Reference Models and Automatic Oracles for the Testing of Mesh Simplification Software for Graphics Rendering *

W. K. Chan and S. C. Cheung
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{wkchan, sccheung}@cse.ust.hk

Jeffrey C. F. Ho and T. H. Tse †
The University of Hong Kong
Pokfulam, Hong Kong
{jcfho, thtse}@cs.hku.hk

Abstract

Software with graphics rendering is an important class of applications. Many of them use polygonal models to represent the graphics. Mesh simplification is a vital technique to vary the levels of object details and, hence, improve the overall performance of the rendering process. It progressively enhances the effectiveness of rendering from initial reference systems. As such, the quality of its implementation affects that of the associated graphics rendering application. Testing of mesh simplification is essential towards assuring the quality of the applications. Is it feasible to use the reference systems to serve as automated test oracles for mesh simplification programs? If so, how well are they useful for this purpose?

We present a novel approach in this paper. We propose to use pattern classification techniques to address the above problem. We generate training samples from the reference system to test samples from the implementation. Our experimentation shows that the approach is promising.

Keywords: *Test oracles, software testing, mesh simplification, graphics rendering, pattern classification reference models.*

1. Introduction

Computer graphics components are crucial in various real-life applications. Two examples are medical imaging [1] and graphics-based entertainment. Many of these components use polygonal models to render graphics, as illustrated in Figure 1(a), because of mathematical simplicity and efficient hardware or software support [19, 21]. For interactive graphics-based software, such as the two examples above, it is important to be responsive to the environment. Slow rendering of graphics is, therefore, unacceptable. A mainstream technique to alleviate this problem is known as *mesh simplification* [10, 19, 21]. It transforms a given three-dimensional (3D) polygonal model to one that has fewer polygons but resembles the original shape and appearance as much as possible. Figure 1 from http://www.melax.com/polychop/lod_demo.zip shows three examples of mesh simplification, in which a Beethoven statue and two apples are modeled by different numbers of polygons. The number of polygons to model the same statue or apple decreases gradually from left to right, yielding a progressively coarser image as a result.

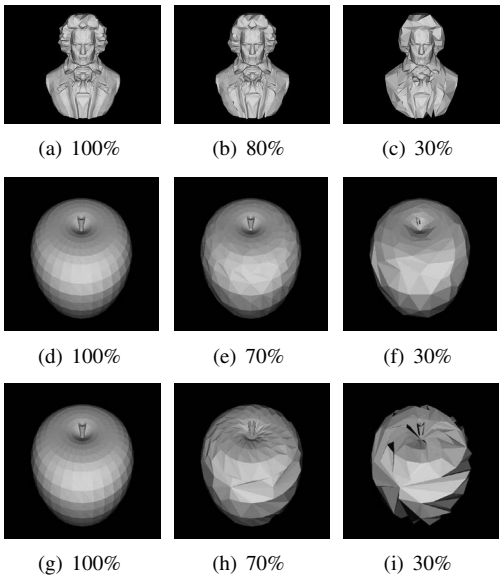


Figure 1. Mesh simplification of polygonal models of a Beethoven statue (from top row), a properly rendered apple, and a badly rendered apple.

A *test oracle* is a mechanism against which testers can check the output of a program and decide whether it is correct. Is there a convenient oracle for testers of mesh simplification software? Software developers normally develop their own mesh simplification programs

* This research is supported in part by a grant of the Research Grants Council of Hong Kong (project no. HKU 7145/04E) and a grant of The University of Hong Kong.

† All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2557 8447. Email: thtse@cs.hku.hk.

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

because no existing algorithm excels at simplifying all models in all situations [19–21]. The performance and storage requirements of each program may be different, which results in different implementation decisions on various polygon simplification techniques. Since software developers may have some idea about the expected improvements of a particular scenario, they may judge the generated graphics manually by referring to published prototypes of other mesh simplifications. Such a labor-intensive judgment process is subjective, time-consuming, and error-prone. On the other hand, an automatic pixel-by-pixel verification of the graphics output is out of the question. In short, there is a test oracle problem in the testing of mesh simplification software for graphics rendering. It is challenging and useful to find an automatic and convenient test oracle.

