An Automated Oracle for Verifying GUI Objects

by

Juichi Takahashi

Bachelor of Science
in Production Engineering
Tokai University, Japan
1989

A thesis submitted to Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of
Master of Science
in
Software Engineering

Melbourne, Florida
December 2000

We the undersigned committee hereby recommended that the attached document be accepted as fulfilling in part the requirements for the degree of Master of Science in Software Engineering.

"An Automated Oracle for Verifying GUI Objects"

a thesis by Juichi Takahashi

James A. Whittaker, Ph.D.
Associate Professor, Software Engineering
Thesis Advisor

Cem Kaner, J.D., Ph. D.
Professor, Computer Science
Committee Member

Harold K. Brown, Ph. D.
Associate Professor, Electrical and Computer Engineering
Committee Member

William D. Shoaff, Ph. D.
Associate Professor and Program Chair
Computer Science

ABSTRACT

Title:
An Automated Oracle for verifying GUI objects

Author:
Juichi Takahashi

Thesis Advisor:
James A. Whittaker, Ph. D.

The promise of test automation - the automatic application of software inputs to find bugs - is tempered by the difficult of automatically verifying behavior. Indeed, the lack of tools for behavior verification is a major factor in keeping automated testing out of the mainstream.

This thesis develops a technique that aids automatic behavior verification for a particularly difficult problem: determining the correction of screen output. A methodology to capture and compare screen output is presented and a case study using Microsoft® PowerPoint® is described.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations

ANSI	American National Standard Institute
API	Application Program Interface
CAD	Computer Added Design
DDE	Dynamic Data Exchange
GIF	Graphics Interchange Format
HTML	Hyper Text Markup Language
JPEG	Joint Photographic Expert Group
PC	Personal Computer
RTF	Rich Text Format
XML	eXtensible Markup Language

ACKNOWLEDGEMENT

I wish to thank the members of the thesis committee, Dr. Whittaker, Dr. Kaner, and Dr. Brown who have provided comments, suggestions, and reviews of my work. In particular, I would like to thank Dr. Whittaker for his guidance in completing my thesis.

DEDICATION

To my wife, Yumiko, support, and encouragement while withstanding numerous lost night and weekend as I pursed this research.

Chapter 1

BACKGROUND AND INTRODUCTION

Testing is the process of executing a program with the intent of finding errors.

Glenford J. Myers
"The Art of Software Testing" [Myers79]

A test that reveals a problem is a success. A test that did not reveal a problem was a waste of time.

Cem Kaner, Jack Falk, Hung Quoc Nguyen
"Testing Computer Software" [Kaner93]

Testing is the process that testers use to find bugs. Of course, determining that software works correctly is also part of the testing endeavor. Beizer [Beizer99] mentioned:

We test software in order to: "Validate the object, that is, show that it works. "

Boriz Beizer
"Black-Box Testing"

To find bugs and determine that software works correctly, testers [Whittak2]:

- Input variables
- Input combinations
- Input sequence

Testers are expected to input variable in order that the software works correctly. In addition, testers have to prove not only that the software works with variables for each independent user interface, but also that it works with variables across given software interface to operate the overall software. In the case that there are various inputting combinations, so these combinations may cause defects and be numerous. The inputting sequence also can cause defects. A number of testers are usually very less than that of users, and testers cannot simulate users' input sequence. There are incredible amount of input sequence, which may imply whether software works or fail. Because testers analyze and test all the things above, testing is an expensive endeavor. Whittaker [Whittak3] uses the analogy that using software is akin to walking a jungle path. The test cases applied by testers represent the path. The more testing, the wider the path. The challenge for users is to walk the path by applying only pre-tested input. Applying untested inputs means straying from the path and facing jungle predators. The best thing testers can do for users is to blaze a wide path. This is where automation comes into play: it allows a substantially larger number of tests to be run. Automated testing is a tool for exploring the jungle (and widening the path through it) efficiently and quickly.

1.1 Manual Testing

In the 1970's, there were only a limited number of testing methods [Myers79] and they were mostly executed manually. Manual testing is defined as testing each case by hand, so these expected and unexpected outputs are verified by the faculty of sight (In a later section we will discuss the differences between manual and automated testing costs). We find that the most important issue for manual testing is that people tend to make mistakes. Unfortunately, we do not have any research on how or why people make mistakes. However, it is certain that people make mistakes sometimes during testing [Beizer99][Whittak2]. Testers may type wrong keys or judge a test result incorrectly.

1.2 Automated Testing

Automated testing involves executing test cases and verifying the result programmatically instead of relying on human ability. Automated testing has the ability to reduce testing costs. Research shows automated testing can save up to 80% [Fewst99] [Bach99] of testing costs because automated tests can execute test cases much faster than manual testing.

In the 1960's, Myers said "determining the number of unique logic paths is the same as determining the total number of unique ways of moving from point A to point B (assuming that all decisions in the program are independent from one another). This number is approximately 10^{14} or 100 trillion" [Myers79]. Though there are now many ways to reduce these kinds of test case paths, such as domain analysis [Beizer83][Dalal97], large numbers of test cases are still necessary. The Black box testing has the same story. If 8-alphabetic characters are inputted, there will be 26^{8} test cases for complete coverage. In these circumstances, testers can use automated tests.

Usually when an application is tested using automated testing, there are four phases:

- Design test cases
- Develop the test cases
- Execute the test cases
- Verify results

Each of these phases is discussed below.

1.2.1 Design and Develop Test Cases

The automated test case design concept is similar to manual-based test case design. Almost all the existing test case design techniques, such as, boundary value analysis and path coverage can be used for automated test case design. Yet sometimes testers design test cases for specific automated tasks. For example, if it were arduous to design test cases for a drawing tool, testers would reduce the number of drawing-related test cases or shorten each individual test case. Automated test techniques are still being developed, and some testing techniques cannot be done with automated testing [Dustin991][Kaner97]. In such cases, it is better to use manual testing.

On the other hand, there are certain test cases, which are preferably developed for automation (see section ”When to Automate”). It is suggested that testers or test managers decide which tests should be automated before developing test cases.

1.2.2 Test Case Execution

Test execution means running or exercising test cases to find failures and demonstrate that software functions as specified.

Robert M. Poston [Poston96]
“Automating Specification-based Software Testing”

Automated test execution means coding an automatic input delivery mechanism to replace hands-on-the-keyboard manual testing.

Prior to the 1990s, there were a small number of test execution tools. Many testers were forced to develop their own execution tools. Since then, many software companies have developed a diverse range of test execution tools, such as capture-playback, code coverage, data tracking, and metrics tools. The general idea in automated execution is to take the human out of the test case generation, execution and verification process.

1.2.3 Verification

Test results must be verified during or after test execution. In advance of test executing an expected result is stored in an automated test script. The verification tool then compares the actual result with the stored expected result. When the result is different from the stored expected result, the tool lets testers know that the test case failed. Conversely, when the result is the same as the expected one, the tool tells that the test case passes. Verification matters will be discussed more in a later section.

1.3 Automated Testing Versus Manual Testing

"*Testing is an activity most of us have endured or experienced, and on which we spend a great deal of time and money*"

Bill Hetzel [Hetzel88]
"The Complete Guide to Software Testing"

Since 1960s, testers have struggled with reducing the cost of testing. Clearly, automated testing has the potential to help in this endeavor. In this section, the cost of automated and manual testing will be analyzed.

1.3.1 When to Automate

There are some situations in which automated testing is more expensive than manual testing. Since cost varies from case to case, neither method is universally superior.

Developing an automated test and then running it once costs more than running a single manual test [Maric98]. Manual testing should be used whenever only one test cycle is needed. However, when two or more tests are needed, testers can use either manual or automated testing. Certain types of tests have historically been pursued with automation as described below.

1.3.1.1 Build Verification Test (Smoke Test)

Modern product-build environments are complicated, and the size of source code is much larger than even a decade ago. Consequently, there are often defects that are caused by developers using the wrong version of header files, libraries, and so on. Thus, it is common to briefly test each build in terms of the main functions before delivering the build to testers. This is called a build verification test or smoke test [Dustin992]. In the case that the build verification test fails, the developers will debug immediately and then build again. When the build verification test passes, the build is delivered to testers. Since the build verification test is needed in every build and should be done quickly, there is good reason to consider automating the build verification testing.

1.3.1.2 Regression Test

Regression testing is one of the major areas in which test automation is pursed [Pettic99][Korek98][Kit99]. In general, regression tests are more frequently executed than other tests. Therefore, regression tests are suitable tests to be automated.

Normally, we do not find a lot of bugs during regression testing. However, if a large percentage of bugs are found, managers might question the overall design of the software. Project managers usually count cumulative defects as a test metric [Grady92], yet it is difficult to distinguish re-activated bugs from new bugs without regression testing every build. If we do not execute a regression testing on every build, we might fail to see that the project is out of control. Therefore, testers often develop automated regression tests as early as possible and use the regression bug rate as one of metrics to determine a project's health.

Regression tests are time consuming when the tests are started from beginning of the development phase because designs and requirements often change during that period

[Black99]. But even though automated test development cost is high, finding development problems are well worth the cost.

1.3.1.3 API Testing

API testing is an obvious place to use automated testing [Fewst99][Pettic99][Pettic96], and it is generally straightforward to develop, execute, and verify. Because the verified objects are always text, testers do not have to deal with a complicated graphical output. API automated testing tools are developed by testers using programming languages, such as C language [Jorgen00] or Perl [Pettic99]. Testers also can use GUI oriented commercial tools, such as Rational Robot and Mercury WinRunner [Zambe98] to exercise APIs.

1.3.1.4 Stress /Performance Testing

Stress and performance testing is adaptable for automated testing [Fewst99][Maric98]. For example, in e-mail server testing, testers often need to load the mail server with incoming e-mail. It is difficult to send thousands of messages over a short period of time by hand and to assemble enough human testers to send e-mail for stress/performance test.

1.3.1.5 Internationalization Testing

In international testing, the international characters sometimes cause specific types of bugs. Testers must test an application with a large number of characters variations [Taka00]. In the case that international applications use 3 byte character encoding scheme, the applications use 1,638,400 types of characters. Testers can use automated testing so that a large number of these characters can be tested.

1.3.1.6 Multiplatform Compatibility Testing (Configuration Testing)

In terms of automated tests, the number of tests that can be executed is an issue. Multiplatform compatibility testing is executed many times and can be automated [Pettic99]. Software usually does not directly communicate with hardware but with the operating system. There are sometimes compatibility problems between an operating system and hardware. These problems show up when an application uses functions of the operating system. From the user's point of view, incompatibility problems with the operating system are seen as defects. Thus, application testers need to execute configuration testing to verify the absence of such problems.

In PC application testing, there are hundreds or even thousands of hardware configuration tests that need to be executed if the application rely on hardware performance or use new hardware technology. If the application is for multiplatform, such as Windows, Linux and Unix, the number of test cases increases dramatically. Thus, automated tests are helped when the application has hardware dependency issues or runs on multiplatform. Prior to the development of automated tests, testers must make sure the automated tools support the application running on various platforms [Dustin992].

1.3.2 When Not to Automate

1.3.2.1 Drawing and Image Processing Application Testing

Because capturing and comparing drawing objects are tricky and can easily give the wrong result, graphical based application testing is not usually automated [Maric98][Fewst99][Dustin991]. In this thesis, a reliable and cost-effective form of graphical automated testing is offered for comparing drawing objects.

