, J. 2005. On testing image processing applications with statistical methods. In: Software Engineering 2005 (SE 2005), Lecture Notes in Informatics. Gesellschaft fu"r Informatik, Bonn, pp. 69–78.
Mayer, J., Guderlei, R. 2004. Test oracles using statistical methods. In: Proceedings of the 1st International Workshop on Software Quality (SOQUA 2004) and the Workshop on Testing Component-Based Systems (TECOS 2004) (in conjunction with Net.ObjectDays 2004), Lecture Notes in Informatics, Vol. 58. Springer, Berlin, pp. 179–189.
Melax, S., November 1998. A simple, fast, and effective polygon reduction algorithm, Game Developer Magazine, 44–49.
Memon, A., Banerjee, I., Nagarajan, A. 2003. What test oracle should I use for effective GUI testing? In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003). IEEE Computer Society Press, Los Alamitos, California, pp. 164–173.
Memon, A. M., Pollack, M. E., Soffa, M. L. 2000. Automated test oracles for GUIs. In: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2000/FSE-8). ACM Press, New York, pp. 30–39.
Meyer, B. 1992. Eiffel: the Language, Prentice Hall, New York.
Nixon, M. S., Aguado, A. S. 2002. Feature Extraction and Image Processing, Elsevier, Amsterdam.
Ostrand, T., Anodide, A., Foster, H., Goradia, T. 1998. A visual test development

environment for GUI systems, in Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98), ACM SIGSOFT Software Engineering Notes 23 (2), 82–92.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B. 2003. Automated support for classifying software failure reports. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003). IEEE Computer Society Press, Los Alamitos, California, pp. 465–475.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P. 1992. Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, Cambridge.

Quinlan, R. 1993. C4.5: Programs for Machine Learning, Morgan Kaufmann, San Francisco, California.

Richardson, D. J., Thompson, M. C. 1993. An analysis of test data selection criteria using the RELAY model of fault detection, IEEE Transactions on Software Engineering 19 (6), 533–553.

Segal, M., Akeley, K. 2004. The OpenGL Graphics System: a Specification. Version 2.0, Silicon Graphics, Mountain View, California.

Shulmeyer, G. G., and McCabe, T. J. 1998. The Pareto principle applied to software quality assurance. In: Handbook of Software Quality Assurance, third edition. Prentice Hall, Upper Saddle River, New Jersey, pp. 291–328.

Sun, Y., Jones, E. L. 2004. Specification-driven automated testing of GUI-based Java programs. In: Proceedings of the 42nd Annual Southeast Regional Conference (ACM-SE 42). ACM Press, New York, pp. 140–145.

Takahashi, J. 2001. An automated oracle for verifying GUI objects, ACM SIGSOFT Software Engineering Notes 26 (4), 83–88.

Vanmali, M., Last, M., Kandel, A. 2002. Using a neural network in the software testing process, International Journal of Intelligent Systems 17 (1), 45–62.

Weiss, G. M. 2004. Mining with rarity: a unifying framework, ACM SIGKDD Explorations 6 (1), 7–19.