It would be interesting to explore whether it is feasible to use a reference program as a model for checking automatically the correctness of the test outputs of another program. Specifically, we would like to study the following question in this paper: *How well does a reference program serve as an automatic test oracle for mesh simplification software?* This paper proposes to construct the test oracle using a pattern classification approach. The experimental results show that our approach is promising.

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 approach to tackling the test oracle problem above. We evaluate our technique via four mesh simplification programs. The evaluation results are discussed in Section 4. Finally, Section 5 concludes the paper.

2. Related Work

In this section, we review related work on the oracle problem for the testing of software with graphical interfaces.

Berstel et al. [4] 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. [11] propose a software environment to include formal operations in the design process of user interface systems. Like the work of Berstel et al. [4], the verification of a GUI specification may then be arranged. Memon et al. [27] 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. This type of approach is intuitive and conventional in the conformance testing of

```
glBegin(GL_QUADS);
 glVertex3f(-1.0f, 1.0f, 0.0f);
 glVertex3f( 1.0f, 1.0f, 0.0f);
 glVertex3f( 1.0f,-1.0f, 0.0f);
 glVertex3f(-1.0f,-1.0f, 0.0f);
glEnd();
```

(a) Method 1

```
glBegin(GL_TRIANGLES);
 glVertex3f( 1.0f,-1.0f, 0.0f);
 glVertex3f(-1.0f, 1.0f, 0.0f);
 glVertex3f( 1.0f, 1.0f, 0.0f);

 glVertex3f(-1.0f, 1.0f, 0.0f);
 glVertex3f( 1.0f,-1.0f, 0.0f);
 glVertex3f(-1.0f,-1.0f, 0.0f);
glEnd();
```

(b) Method 2

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

telecommunication protocols. Sun et al. [35] propose a similar approach for test harnesses. Memon et al. [26] further evaluate different types of oracle for GUIs. They suggest using 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 [34]. 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. [5] also point that not all graphical applications make use of GUI widgets for graphics output.

Bierbaum [5] presents an architecture for automated testing of virtual reality application interfaces. It firstly 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 [36] 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 [36]. This is a type of metamorphic testing [8, 9]. Mayer [23] 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. [30] 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 Ramakrishna [12] discuss a technique to reduce the search space of test oracles constructed from

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

a specification. Baresi et al. [3] propose to use program assertion [28] to check the intermediate states of programs.

More specifically, there are techniques for applying pattern classifications to alleviate the test oracle problems. Last and others [18, 37] 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. [31] classify failure cases into categories. However, they do not study how to distinguish correct and failure behaviors in programs. Their research group further proposes classification tree approaches to refine the results obtained from classifiers [14]. Bowring et al. [6] 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. [7] use classifiers to identify different types of failure-revealing behaviors related to the synchronization of multimedia objects. They do not study the failures within a media object, such as failures in graphics.

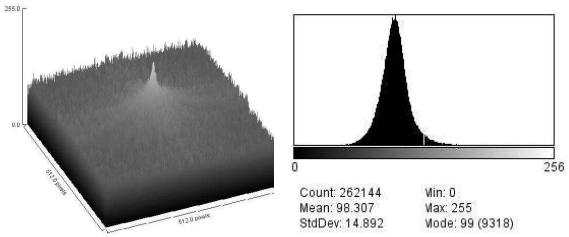
Despite various attempts in related work, none of them considers using a reference program to train classifiers to identify failures of a target program. This is understandable. Even if a reference implementation for mesh simplification software is available, its behaviors will not be identical to those expected for the program under test. Hence, GUI testing techniques may not be applicable to mesh simplification software. As pointed out by Mayer [24], statistical methods are inadequate for effectively identifying failures. Such methods require a formulation of the expected characteristics of the program under test. Studies of adopting pattern classification techniques to software testing indicate that classifiers may be suitable for learning program behaviors. However, the subjects used in the training and testing phases of these studies are normally restricted to programs with single faults, so that the applicability of their results is rather limited.

3. Pattern Classification Technique and Evaluation Study

We propose to use (publicly available) reference systems for automatically identifying failures in the implementations of mesh simplification programs. On the other hand, the implementation of a mesh simplification program may differ a lot from that of a reference system. Their outputs may also be different. We shall evaluate the impact of such differences on our approach.