1.3.3 Current Trends

The current trend is still focused on whether automated testing is too time consuming. It has been estimated that the cost of automating tests can be recovered after reusing the tests two or three projects [Binder99] [Beizer99]. But as software development costs continue to rise, automation is likely to increase in importance over the foreseeable future.

Chapter 2

THE VERIFICATION PROBLEM

"A necessary part of a test case is a definition of the expected output or result".

Glenford J. Myers
"The Art of Software Testing"[Myers79]

Verification of behavior is a key part of the automated testing process. Automated scripts should verify that the application correctly processes input supplied. For verification, we compare the specified behavior of the software with observed behavior to determine if the observed behavior is the same as the specified behavior. The automated test scripts show the test result, which is either pass or fail (there are some observable differences in behavior). For example, when we run an automated test script, which compares a Windows title with a pre-saved Windows title string, an automated verification function tells whether the pre-stored Windows title and the actual title string are equivalent. If the pre-stored Windows caption string is equivalent to actual Windows title string, the test passes.

The best situation is when both execution and verification are automated. In some cases, execution is automatic and verification is manual. This is a non-optimal situation because the main benefit of automated testing is to increase the number of test cases that can be applied. Sitting in front of computer to manually verify test results negates the time saving gained through automated execution.

2.1 Background of Comparison Techniques

One problem with testing desktop applications and web applications is that verifying graphically rendered object is difficult to automate. Automated tools have difficulty fetching and comparing these graphical objects. For example, web applications should handle many types of object controls and images. However most existing automation tools usually have trouble fetching the object's information. These are the same issues testers face when using third party interface controls [Dustin991] and custom controls [Kaner002]. In these cases testers must physically watch the running test case on their computer screen because the tools do not have ability to automatically verify behavior of such objects. Although, there are many objects which cannot be verified by automated tools, some common objects can be handled and verified by tools, such as:

Strings

There are ANSI code, Unicode, and multi-byte character strings. Most automated testing tools have the capability to fetch these objects. There are also font and text formats such as HTML, RTF, and XML. Automated test tools support some of these formats, but not all of them. Before starting the test, testers should make sure the tool can handle specific format requirements for their application.

Files

Testers can easily compare files by operating system commands such as 'diff' and native APIs. Most automated test tools usually have a function for comparing files as well. In addition, testers can confirm if certain file exists, what its permissions are, etc.

Memory Information

Using modern programming languages and operating systems, it is not recommended that developers store data directly into system memory where automated tools cannot get stored data information. But there are some standardized ways to save information into memory, such as the DDE function in Windows, and this data could then be used as a verification object [Rational99].

Menus

The Windows operating system offers a menu system to developers and users. Automated tools treat menus as objects and fetch menu information, such as strings and key accelerations based on the handle assigned to the object by the operating system.

Windows

Window operating systems also offer window-oriented operations. Automated test tools can fetch their size, attribute, caption string, handle, and class information.

Images

Image objects, such as disk or screen images, are stored on the file system. Testers can perform standard binary file comparison to compare such images. In screen images, testers compare between images bit-by-bit. However, in the case that an application uses graphical images, bit-by-bit comparison is the only way to compare them. It is thus difficult to compare actual images with expected images because of storage constraints (the images tend to be large) and time constraints (bit-by-bit comparison is computationally intense).

Although almost all the automated verification tools have the capability of comparing drawing objects, several researchers do not recommend capture/replay automated testing for drawing object because the verification is overly sensitive to any change [Maric98][Fewst99][Dustin991][Kaner97].

Communication Data

Testers can easily compare communication data by treating it as a file. When testers test communication protocols, the application may use a socket to communicate between computers. Testers simply consider the socket to be a storage device and perform the comparison using the file comparing techniques mentioned above.

2.2 Verification for Graphic Object

The previous section gave an explanation of various objects which can be verified with existing tests. An enormous issue for verification is how to compare graphical objects because the bit-by-bit verification method offers limited applicability. In this section, our discussion begins with bit-by-bit and API comparison methods and their benefits and drawback.

2.2.1 Bit-by-Bit Comparison Model

This method compares graphical images bit-by-bit and most commercial tools use this method. The problem with this method is that it is too difficult to compare pre-saved bitmap images with actual bitmap images. For example, some testers save an image with Windows title bar and some testers do not. Consequently, when testers compare the pre-saved bitmap with actual bitmap images, the comparison program may judge that the verification failed because the two objects are not equivalent.

On the other hand, there are other techniques to compare objects intelligently by bitmap image. Various biometric verifications [Pentl00], such as print finger, face [Panka00], signature, and iris are used in the real world. However these methods are still being researched and their algorithms have not yet to be applied to software testing.

2.2.2 API Comparison

In modern commercial operating systems, such as Windows and Unix, applications do not access the CPU or graphic device directly because the operating systems are designed for multi-tasking to protect conflict of shared hardware resources. Therefore, applications use the same system APIs to access hardware. Drawing applications also use APIs to render screen images. We interrupt calls to such APIs and store the information as drawing information (Figure 2.1).

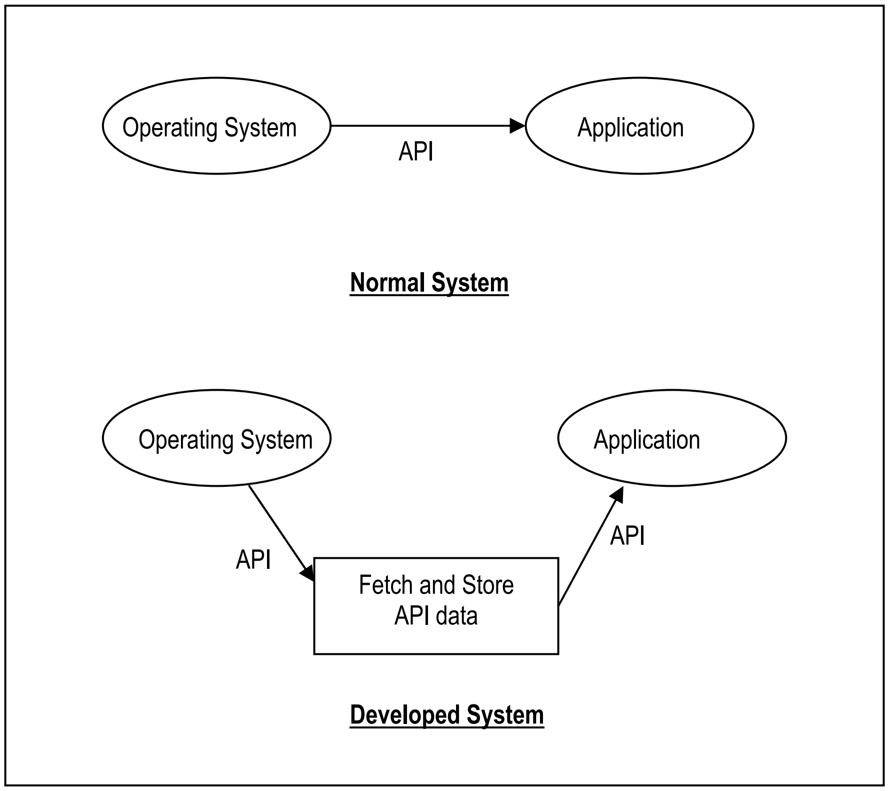


Figure 2.1: API Comparison Model

The benefit of this approach is that an application does not require any special development or operating system hooks. We can test any type of application on any operating system by the API comparison method.

2.3 Benefit of Using the API Comparison Approach

2.3.1.1 System Configuration

Bit-by-bit verification is extremely sensitive to changes [Fewst99]. Various uncontrollable factors can often affect results. For example, it is very difficult to have the same type of machines and graphic systems in a testing team. A tester may have an advanced graphic system, which can display over 1600 x 1200 resolution. Another tester may have a normal graphic system, which displays 1024 x 768. If there are differences between computing environments, the differences can cause failure even when screen images match.

2.3.1.2 Disk Size

When testers store a drawing image as 1600 x 1200 screen resolution at 3 byte per pixel, the image requires 5.76 M bytes (1600 x 1200 x 3). Storing files of this size is acceptable only when a test case is considered strategically crucial. In this case, we might use image-compressing methods to reduce the file size or chosen the API comparison method.

2.3.1.3 Processing Speed

The issues of speed are similar to issues for disk size. Bit-by-bit comparison requires time. For example, comparing 300x300 dot drawing objects, a verification program will substantially calculates 300 times 300, requiring 90000 operations. Because of the same reason in the section 2.3.1.2 (disk size), it is recommended that we use either image compression methods to reduce the calculations or API comparison method.

2.3.1.4 Masking Technique

Testers often use a masking technique to compare results. For example, when the target object includes date or time (Figure 2.2).

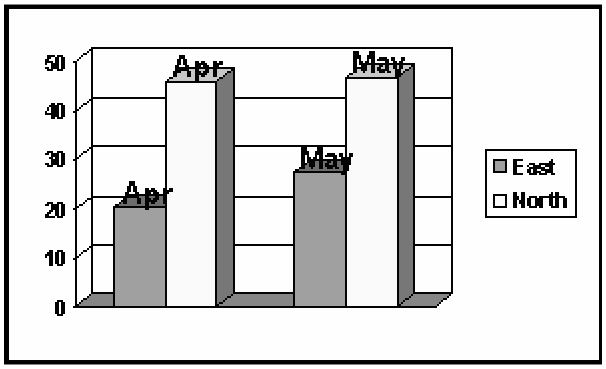


Figure 2.2: Masking Technique 1

When testers want to remove or ignore the date or time when comparing graphical objects (because the time changes as test are run) , they can mask the data or time string (black squares) from the bitmap image by existing tools (Figure 2.3) [Rational99].

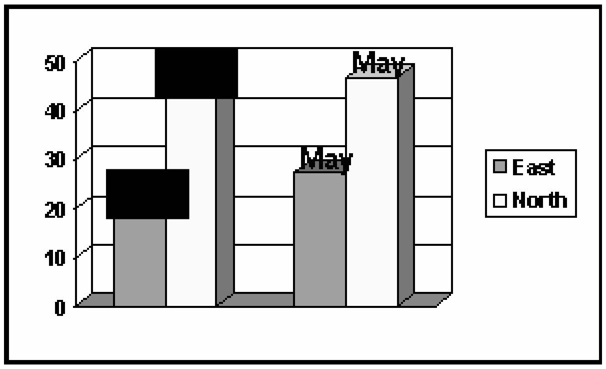


Figure 2.3: Masking Technique 2

However, in this case, not only the text but also parts of target graphical objects are masked. Therefore, automated tests may not verify parts of the object and then by give the wrong result. On the other hand, an API comparison system can simply remove or ignore the time or data string. For example, API comparison method can easily eliminate Win32 TextOut( ) API call from the whole graphical API set. Consequently, testers can fetch the target API calls.

2.3.1.5 Expansion and Reduction Testing

A number of drawing and CAD applications support object expansion and reduction functions. When testing such applications, testers are required to exercise and verify expansion and reduction functions.

Testing expansion and reduction functions using the bit-by-bit method require that we store images for every zoom rate possible. When testing the zoom rate of 1% thorough 100% in increments of 1%, there will be 100 test cases and testers must store 100 bitmaps to verify results. These storage tasks are time consuming and require large amounts of hard drive space. In addition, verification is not straightforward because only small bitmap differences can cause failure of the bit-by-bit comparison techniques.

On the other hand, testers can test any type of expansion and reduction by the API comparison method. Testers do not have to store 100 images to compare the results. Storing only one image is enough to compare the original result and the post-tested result for any degree of zoom rate. More detail is provided in a later section.

2.3.1.6 Oracle Testing

Considering the PowerPoint graph below:

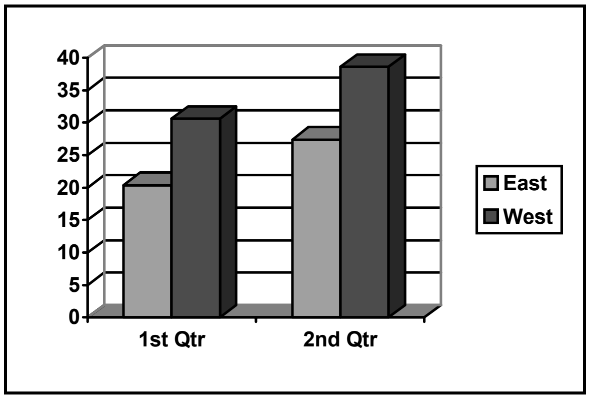


Figure 2.4: Oracle Testing 1

Testers must test a variety of values including this shown Figure in 2.5, 2.6, and 2.6 (boundary test cases).

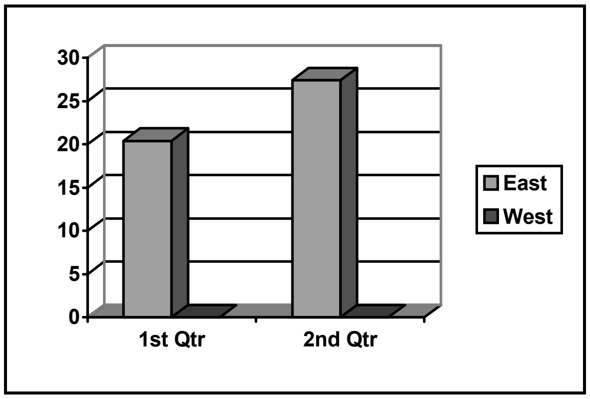


Figure 2.5: Oracle Testing 2

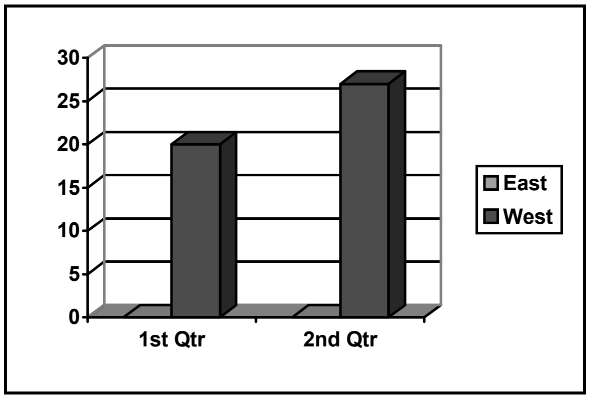


Figure 2.6: Oracle Testing 3

It is easy to see that between such extremes lay a vast number of intermediate tests. Verifying each of this using bit-by-bit comparison is more computationally intense. However, the API comparison method offers a much faster and more reliable solution.

2.4 Limitations of the API Comparison Approach

2.4.1 Bitmap Images

When applications draw graphical objects as bitmap images, the API comparison method does not have any advantage over the bit-by-bit comparison method. It is possible for testers to fetch the bitmap images by using bitmap operation APIs, however the fetching bitmap images operations are same as bit-by-bit comparison method.

2.4.2 Advanced Graphical Controlled Applications

Some applications optimize rendering algorithms based on the performance of various graphic cards and may not benefit from the API comparison method. Because the applications draw graphical objects based on graphical performance and change the drawing APIs to realize best performance, the API comparison method would become unwieldy. Consequently, API comparison method cannot fetch correct API information. For example, when users use 1600 x 1200 graphic cards, some applications will draw larger graphical objects. When user use 640 x 400 graphic cards, the applications draw smaller objects. In this case, API commands are changed by applications in ways not accessible to the API comparison method. Thus, API comparison method may give inaccurate test results.

2.4.3 Graphical Device Dependency Applications

In PC games and other applications, which directly access hardware, API comparison method may not fetch API commands. These applications often directly access graphic cards (direct access to video memory) without using any Win32 APIs to realize higher performance. When the applications do so, API comparison method cannot fetch API commands, which are passed from applications to the operating system. Thus, the API comparison method cannot verify test results.

Chapter 3

A SOLUTION USING API CALL COMPARISON

In this thesis, two kinds of API comparison models are offered to reduce the need for bit-by-bit comparison. The solutions are point-based and vector-based comparison.

In the API comparison technique, testers can use images as logical objects, such as lines, circles, and triangles. As explained in the previous section, the API comparison system will fetch the API command. Drawing a line on screen coordinates (0, 0) to (100, 100), an application calls an API as follows (on Microsoft Windows platform).

```
POINT pPoint[2];
PPoint[0].x = 0;
PPoint[0].y = 0;
PPoint[1].x = 100
PPoint[1].y = 100

LineTo(pPoint);
```

Then the tool fetches the information and stores it for comparison.

3.1 Point-based Comparison

Once an API command from an application is fetched, the API command information is saved as point information to be analyzed and compared.

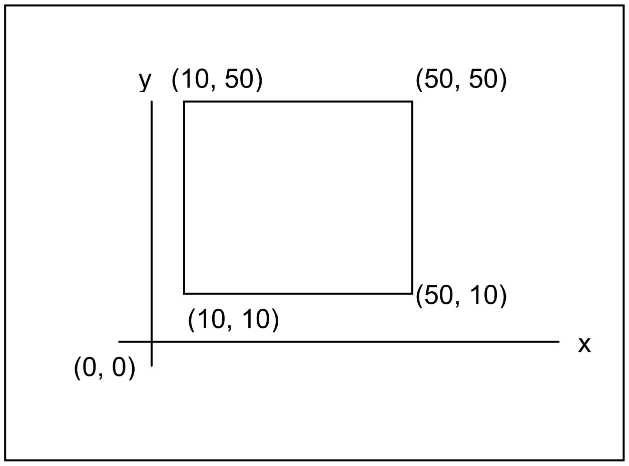


Figure 3.1:   Point-based Comparison 1

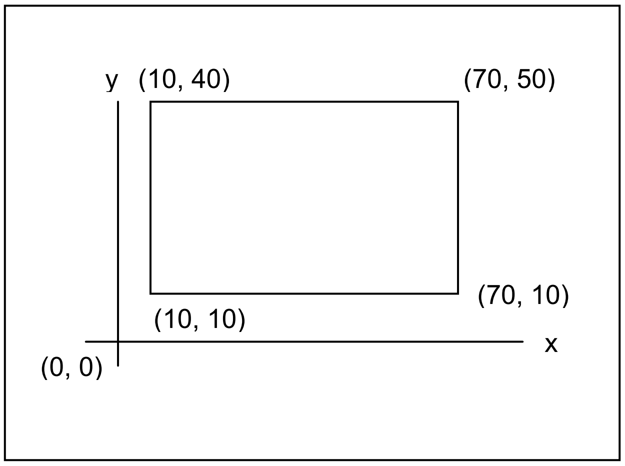


Figure 3.2:   Point-based Comparison 2

For instance, when there is a rectangle (Figure 3.1), and the rectangle coordinates are (10, 10), (50, 40), (10, 10), and (50, 10), coordinates will be compared one by one between objects. The result is Table 3.1:

Table 3.1 Comparison Result

Figure 3.1	**Figure 3.** 2	**Result**
(10, 10)	(10, 10)	**Pass**
(10, 50)	(10, 50)	**Pass**
(50, 10)	(70, 10)	Fail
(50, 50)	(70, 50)	Fail

3.2 Vector Comparison

In a vector comparison system, point information generates vector information, which is structured by length and angle (Figure 3.3).

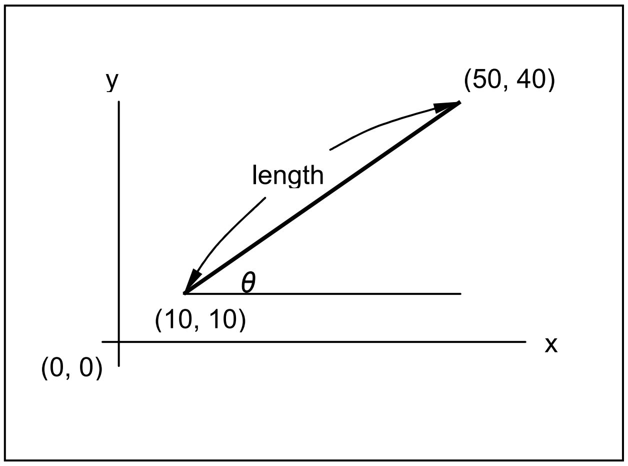


Figure 3.3: Vector Comparison

In Figure 3.3, the vector value (10, 10) through (50, 40) is calculated by:

$$length = \sqrt{(10-50)^2 + (10-40)^2}$$

$$\theta = \sin^{-1}\left(\frac{40-10}{length}\right)$$

The vector comparison technique is useful because testers can easily accomplish expansion and reduction testing, which is explained the "*Expansion and reduction testing*" section above. For example, Figure 3.3 shows the line zoomed out 50% (Figure 3.4)

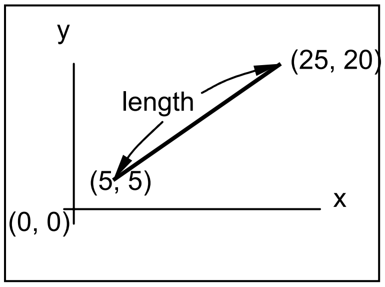


Figure 3.4:   50% Zooming

The length and angle are:

$$length = \sqrt{(10-25)^2 + (10-20)^2}$$

$$\theta = \sin^{-1}\left(\frac{25-10}{length}\right)$$

Even when the application zooms out of 50%, the angle remains the same. The length can also be calculated by:

$$\textit{Original length} = (\textit{Zoom Rate}) \times (\textit{Changed length})$$

Between similar objects, the angles are the same and the lengths are similar. The bit-by-bit comparison method cannot directly compare between similar objects.

Chapter 4

TOOL DEVELOPMENT

To use the API comparison method, a tool has been developed. The tool is capable of:

- Fetching drawing information from API calls made by an application.
- Reproducing rendered objects based on the information from API calls.
- Comparing rendered objects via either point-to-point comparison or vector comparison.

4.1 Design and Development Environment

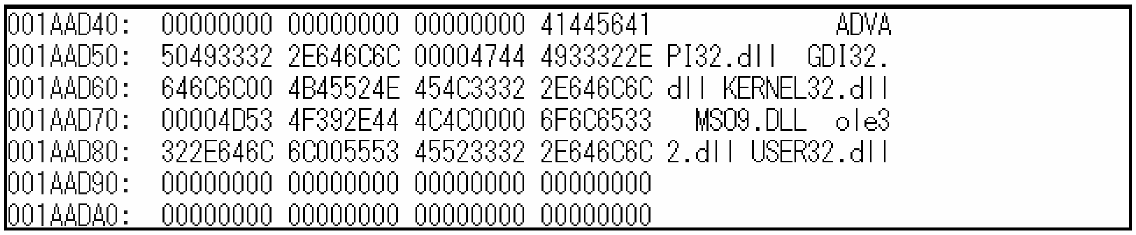
Microsoft® Visual C++® Version 6.0 used for tool development because the C++ programs can directly access memory. Assembly language could be used for the tool development, but the assembly programming is time-consuming. Visual C++ offers in-line assembly code and higher-level C++ syntax. Java, on the other hand, offers limited memory access and we could not find better Java development environment for Windows 2000 than that offered by Visual C++. For the reason, Visual C++ was chosen. All source code in the CD attached to this thesis .