3.1. Pattern Classification Technique

Software developers of mesh simplification programs frequently modify existing reference algorithms to suit their specific needs. Since existing reference algorithms are available to software developers, we exploit the information available in these reference systems to serve as an automatic oracle for verifying the test results of mesh simplification programs. We propose to use a pattern classification approach. We shall verify the effectiveness of our proposal via an evaluation study.



(a) Frequency representation (b) Histogram

Figure 3. Frequency representation and histogram of contributions of signals for Figure 1(d).

A pattern classification technique [13] 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 of the samples. In the testing phase, the trained classifier assigns a test case to a trained class. For the aim of identifying failures, we restrict our attention to the study of two classes in this paper, namely one with correct behaviors and the other with failure behaviors. In the rest of Section 3, we shall firstly discuss the approach to extracting pattern classification features. We shall then describe the subject programs to be used for the evaluation study. Next, we shall describe the technique for selecting sample data. Finally, we shall describe the experimental procedure for evaluating the pattern classification technique.

3.2. 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. Since the actual shape of a simplified 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 features and the lighting effects.

Classification feature set 1: Change of ratios of major and minor image features. Our first classification feature aims at extracting the amount of major and minor image features that remain in a simplified model for a given percentage of simplification. The level of simplification is normally defined by a simplification percentage, as illustrated in the labels in Figure 1. To identify major and minor image features, we transform images to its

frequency domain through fast Fourier transform [17, 29]. We shall use the notation "$Image_{r\%}$" to represent an image simplified to $r\%$ from the original (which will be denoted as $Image_{100\%}$).

Since rendering an image is relatively computationally expensive, we adopt the advice of Memon et al. [26] to use a *complex* test oracle to trade with the number of test cases. We propose to extract a number of features from the images, which we shall refer to as *image features*. We use them to synthesize classification features, which will serve as the basis for training the classifier for recognition as a test oracle. We extract classification features from different orientations.

We shall describe how to extract classification features from an orientation in the rest of this section. In Section 3.4, we shall describe the selection of different orientations.

We observe that a mesh simplification program may simplify a given model to different levels (such as $Image_{90\%}$, $Image_{80\%}$, ..., and $Image_{10\%}$). We take advantages of this model simplification characteristic to construct our classification feature. We firstly determine a sequence of ratios of major to minor image features for the simplification of the same polygonal model to various levels. We then fit the sequence of normalized ratios of major to minor image features (against the level of simplification) using regression techniques. The coefficients of the fitted curves represent the values of the corresponding classification feature. For a given 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 experiment is ImageJ [2], which uses a simplex method based on [32]. The details for determining a ratio of major to minor image features is as follows.

Ratio of major to minor image features. We firstly extract the amount of signals of the original model $Image_{100\%}$ that remains in a simplified model $Image_{r\%}$. The signals of a model can be obtained from the frequency histogram as shown in Figure 3. We deconvolve $Image_{100\%}$ with $Image_{r\%}$ [17, 29]. The result forms a filter, $Filter_{100\%-r\%}$, representing the transformation from $Image_{100\%}$ to $Image_{r\%}$. A more simplified model will have a higher overall value in the resultant filter. This is because when fewer polygons are suffice to model an image, less amount of image features of the original model remains in the simplified version. Signals with major contributions are basically low frequency signals contributing major image features of $Image_{100\%}$. Signals with minor contributions are basically high frequency signals contributing minor image features of $Image_{100\%}$. Thus, we sort the image features according to signal strengths to facilitate the determination of a threshold, T, for us to extract major and minor contributions passing through a filter $Filter_{100\%-r\%}$.

One popular choice of T is the mean contribution of all signals of $Image_{100\%}$. Other choices include the mean $\pm$ various multiples of the standard deviation. [3] 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 features remains). The other part is responsible for keeping the minor counterpart (that is, how large the portion of minor image features remains). We recall that, as a model is being simplified, a smaller amount of minor image features from the original model will be retained. Major image features are also reduced but usually to a lesser extent. We compute the sum of values of the parts responsible for the major image features, as well as that for the minor image features. The ratio of the two sums, sum_{minor}/sum_{major}, should be progressively smaller as the model is being simplified.

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. For each threshold value, we construct one set of the above coefficients.

Classification feature set 2: Lighting effect. The second set of classification features is concerned with the general lighting of the rendered models. 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 model rendered with extreme brightness or darkness.