4.2 Microsoft PowerPoint File Structure

PowerPoint uses Microsoft® Office® common components. When drawing graph objects, PowerPoint uses the Office components, which is GRAPH9.EXE.

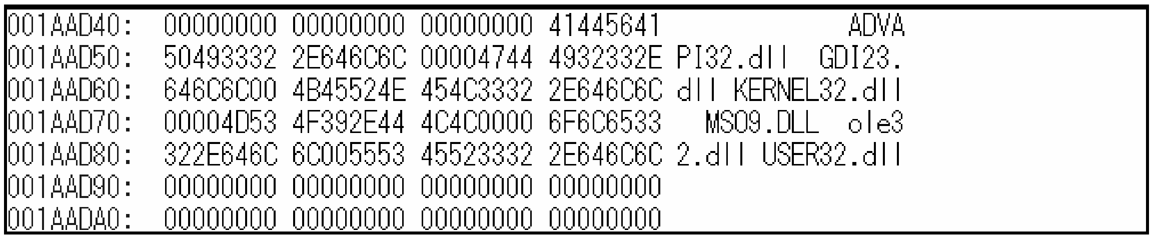
4.2.1 Calling the gdi23.dll

In order to store API information, we developed a program to changes GRAPH9.EXE file to capture the API information. GRAPH9.EXE file is edited using binary editor. Figure 4.1 shows the original GRAPH9.EXE. A GRAPH9.EXE's string is changed from "GDI32.dll" (Figure 4.1) to "GDI23.dll" (Figure 4.2)



```
001AAD40:  00000000 00000000 00000000 41445641             ADVA
001AAD50:  50493332 2E646C6C 00004744 4933322E PI32.dll  GDI32.
001AAD60:  646C6C00 4B45524E 454C3332 2E646C6C dll KERNEL32.dll
001AAD70:  00004D53 4F392E44 4C4C0000 6F6C6533   MSO9.DLL  ole3
001AAD80:  322E646C 6C005553 45523332 2E646C6C 2.dll USER32.dll
001AAD90:  00000000 00000000 00000000 00000000
001AADA0:  00000000 00000000 00000000 00000000
```

Figure 4.1: Original Binary Image for GRAPH9.EXE



```
001AAD40:  00000000 00000000 00000000 41445641             ADVA
001AAD50:  50493332 2E646C6C 00004744 4932332E PI32.dll  GDI23.
001AAD60:  646C6C00 4B45524E 454C3332 2E646C6C dll KERNEL32.dll
001AAD70:  00004D53 4F392E44 4C4C0000 6F6C6533   MSO9.DLL  ole3
001AAD80:  322E646C 6C005553 45523332 2E646C6C 2.dll USER32.dll
001AAD90:  00000000 00000000 00000000 00000000
001AADA0:  00000000 00000000 00000000 00000000
```

Figure 4.2: Changed Binary Image for GRAPH9.EXE

After changing the string, GRAPH9.EXE calls gdi23.dll when PowerPoint uses graphical APIs such as LineTo( ) and Polygon( ).

4.3 Gdi23.dll Development

After editing GRAPH9.EXE, the graphical APIs are passed to the gdi23.dll. Gdi23.dll has two functions: to store API information and to pass the API information to the operating system. As Figure 4.3 shows, the API information goes into gdi23.dll and is passed into gdi32.dll. Consequently, the API information is stored on the hard drive. The stored information is used to compare objects (this comparison will be discussed in a later section). An example of the source code and flow chart are shown in Figure 4.4 and Figure 4.5.

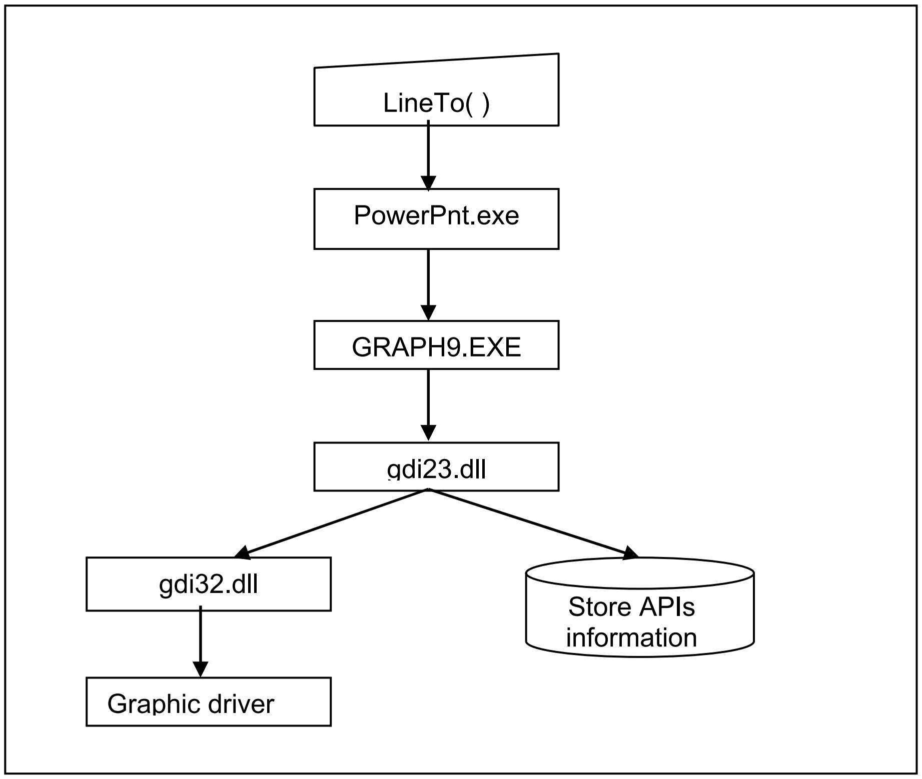


Figure 4.3: Gdi32.dll Behavior Flowchart

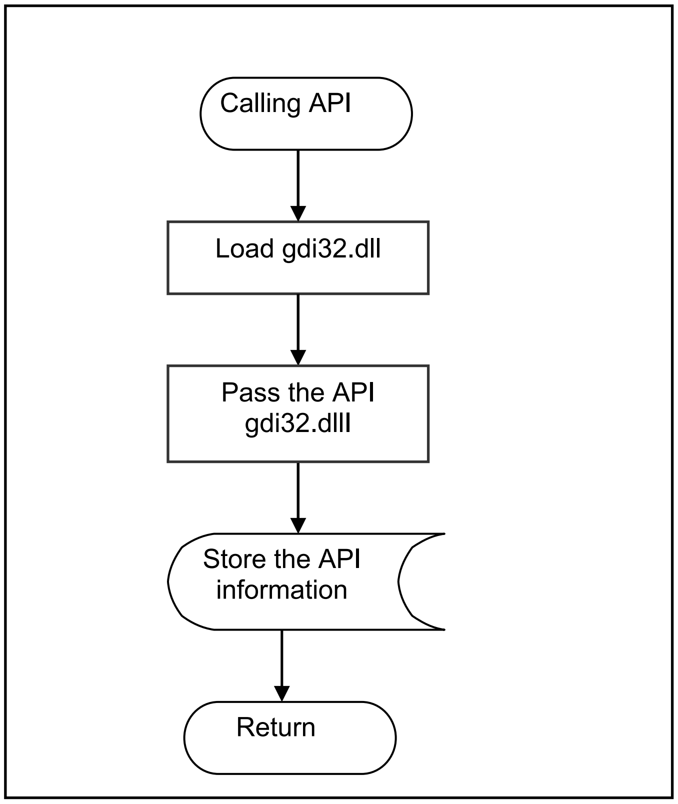


Figure 4.4: Flowchart for Gdi23.dll

```
KERNEL23_API BOOL WINAPI myLineTo(HDC hdc, int iX, int iY)
{
	typedef BOOL (CALLBACK *LPFN)(HDC, int, int);

	HINSTANCE bltH_Dll;
	LPFN bltPtrFn_Function;

	BOOL ReturnValue;

	//Loading Dll
	char ptrChr_DllPath[MAX_STR];
	GetSystemDirectory(ptrChr_DllPath, MAX_STR);
	strcat(ptrChr_DllPath, "\\gdi32.Dll");
	bltH_Dll=LoadLibrary(ptrChr_DllPath);

   //In case failing the library
   _ASSERT(bltH_Dll);
   if (bltH_Dll==NULL)
   {
		ErrorLoading("LineTo");
		return 0;
   }
   bltPtrFn_Function=(LPFN)GetProcAddress(bltH_Dll, "LineTo");
	//In case failing the function
	_ASSERT(bltPtrFn_Function);
	if (bltPtrFn_Function == NULL)
	{
		ErrorLoading("LineTo");
		FreeLibrary (bltH_Dll);
		return 0;
   }
	FreeLibrary (bltH_Dll);
	ReturnValue = bltPtrFn_Function(hdc, iX, iY);
	LogFile2("LineTo(*hdc,", iX, iY, LINETO_LOG);

	return ReturnValue;
}
```

Figure 4.5: Sample Program for Gdi23.dll

4.4 Main Control Program

To make graphical object comparisons, a main control program was developed. An interface is shown at Figure 4.6.

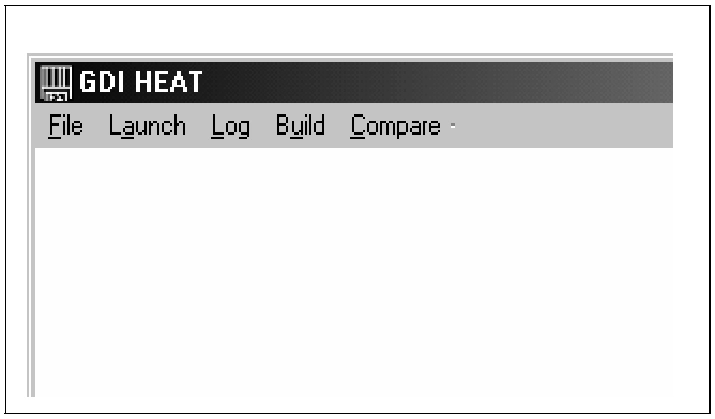


Figure 4.6: Main Screen

The main control program has three major functions: fetching rendered information, regenerating graphical objects, and comparing graphical objects. Its detailed operations are in Appendix C.

4.4.1.1 Fetching Rendered Information

As explained in an earlier section, gdi23.dll can store API information. However, when gdi23.dll stores information from all graphical API's, a large number of duplicated API calls are stored on the hard drive. Thus, the main control program restricts the outcome of gdi23.dll API calls.

In the Windows operating system, when part of or all screen images are required to redraw because windows move or resize, the operating system sends WM_PAINT messages to an application so that the application redraws graphical objects. In some cases the operating system may keep sending WM_PAINT messages, and gdi23.dll may keep this information. Consequently, it becomes difficult to compare original and test result objects because there is duplicated information. The main control program is programmed only to fetch unique API information. In order to do that, the main control program does two things:

- Close and resize an application window forceably.

- Command to start and stop logging API's information.

When an application forceably closes and resizes, the Windows operating system sends a WM_PAINT message. The main control program only fetches unique graphical API information. However, some applications redraw graphical objects without receiving WM_PAINT messages and the operating system may send WM_PAINT message without any reason. In this case, the main control tool may receive the same API messages multiple times. Thus, the main control program commands gdi23.dll only to store information for a very short time between starting a close and finishing resizing the window.

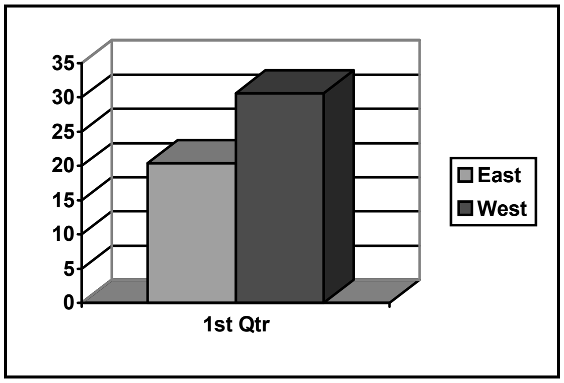


Figure 4.7:   Fetching Information

Looking at the object in Figure 4.7 object, the tool could fetch the points of the polygons (Figure 4.8).

Polygon 1	*(34, 172), (53, 158), (248, 158), (229, 172), (34, 172)*
Polygon 2	*(34, 172), (34, 22), (53, 8), (53, 158), (34, 172)*
Polygon 3	*(53, 158), (53, 8), (248, 8), (248, 158), (53, 158)*
Polygon 4	*(248, 158), (229, 172), (34, 172), (53, 158), (248, 158),*
Polygon 5	*(34, 172), (34, 22), (53, 8), (53, 158), (34, 172)*
Polygon 6	*(53, 158), (53, 8), (248, 8), (248, 158), (53, 158)*
Polygon 7	*(132, 172), (132, 85), (150, 70), (150, 158), (132, 172),*
Polygon 8	*(76, 172), (76, 85), (132, 85), (132, 172), (76, 172),*
Polygon 9	*(132, 85), (150, 70), (95, 70), (76, 85), (132, 85),*
Polygon 10	*(187, 172), (187, 41), (206, 27), (206, 158), (187, 172)*
Polygon 11	*(132, 172), (132, 41), (187, 41), (187, 172), (132, 172)*
Polygon 12	*(187, 41), (206, 27), (150, 27), (132, 41), (187, 41)*

Figure 4.8:   Sample Stored Information

4.4.1.2 Regenerating Graphical Object

The tool also has a regenerating function. After the tool fetches rendered information, the information reforms it into C++ code: "LineTo(10, 10, 50, 50)", and it binds it to a prepared C++ file to render the object data to the screen. The bound file is then compiled and linked to show the regenerated objects. For example, the main program can generate a rendered object (Figure 4.9) from the stored information (Figure 4.8).

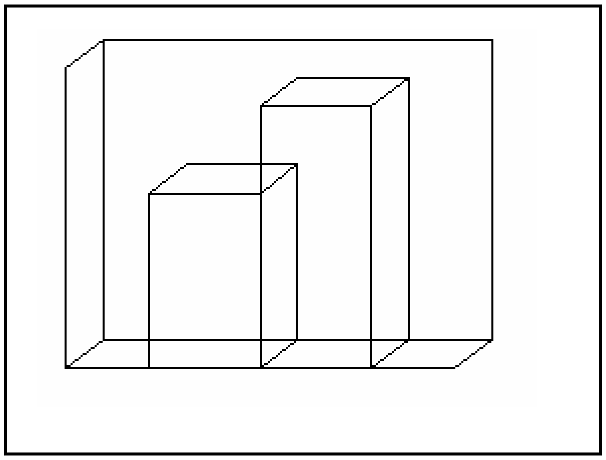


Figure 4.9: Regenerating Points Information

4.4.1.3 Comparing Graphical Objects

In order to compare an original object and a test result, the main control program has a comparison function. After the main control program stores the original information and the test result, it binds the original program and the test result and prepares a C++ program that includes APIs comparison routine. And then the bound file is compiled and linked to compare the original rendered object and the test result (Figure 4.10).

The comparison program is developed in C++ and implemented in an object oriented fashion. A root class is the whole graph object and inherits to child classes such as LineTo, MoveTo, and Polygon class (Figure 4.11). An example program (Figure 4.12) shows both a root class (Graph class) and a child class (Polygon class). Additionally, this object oriented design can expand to other APIs without any major design changes.

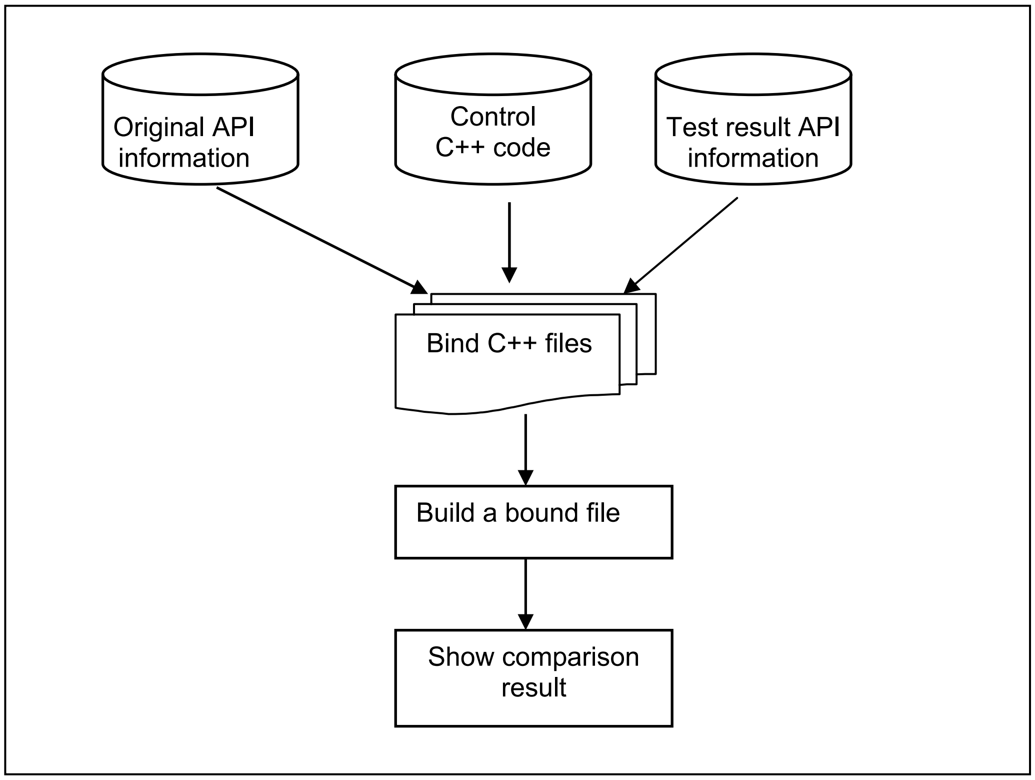


Figure 4.10: Flowchart for Gdi23.dll

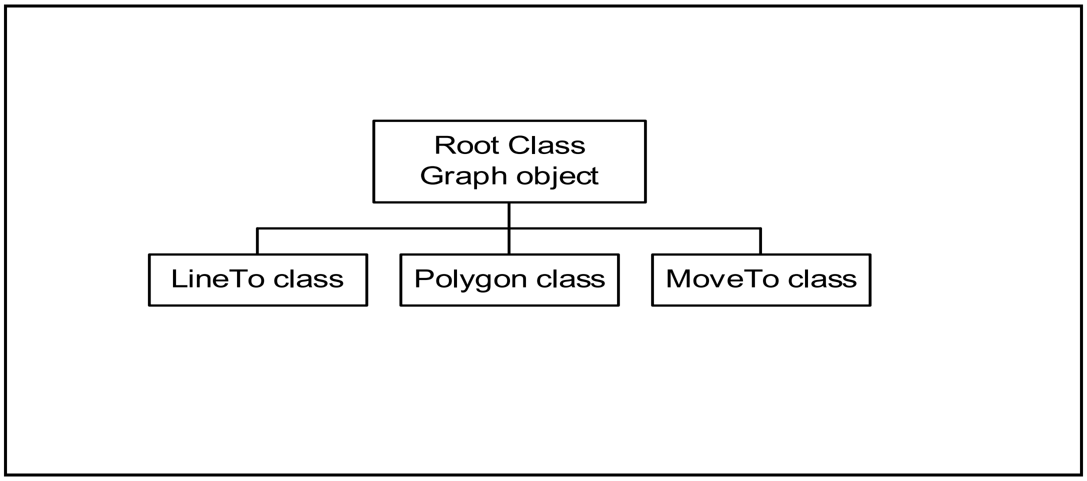


Figure 4.11: Class Structure

```
/************************************************************
 *     Polygon Class
 ************************************************************/
class CPolygon
{
private:
       POINT  *pPoint;     //Points information
       float  *pLength;    //Length information
       float  *pAngle;     //Angle information
       int    iNum;        //Number of polygon

public:
       CPolygon(void);
       int    GetNumber(void){return iNum;}
       POINT* GetPoints(void){return pPoint;}
       float* GetLength(void){return pLength;}
       float* GetAngle(void){return pAngle;}
       void   Add(POINT *point, int num);
       char   Compare(POINT *plygon, int num, float
                            *dstAngle, float *dstLength);
};
/************************************************************
 *     Graph Object Class
 ************************************************************/
class CGraph : CPolygon{
private:
       int    iNumLine;    //Number of lines
       CPolygon     *cPolygon;   //Polygon object pointer
       int    iNumPolygon;//Number of polygon
       int    iNumPolygonNotSameAsPoint;//Not same polygon  No.
       int    iNumPolygonNotSameAsAngle;//Not same angle No.
       int    iNumPolygonNotSameAsLength;//Not same length No.
public:
       CGraph(void);
       void AddLine(int ixstart, int iystart, int ixend, int iyend);
       void   AddPolygon(POINT *point, int num);
       CPolygon* GetPolygon(int LineNo);
       int GetPolygonNumber(void){return iNumPolygon;}
       int GetSamePolygonNumber_Point(void);
       int GetSamePolygonNumber_Angle(void);
       int GetSamePolygonNumber_Length(void);
       BOOL CompareLine(CGraph *graph);
       BOOL ComparePolygon(CGraph *graph);
};
```

Figure 4.12:   Sample Program for Gdi23.dll

Chapter 5

AN EXAMPLE: MICROSOFT POWERPOINT

Microsoft PowerPoint is used as a sample application to confirm the API technique. PowerPoint is one of the most popular graphical presentation applications and is complicated enough to demonstrate the API comparison technique working on a real application.

In this chapter, we will examine which types of graphical objects PowerPoint uses and which types of bugs may be attributed to test object. Furthermore, to fully realize API comparison, a tool is developed that implements the method on the Win32 platform. The tool has many advanced abilities for comparing graphical objects. A case study applying the tool to PowerPoint is conducted and the results are reported.

5.1 Graphic Objects

In this section, we will examine the types of graphical objects used by PowerPoint.

Line, triangle, and rectangle

Most drawing objects are composed using simple line objects. A triangle is made up of 3 connected lines, a rectangle is 4 lines, and a polygon is 4 or more lines. It is possible to fetch and compare these objects by the API comparison method because the APIs contain the information that describes the lines that compose the graphical object.

Bitmap fonts

Text is displayed by using font or text APIs. When an application uses text-handling API, any commercial automation testing tools can fetch text information. Yet, for a variety of reasons, developers sometimes choose not to use text-handling API to display text. In such cases, bitmap fonts are used so that text information is rendered a bitmap instead of using text-handling APIs. When it is stored as a bitmap, existing commercial automation testing tools cannot fetch text information. The only way to fetch such information is to load and store the information as a bitmap object. .

Image object

Because a large number of multimedia applications and tools were developed over a decade ago, testers are required to test a large number of image types such as bitmap, GIF, and JPEG.