3.3. Subject Programs

Our subject programs consist of four different programs, each with a unique mesh simplification algorithm implemented. They are all written in Java. These four algorithms are *Shortest* (shortest edge), *Melax* (Melax's simplification algorithm [25]), *Quadric* [16], and *QuadricTri* (Quadric weighted by the area of the triangles [16]). *Shortest* is one of the simplest mesh simplification algorithms. It always picks the shortest

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

[3] Seven choices of T will be used in the experiment. They will be described later in this section.

edge of a mesh to collapse. *Melax* is 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 firstly removed. *Quadric* is 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. *QuadricTri* is a variant of the *Quadric* algorithm that 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. We note that *Quadric* and *QuadricTri* are two subject programs with roughly resembling algorithms. We say that these two algorithms are *similar*. We also say that other combinations of algorithms in the experiment are *dissimilar*. These two are treated as reference versions of each other. They will help us simulate the situations when software developers adapt an existing algorithm to implement their own versions.

Each program takes two inputs: a file storing the input of a 3D model in standard .PLY format and an integer (0–100) indicating the percentage of polygons in the 3D model that should result from the simplification process. If the value of the input integer is zero, only the background will be shown. We use the black color as the background in our experiment. Each program scales the 3D 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 3D model.

3.4. Test Case Selection

In our approach, there are two classes for pattern classification, namely *passed* and *failed*.

To collect training samples of the *passed* class, we execute a set of 44 3D models[4] over every reference system. The numbers of polygons of these models range from 1,700 to 17,000. In order to better utilize the 3D 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 3D 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 3D model.

To collect training samples for the *failed* class, program mutants are generated from the reference system using a mutation tool known as *mu*Java[5] [22]. We use all the mutants generated from the conventional mutation operators of the mutation tool. Conventional mutation operators have been evaluated to be good indicators of fault detection capabilities of test suites [2]. We have taken a few measures to vet the mutants thus generated in our experiment. Normally, our subject programs take no more than 30 seconds to generate an image from the input of a 3D model. Mutants that take more than 3 minutes to execute an input are considered to have failed to terminate. They will not be used in our experiment. Mutants that produce non-background contents for rendering 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 experiment. Based on these mutants, we collect the classification features from a total of more than 440 000 program executions. Although only 44 models are used, the experimental effort translates into nearly 4 months of non-stop test executions in order to mimic a realistic testing situation.

Shortest	*Melax*	*Quadric*	*QuadricTri*
350	401	1122	1187

Table 1. Numbers of mutants used in subject programs.

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 [38].

3.5. Experimental Procedure

We divide the set of publicly available 3D models (see Section 3.4) into two sets. Set_1 contains 13 models and Set_2 contains the rest. As we shall explain below, the value of 13 is immaterial to our experiment. 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 subject program is treated in turn as the reference implementation, which we shall call $Program_A$. In each case, the three remaining subject programs are treated as implementations under test (IUT), which we shall refer to as $Program_B$. Ten iterations are carried out for the preparation of different sets of training and testing

[4] 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.

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

examples. Three iterations are carried out for each set of training and testing examples.

Classifier. A pilot study with several major categories of classification algorithms has been carried out using sample data. The results indicate that the classifier C4.5 [33] plus the Adaboost M1 boosting method [15] gives the most favorable performance. Hence, we conduct our main experimentation using this classifier. To select 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.

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

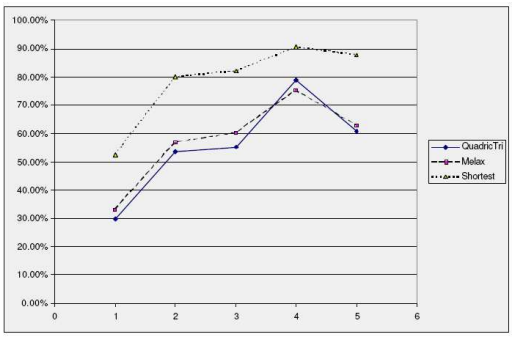
We use the following procedure to train a classifier: N 3D models from Set_1, where $1 \leq N \leq 5$, are randomly selected and executed with mutants of $Program_A$ for every 10% from 100% to 0% to produce training examples of *failed* outputs. These N 3D models, together with the 3D models from the remaining nine-tenth of Set_2, are then input to the original implementation of $Program_A$ 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 models unused in the training stage are input to the original implementation as well as the mutants of $Program_B$ for every 10% from 100% to 0%. We note that all the 3D 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 value of *50%* is chosen because there are two classes in the classification, namely *passed* and *failed*.

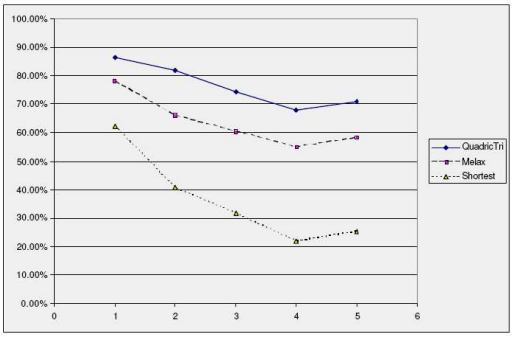
4. Results and Discussions

In the evaluation study, we have used linear, quadratic, and cubic forms of curves for fitting so as to extract classification features (see Section 3.4). We find their results similar and, therefore, discuss only the quadratic case.

Figure 4 shows the results of using *Quadric* as the reference system to identify failures of other subject programs. The horizontal axes show the number of models (N) used for the corresponding training stages. Figure 5 shows the case of using *Shortest* to identify failures of other subject programs.



(a) Effectiveness



(b) Precision

Figure 4. Effectiveness and precision of using a resembling reference system to identify failures.

We firstly define the measures to interpret our findings.

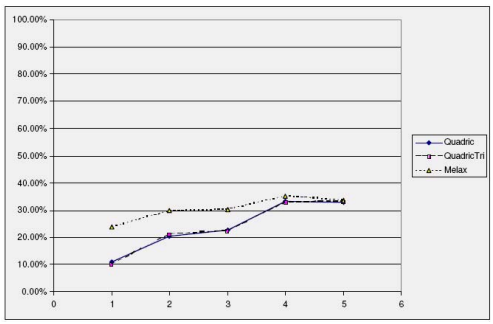
Effectiveness. We use the percentage of mutants killed as the effectiveness measure. It is plotted in the graphs (a) of Figures 4 and 5.

Precision. We use the percentage of test cases being classified correctly as the precision measure. It is plotted in the graphs (b) of Figures 4 and 5.

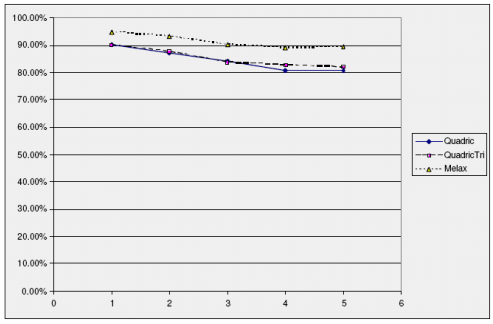
The topmost curve in Figure 4(a) shows that about 70% of the mutants can be killed using the resembling reference system strategy, in which a relatively sophisticated approach is used as the reference system. The result is promising, which confirms the common practice of mesh simplification practitioners to use resembling reference systems to help them judge the outputs of their own implementations.

Furthermore, Figure 4(b) indicates that *Quadric*, which resembles *QuadricTri* but is dissimilar to *Shortest*, can identify a large portion of mutated faults.

On the other hand, if testers use a dissimilar reference system to identify failures, even though it can largely classify outputs correctly (as illustrated in Figure 5(b)), the experiment shows that it kills much fewer number of mutants (as illustrated in Figure 5(a)). The strategy is ineffective. It indicates that, while it is tempting to reuse previous results, testers should only select test cases from



(a) Effectiveness



(b) Precision

Figure 5. Effectiveness and precision of using a dissimilar reference system to identify failures.

resembling reference systems to uncover faults of new implementations.

Would it be better to use the most sophisticated reference system available to test implementations? A comparison of Figures 4(a) and 5(a) indicates that the difference in the sophistication of the mesh simplification approaches has a large impact on the effectiveness. The effectiveness of using *Quadric* as a reference system is much better than that of using *Shortest*. The results of using *QuadricTri* yield very similar graphs (not shown in the paper). However, the percentage of correct classifications in the *passed* class for resembling subject programs (*Quadric* and *QuadricTri*) is only about 87%. It indicates that testers should not aim at using a relatively more sophisticated approach as the reference system. Instead, they should consider using a reference system that is the basis for the current implementation under test.