5.2 Type of Graphical Bugs

Software failure is categorized into improperly constrained input, improperly constrained stored data, improperly constrained computation, and improperly constrained output that API compassion confined [Whittak1]. Based on Whittaker's idea, we can categorize the types of graphical bugs. These are improperly constrained computing coordinate, improperly computing color information, and improperly constrained output.

5.2.1 Improperly Constrained Computing Coordinate

One common bug is that drawing objects are often rendered at the wrong coordinate. Consider the following bug report:

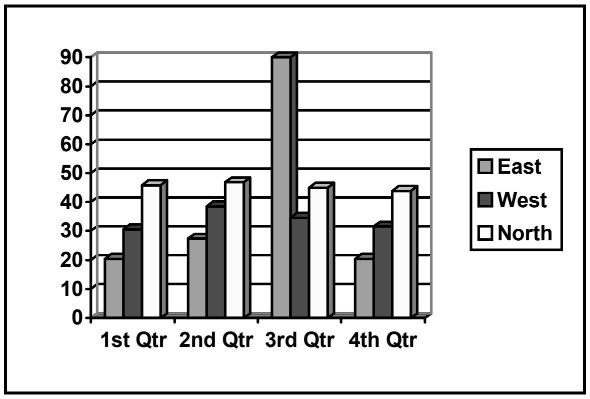


Figure 5.1: Pre-saved Bitmap Image

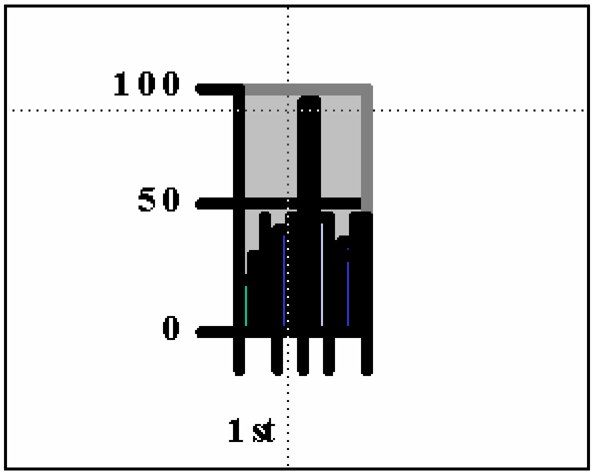


Figure 5.2: Actual Bitmap Image

Bug Report:

Application: Microsoft PowerPoint 2000

Steps:

1. *Launch Microsoft PowerPoint*
2. *Insert a graph (Figure 5.1).*
3. *Minimize the graph*

4. Click at another location (not on the graph)

5. Restore the original size

Expected Result:

The graph is shown as original image (Figure 5.1)

Result:

The graph is shown with corruption (Figure 5.2)

5.2.2 Improperly Computing Color Information

Applications use graphical APIs to show objects with color information. First, the operating system receives APIs calls from the application and then calculates the color information based on specific capabilities of the installed graphic card. Next, the operating system passes the information to a driver for the graphic card. The problem is that users tend to have vastly different graphic device with varying ability to display colors. Therefore, it is difficult to develop an application to adjust to all of these graphic systems. Another problem is that drawing a graphic uses multiple software modules and hardware, including: the application, operating system, graphic card drivers, and graphic card hardware (Figure 5.3). Consequently, bugs may be in the application, may come from the operating system, the graphic driver or any combination thereof.

Although improperly constrained computing coordinate bugs may have the same type of the problems, which are not from the application, it does not often happen because the range of the typical monitor is simply 0 through a couple of thousand but the range of color is from black and white through over millions of colors.

5.2.3 Improperly Constrained Output

It is obvious that some graphic driver software and hardware have defect. However some bugs are not from the graphics driver and it is difficult to find the bug and the cause of the defect. As explained above, to output graphic image, the data, which application made, go through application, API, operating system, graphic driver, and graphic card software (Figure 5.3)

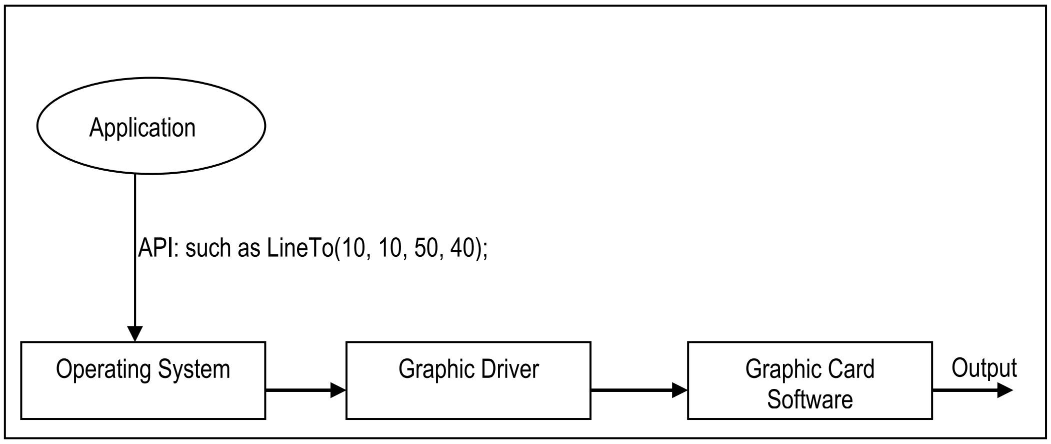


Figure 5.3: Graphical Object Output

Interface bugs are between the application and API; API and operating system; operating system and graphic driver; graphic driver and graphic card software. Therefore, even when testers find a bug, which is graphically related, it is not easy to find the cause of defect. To find the cause of defects, testers often attempt to test using another graphic card and operating system. Though this work is time consuming, there is no other way to distinguish the application's bugs from others bug. In addition, both API comparison and bit-by-bit comparison cannot find this type bugs.

5.2.4 Testing

Testing is executed by another tool written for PowerPoint. The target object is one of PowerPoint Graph's functions (Figure.5.4). The Graph's object is mainly structured in terms of polygons. To simplify the analysis and results, only the polygon object information is tested by the tool.

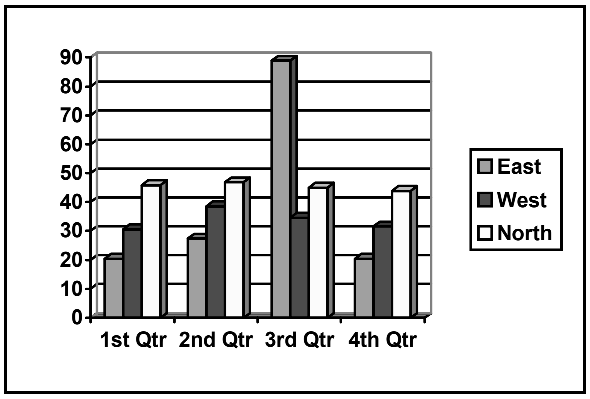


Figure 5.4: Original Object

5.2.4.1 Point Comparison

In the point comparison method, two objects (Figure 5.4 and Figure. 5.5) are compared. The Figure 5.5 is only changed at 1st qtr East value from the original object (Figure 5.4). The comparison program is supposed to show that only one polygon is different. The following is a test case:

Case 1: Compare two slightly different objects

1. *Launch PowerPoint*
2. *Insert graph object (Figure 5.4)*
3. *Save the object as original object*
4. *Change a value 1st quarter East to "90" (Figure 5.5)*
5. *Save the object as destination object*

Expected result: The comparison program shows differences between objects

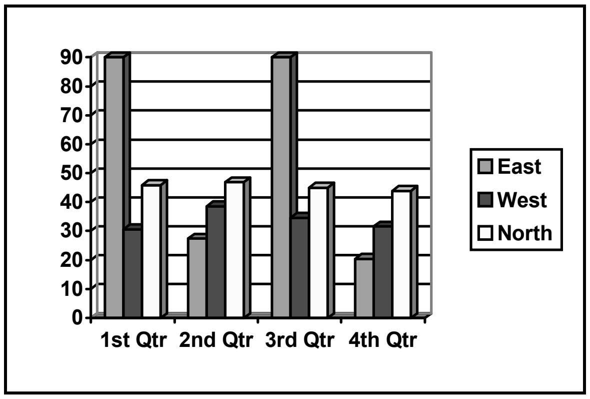


Figure 5.5: Modified Object

Running on the Figure 5.4 objects, the tool fetches and stores Figure 5.6's point information as original information.

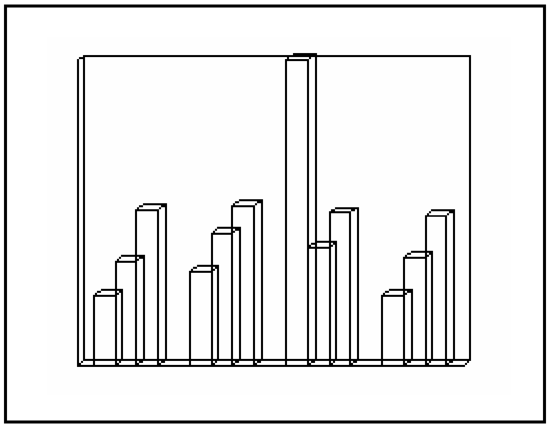


Figure 5.6: Regenerated Object (Original)

Running on the Figure 5.5 object, the tool fetches and stores Figure 5.7 point information as destination information.

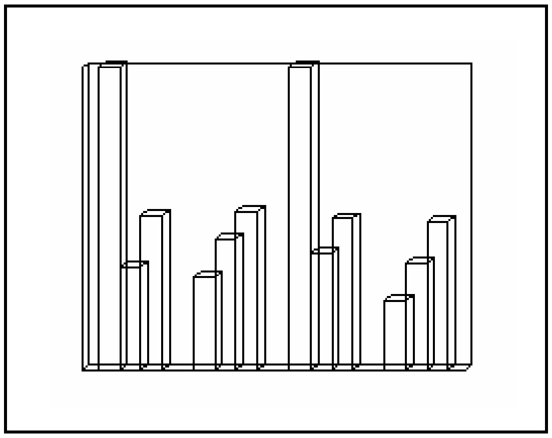


Figure 5.7: Regenerated Object (Modified)

Test Analysis:

- 42 polygons are detected
- 2 polygons have different point values
- 9 lines have different values
- 168 points are detected
- All detailed results are in Appendix A.

Test Result: PASS

In this case, there are 42 polygons. Comparing between Figures 5.4 and 5.5, the program detects two polygons that are different, this includes 9 lines that are different. Therefore, the tool can detect the difference as expected (more details are described in Appendix A).

5.2.4.2 Vector Comparison

To illustrate vector comparison, 2 test cases are performed below.

***Case 1**: Compare two slightly different objects*

1. *Launch PowerPoint*
2. *Insert graph object(Figure 5.8)*
3. *Save the object as original object*
4. *Change a value 1st quarter East to "90"(Figure 5.5)*
5. *Save the object as destination object*

Expected result: The comparison program shows the difference between objects

***Case2**: Compare two similar objects.*

1. *Launch PowerPoint*
2. *Insert graph object(Figure 5.8)*
3. *Save the object as original object*
4. *Zoom 33%(Figure 5.10)*
5. *Save the object as destination object*

Expected result: The comparison program shows objects are not the same but similar.

Case1 shows that the vector comparison method works to detect the object differences as the point comparison method did above. ***Case 2*** confirms that the vector comparison method distinguishes an object from a similar object. PowerPoint has a zoom function, which can change object viewing from 10% through 400%. Even when the zoom rate is changed, the comparison process can detect that they are similar objects.

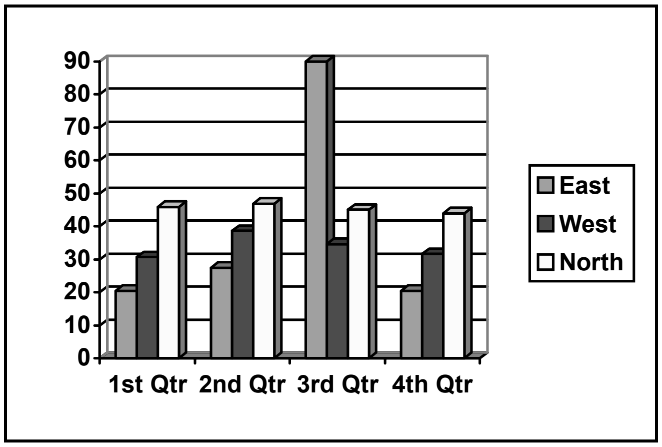


Figure 5.8: Original Object for Case2

The tool fetches and stores the rendered objects point's information (Figure 5.9) from Figure 5.8 as an original object.

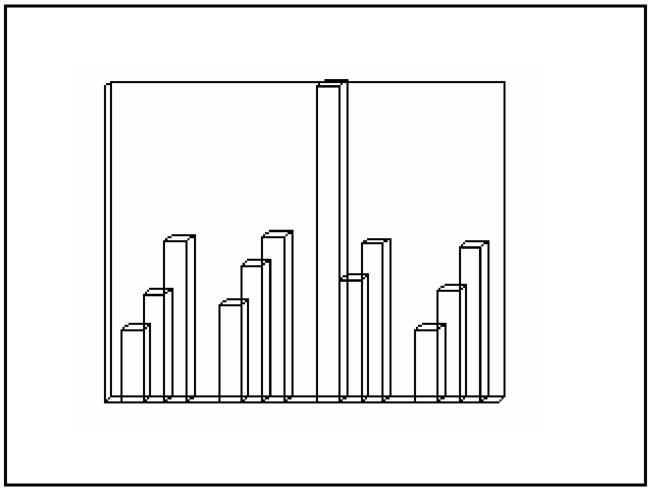


Figure 5.9: Regenerated Original Object for Case2

The tool fetches and stores point's information (Figure 5.11) from Figure 5.101 as a destination object.

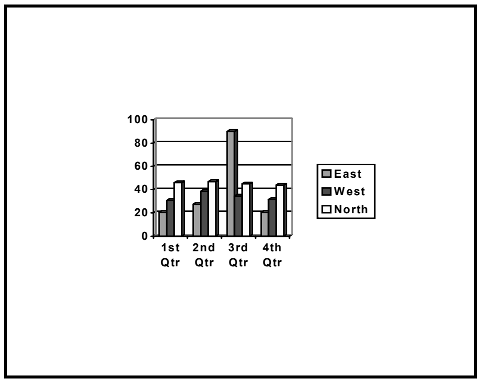


Figure 5.10: Destination Object for Case2

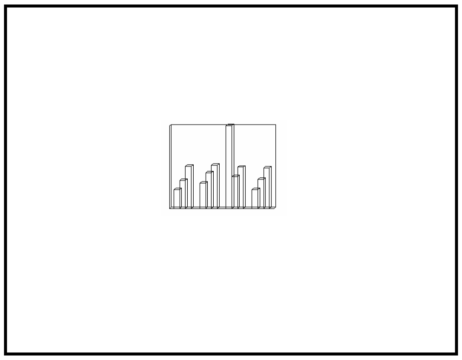


Figure 5.11: Regenerated for Destination Object for Case2

Test Analysis:

- 42 polygons are detected
- 168 lines are detected
- 168 angles are same between original and destination object
- 168 vector lengths have same rate between original and destination object.
- All details are described in Appendix B.

Test Result: PASS

The comparison program shows objects are not same but similar because:

- All angles are the same.
- All points are different.
- Original and destination lengths have the same proportion.

Thus, the tool can distinguish between objects with similar characteristics.

Chapter 6

CONCLUSION

6.1 Summary of This Thesis

The thesis has succeeded in providing the concept that API comparison method can be useful comparing graphical objects automatically. This is a great advancement for automated verification work of what has traditionally been a manual intensive-process. Though testers have struggled to compare bitmap and drawing images automatically, we can now offer an alternative method to verify rendered images using API call comparison.

This research has demonstrated the basic steps to accomplish API comparison techniques:

Point-based comparison was demonstrated on Microsoft PowerPoint to compare each point in a rendered polygon.

Vector-based comparison was also demonstrated on Microsoft PowerPoint to allow more advanced comparison of vectors that polygon points represent. This allows more advanced behavior like scaling on object.

The use of a real application like Microsoft PowerPoint was chosen to show that the technique work on actual retail applications.

6.2 Future Work

This thesis demonstrated innovative approach to automatically verifying rendered screen object. A number of extensions are possible.

Graphical object comparison with color Information: In this thesis, color information was not considered in the comparison of graphical objects. However, since color information is contained the API call information, it is available to this technique.

Random and smart monkey testing: For PowerPoint and other graphical applications, it was demonstrated that testers could verify rendered objects by the API comparison method. To expand this method, we could possibly perform random and monkey tests by the API comparison method [Nyma00]. A major problem with random testing and monkey testing for drawing object is manual verification of the test result. Testers now have the API comparison method and can execute random and smart monkey with automate verification.

Integration with commercial automated tools: During this research, a tool comparing drawing objects was developed. Consequently, it is natural that the tool can be integrated into commercial automated testing products like Rational Robot, Rational Visual Test, and Mercury WinRunner.

Web application testing: Web application testing is much more important than it was a decade ago. Every year, web software is incorporating many existing objects such as JPEG, MPEG, and XML. Testers can potentially use the API comparison technique for comparing these objects.

Testing print functions: Printing is a very complex test because it is difficult to compare between printed objects and screen objects image by automation. Especially, drawing objects are difficult to compare. For example, CAD applications require mathematical accuracy for matching between printed and screen images. The API comparison model can offer the possibility of accuracy by automating comparisons between printed and screen images. For example, Win32s API represents a line as:

```
LineTo(10, 10, 50, 50)
```

And Postscripts language represents the same line as:

```
10 10 moveto
50 50 lineto
```

Therefore, it is possible to compare Postscript and screen object based on information contained in the API call stream.

Operational testing: Beta testing and field testing are important and difficult because ordinary customers do not have the skill to report bugs. Thus, capturing information from user session would be valuable way to capture debugging information. Steven has stated [Steave00] that the Java based operational testing method captures various input information better than commercial tools such as Rational Robot and Visual Test do.

It is clear that the API comparison method can help diagnose a customer facing a problem because the API comparison method is not only capable of fetching the output information, it can also fetch input information. The most difficult issue when the problem is found on the customer side, is the difficultly to distinguish between a general bug and a specific configuration bug. In case of finding a configuration bug, input capturing information is not enough to reproduce the defect. Both output information in addition to input information will aid in problem reproduction and diagnosis.

REFERENCES

[Bach99] Bach, James. “Test automation snake oil.” *14th International conference on Testing Computer Software*, US Professional Development Institute, June 1997.

[Beizer99] Beizer, Boris. *Black-Box Testing*, John Wiley & Sons, Inc, New York, 1995.

[Beizer83] Beizer, Boris. *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1983.

[Binder99] Binder, Robert V. *Testing Object-Oriented Systems*, Addison Wesly Longman, Massachusetts, 1999.

[Black99] Black, Rex. *Managing the Testing Process*, Redmond WA, Microsoft Press, 1999.

[Dalal97] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. “The AETG System: An Approach to Testing Based on Combinatorial Design.” *IEEE Transaction on Software Engineering*, Vol. 23, Issue 7, July 1997, pp. 437-444.

[Dustin991] Dustin Elfriede. “Lessons in Test Automation” *Software Testing & Quality Magazine*, September/October 1999 <http://www.stqemagazine.com/featured.asp?id=6>.

[Dustin992] Dustin, Elfriede, Rashka Jeff., and Paul, John. *Automated Software Testing*, Addison-Wesley, 1999.

[Fewst99] Fewster, Mark. *Software Test Automation*, Addison Wesley, New York, 1999.

[Grady92] Grady, Robert B. *Practical Software Metrics for Project Management and Process improvement,* Pentice Hall, New Jersey, 1992.

[Hetzel88] Hetzel, Bill. *The Complete Guide to Software Testing*, John Wiley & Sons, New York, 1988.

[Jorgen00] Jorgensen, Alan. *Class lectures on Advanced Software Testing*, Florida Institute of Technology, May 2000.

[Kaner93] Kaner, Cem. Falk Jack., and Nguyen, Hung Quoc. *Testing Computer Software*, International Thomson Computer Press, MA, 1993.

[Kaner97] Kaner, Cem. "Improving the Maintainability of Automated Test Suites." *Quality Week '97*, 1997.

[Kaner001] Kaner, Cem. *Class lectures on the Software Test and Quality*, Florida Institute of Technology, Sep 2000.

[Kaner002] Kaner, Cem. *Architectures of Test Automation*, Class handout on the Software Test and Quality, Florida Institute of Technology, Sep 2000.

[Kit99] Kit, Edward. "Integrated, Effective Test Design and Automation." *Software Development Magazine*, Feb 1999, pp. 27-41.

[Korek98] Korel, Bogdan., and AI-Yami, Ali M. "Automated Regression Test Generation." *International Symposium on Software Testing and Analysis*, March 1998.

[Maric98] Brian Marick. "When should a Test Be Automated?" *International Software Quality Week*, May 1998.

[Myers79] Myers, Glenford J. *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

[Nyma00] Nyman, Noel. "Using Monkey Test Tools." *Testing and Quality Magazine*, Vol. 2, Issue 1, Jan/Feb 2000, pp 18-26

[Panka00] Pankanti, Sharath., and Bolle, Rund M. "Biometrics: The Future of Identification." *IEEE Computer*, Vol. 33, Issue 2, February 2000, pp. 46-49.

[Pettic96] Pettichord, Bret, "Success with Test Automation." *International Software Quality Week*, 1996

[Pettic99] Pettichord, Bret. "Seven Steps to Test Automation Success." *STAR West*, November 1999.

[Pentl00] Pentland, Alex S., and Choubury, Tanzeem. "Face Recognition for Smart Environments." *IEEE Computer*, Vol. 33, Issue 2, February 2000, pp. 50-55.

[Poston96] Poston, Robert M. *Automating Specification-Based Software Testing,* IEEE Computer Society Press, CA, 1996.

[Rational99] *Using Rational Robot Release 7.5*, Rational Software Corporation, MA, 1999.

[Steave00] Steven, John. “jRapture: A Capture/Replay Tool for Observation-based Testing” *International Symposium on Software Testing and Analysis*, 2000.

[Taka00] Takahashi, Juichi. “Is Special Software Testing Necessary Before Releasing Products to an International Market?” *International Quality Week*, June 2000.

[Whittak1] Whittaker, James., and Jorgensen, Alan. *Why Software Fails*, <http://se.fit.edu/ papers/SwFails.pdf>.

[Whittak2] Whittaker, James A. “What Is Software Testing? And Why Is It So Hard?” *IEEE Software*, Vol. 17, Issue 1, January-February 2000, pp. 70-79.

[Whittak3] Whittaker, James A. *Class lectures on the Software Test & Quality*, Florida Institute of Technology, September 1999.