What if no closely resembling reference system is available? A comparison between Figures 4(b) and 5(b) indicates that the precision deteriorates considerably. Thus, testers should not consider using simpler but dissimilar reference systems as automatic test oracles if resembling reference systems are available. On the other hand, when new mesh simplification approaches are being implemented without any reference system available for training a classifier, testers may consider using simple (intuitive, fundamental, or generic) reference systems such as *Shortest* in our experiment, instead of relatively sophisticated ones such as the *Quadric* in our experiment. It gives a more conservative test report, with a reduced number of false positive cases.

5. 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 object details 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 systems are usually available, software developers implement their own solutions because of other functional and non-functional constraints. The testing of these implementations is essential.

In this paper, we have proposed a novel approach to alleviating the test oracle problem. It uses a *reference system* to train a pattern classifier to identify failures of other implementations. The experimental results show that the approach is promising. It suggests that the reference should preferably be a resembling system that the new implementation improves on; otherwise, the reference should be a generic system. On the other hand, our approach should be less effective if applied to regression testing, because a reference system in a regression setting is likely to contain faults.

Our results are preliminary and, hence, more experiments are warranted. For example, we are using supervised training to train a standard classifier. It is interesting to know whether unsupervised training will give comparable results.

We also envisage the study of a "similarity measure" to refine the results. Moreover, we plan to improve on the precision of using resembling or generic types of reference system. We shall further explore the use of metamorphic approaches [8, 9] for testing mesh simplification software. We have obtained significant initial results and will report them soon.

References

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

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

[3] L. Baresi, G. Denaro, L. Mainetti, and P. Paolini. Assertions to better specify the amazon bug. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge*

Engineering (*SEKE 2002*), pages 585–592. ACM Press, New York, 2002.

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

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

[6] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2004*), pages 195–205. ACM Press, New York, 2004.

[7] W. K. Chan, M. Y. Cheng, S. C. Cheung, and T. H. Tse. Automatic goal-oriented classification of failure behaviors for testing XML-based multimedia software applications: an experimental case study. *Journal of Systems and Software*, accepted for publication.

[8] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 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, 1998.

[9] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*), pages 191–195. ACM Press, New York, 2002.

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

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

[12] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering* (*ISSTA '96*), pages 106–117. ACM Press, New York, 1996.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, New York, 2000.

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

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

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

[17] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 2002.

[18] M. Last, M. Friedman, and A. Kandel. 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*), pages 388–396. ACM Press, New York, 2003.

[19] D. P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21 (3): 24–35, 2001.

[20] D. P. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (*SIGGRAPH '97*), pages 199–208. ACM Press, New York, 1997.

[21] D. P. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, San Francisco, California, 2003.

[22] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon. MuJava: an automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15 (2): 97–133, 2005.

[23] J. Mayer. On testing image processing applications with statistical methods. In *Software Engineering 2005* (*SE 2005*), Lecture Notes in Informatics, pages 69–78. Gesellschaft fu"r Informatik, Bonn, 2005.

[24] J. Mayer and R. Guderlei. 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*), volume 58 of Lecture Notes in Informatics, pages 179–189. Springer, Berlin, 2004.

[25] S. Melax. A simple, fast, and effective polygon reduction algorithm. *Game Developer Magazine*, pages 44–49, November 1998.

[26] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing?. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 164–173. IEEE Computer Society Press, Los Alamitos, California, 2003.

[27] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2000*), pages 30–39. ACM Press, New York, 2000.

[28] B. Meyer. *Eiffel: the Language*. Prentice Hall, New York, 1992.

[29] M. S. Nixon and A. S. Aguado. *Feature Extraction and Image Processing*. Elsevier, Amsterdam, 2002.

[30] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA '98*), pages 82–92. ACM Press, New York, 1998.

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

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

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

[34] M. Segal and K. Akeley. *The OpenGL Graphics System: a Specification*. Version 2.0. Silicon Graphics, Inc., Mountain View, California, 2004.

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

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

[37] M. Vanmali, M. Last, and A. Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17: 45–62, 2002.

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