[Zambe98] Zambelich, Keith. *Using GUI-based Automated Test Tools to Test Legacy Applications* <http://www.sqa-test.com/w_paper2.html>.

APPENDIX A

In this appneix, test results for point-based verification, which is discussed in chapter 5, is showed.

Table A.1 Point-based Verification 1

Polygon No.	Original	Destination	Result
1	(40, 172)	(40, 172)	Pass
	(43, 169)	(43, 169)	Pass
	(236, 169)	(236, 169)	Pass
	(233, 172)	(233, 172)	Pass
	(40, 172)	(40, 172)	Pass
2	(40, 172)	(40, 172)	Pass
	(40, 19)	(40, 19)	Pass
	(43, 17)	(43, 17)	Pass
	(43, 169)	(43, 169)	Pass
	(40, 172)	(40, 172)	Pass
3	(43, 169)	(43, 169)	Pass
	(43, 17)	(43, 17)	Pass
	(236, 17)	(236, 17)	Pass
	(236, 169)	(236, 169)	Pass
	(43, 169)	(43, 169)	Pass
4	(236, 169)	(236, 169)	Pass
	(233, 172)	(233, 172)	Pass
	(40, 172)	(40, 172)	Pass
	(43, 169)	(43, 169)	Pass
	(236, 169)	(236, 169)	Pass
5	(40, 172)	(40, 172)	Pass
	(40, 19)	(40, 19)	Pass
	(43, 17)	(43, 17)	Pass
	(43, 169)	(43, 169)	Pass
	(40, 172)	(40, 172)	Pass
6	(43, 169)	(43, 169)	Pass
	(43, 17)	(43, 17)	Pass
	(236, 17)	(236, 17)	Pass
	(236, 169)	(236, 169)	Pass
	(43, 169)	(43, 169)	Pass

Table A.2 Point-based Verification 2

Polygon No.	Original	Destination	Result
7	(59, 172)	(59, 172)	Pass
	(59, 137)	(59, 19)	**Fail**
	(62, 134)	(62, 16)	**Fail**
	(62, 169)	(62, 169)	Pass
	(59, 172)	(59, 172)	Pass
8	(48, 172)	(48, 172)	Pass
	(48, 137)	(48, 19)	**Fail**
	(59, 137)	(59, 19)	**Fail**
	(59, 172)	(59, 172)	Pass
	(48, 172)	(48, 172)	Pass
9	(59, 137)	(59, 19)	**Fail**
	(62, 134)	(62, 16)	**Fail**
	(52, 134)	(52, 16)	**Fail**
	(48, 137)	(48, 19)	**Fail**
	(59, 137)	(59, 19)	**Fail**
10	(69, 172)	(69, 172)	Pass
	(69, 120)	(69, 120)	Pass
	(73, 117)	(73, 117)	Pass
	(73, 169)	(73, 169)	Pass
	(69, 172)	(69, 172)	Pass
11	(59, 172)	(59, 172)	Pass
	(59, 120)	(59, 120)	Pass
	(69, 120)	(69, 120)	Pass
	(69, 172)	(69, 172)	Pass
	(59, 172)	(59, 172)	Pass
12	(69, 120)	(69, 120)	Pass
	(73, 117)	(73, 117)	Pass
	(62, 117)	(62, 117)	Pass
	(59, 120)	(59, 120)	Pass
	(69, 120)	(69, 120)	Pass
13	(80, 172)	(80, 172)	Pass
	(80, 94)	(80, 94)	Pass
	(84, 91)	(84, 91)	Pass
	(84, 169)	(84, 169)	Pass
	(80, 172)	(80, 172)	Pass
14	(69, 172)	(69, 172)	Pass
	(69, 94)	(69, 94)	Pass
	(80, 94)	(80, 94)	Pass
	(80, 172)	(80, 172)	Pass
	(69, 172)	(69, 172)	Pass
15	(80, 94)	(80, 94)	Pass
	(84, 91)	(84, 91)	Pass
	(73, 91)	(73, 91)	Pass
	(69, 94)	(69, 94)	Pass
	(80, 94)	(80, 94)	Pass
16	(107, 172)	(107, 172)	Pass
	(107, 125)	(107, 125)	Pass
	(110, 122)	(110, 122)	Pass
	(110, 169)	(110, 169)	Pass
	(107, 172)	(107, 172)	Pass
17	(96, 172)	(96, 172)	Pass
	(96, 125)	(96, 125)	Pass
	(107, 125)	(107, 125)	Pass
	(107, 172)	(107, 172)	Pass
	(96, 172)	(96, 172)	Pass
18	(107, 125)	(107, 125)	Pass
	(110, 122)	(110, 122)	Pass
	(100, 122)	(100, 122)	Pass
	(96, 125)	(96, 125)	Pass
	(107, 125)	(107, 125)	Pass

Table A.3 Point-based Verification 3

Polygon No.	Original	Destination	Result
19	(117, 172)	(117, 172)	Pass
	(117, 106)	(117, 106)	Pass
	(121, 103)	(121, 103)	Pass
	(121, 169)	(121, 169)	Pass
	(117, 172)	(117, 172)	Pass
20	(107, 172)	(107, 172)	Pass
	(107, 106)	(107, 106)	Pass
	(117, 106)	(117, 106)	Pass
	(117, 172)	(117, 172)	Pass
	(107, 172)	(107, 172)	Pass
21	(117, 106)	(117, 106)	Pass
	(121, 103)	(121, 103)	Pass
	(110, 103)	(110, 103)	Pass
	(107, 106)	(107, 106)	Pass
	(117, 106)	(117, 106)	Pass
22	(128, 172)	(128, 172)	Pass
	(128, 92)	(128, 92)	Pass
	(132, 89)	(132, 89)	Pass
	(132, 169)	(132, 169)	Pass
	(128, 172)	(128, 172)	Pass
23	(117, 172)	(117, 172)	Pass
	(117, 92)	(117, 92)	Pass
	(128, 92)	(128, 92)	Pass
	(128, 172)	(128, 172)	Pass
	(117, 172)	(117, 172)	Pass
24	(128, 92)	(128, 92)	Pass
	(132, 89)	(132, 89)	Pass
	(121, 89)	(121, 89)	Pass
	(117, 92)	(117, 92)	Pass
	(128, 92)	(128, 92)	Pass

Polygon No.	Original	Destination	Result
25	(155, 172)	(155, 172)	Pass
	(155, 19)	(155, 19)	Pass
	(159, 16)	(159, 16)	Pass
	(159, 169)	(159, 169)	Pass
	(155, 172)	(155, 172)	Pass
26	(144, 172)	(144, 172)	Pass
	(144, 19)	(144, 19)	Pass
	(155, 19)	(155, 19)	Pass
	(155, 172)	(155, 172)	Pass
	(144, 172)	(144, 172)	Pass
27	(155, 19)	(155, 19)	Pass
	(159, 16)	(159, 16)	Pass
	(148, 16)	(148, 16)	Pass
	(144, 19)	(144, 19)	Pass
	(155, 19)	(155, 19)	Pass
28	(166, 172)	(166, 172)	Pass
	(166, 113)	(166, 113)	Pass
	(169, 110)	(169, 110)	Pass
	(169, 169)	(169, 169)	Pass
	(166, 172)	(166, 172)	Pass
29	(155, 172)	(155, 172)	Pass
	(155, 113)	(155, 113)	Pass
	(166, 113)	(166, 113)	Pass
	(166, 172)	(166, 172)	Pass
	(155, 172)	(155, 172)	Pass
30	(166, 113)	(166, 113)	Pass
	(169, 110)	(169, 110)	Pass
	(159, 110)	(159, 110)	Pass
	(155, 113)	(155, 113)	Pass
	(166, 113)	(166, 113)	Pass

Table A.4 Point-based Verification 3

Polygon No.	Original	Destination	Result
31	(176, 172)	(176, 172)	Pass
	(176, 95)	(176, 95)	Pass
	(180, 93)	(180, 93)	Pass
	(180, 169)	(180, 169)	Pass
	(176, 172)	(176, 172)	Pass
32	(166, 172)	(166, 172)	Pass
	(166, 95)	(166, 95)	Pass
	(176, 95)	(176, 95)	Pass
	(176, 172)	(176, 172)	Pass
	(166, 172)	(166, 172)	Pass
33	(176, 95)	(176, 95)	Pass
	(180, 93)	(180, 93)	Pass
	(169, 93)	(169, 93)	Pass
	(166, 95)	(166, 95)	Pass
	(176, 95)	(176, 95)	Pass
34	(203, 172)	(203, 172)	Pass
	(203, 137)	(203, 137)	Pass
	(207, 134)	(207, 134)	Pass
	(207, 169)	(207, 169)	Pass
	(203, 172)	(203, 172)	Pass
35	(192, 172)	(192, 172)	Pass
	(192, 137)	(192, 137)	Pass
	(203, 137)	(203, 137)	Pass
	(203, 172)	(203, 172)	Pass
	(192, 172)	(192, 172)	Pass
36	(203, 137)	(203, 137)	Pass
	(207, 134)	(207, 134)	Pass
	(196, 134)	(196, 134)	Pass
	(192, 137)	(192, 137)	Pass
	(203, 137)	(203, 137)	Pass

Polygon No.	Original	Destination	Result
37	(214, 172)	(214, 172)	Pass
	(214, 118)	(214, 118)	Pass
	(217, 115)	(217, 115)	Pass
	(217, 169)	(217, 169)	Pass
	(214, 172)	(214, 172)	Pass
38	(203, 172)	(203, 172)	Pass
	(203, 118)	(203, 118)	Pass
	(214, 118)	(214, 118)	Pass
	(214, 172)	(214, 172)	Pass
	(203, 172)	(203, 172)	Pass
39	(214, 118)	(214, 118)	Pass
	(217, 115)	(217, 115)	Pass
	(207, 115)	(207, 115)	Pass
	(203, 118)	(203, 118)	Pass
	(214, 118)	(214, 118)	Pass
40	(224, 172)	(224, 172)	Pass
	(224, 97)	(224, 97)	Pass
	(228, 94)	(228, 94)	Pass
	(228, 169)	(228, 169)	Pass
	(224, 172)	(224, 172)	Pass
41	(214, 172)	(214, 172)	Pass
	(214, 97)	(214, 97)	Pass
	(224, 97)	(224, 97)	Pass
	(224, 172)	(224, 172)	Pass
	(214, 172)	(214, 172)	Pass
42	(224, 97)	(224, 97)	Pass
	(228, 94)	(228, 94)	Pass
	(217, 94)	(217, 94)	Pass
	(214, 97)	(214, 97)	Pass
	(224, 97)	(224, 97)	Pass

APPENDIX B

In this appneix, test results for vector-based verification, which is discussed in chapter 5, is showed.

Table B.1 Vector-Based Verification 1

Polygon No.	Angle		Length		Result
	Original	Destination	Original	Destination	
1	-0.785	-0.785	4.243	4.243	Pass
	0.000	0.000	193.000	193.000	Pass
	0.785	0.785	4.243	4.243	Pass
	0.000	0.000	193.000	193.000	Pass
2	-1.571	-1.571	153.000	153.000	Pass
	-0.588	-0.588	3.606	3.606	Pass
	1.571	1.571	152.000	152.000	Pass
	0.785	0.785	4.243	4.243	Pass
3	-1.571	-1.571	152.000	152.000	Pass
	0.000	0.000	193.000	193.000	Pass
	1.571	1.571	152.000	152.000	Pass
	0.000	0.000	193.000	193.000	Pass
4	0.785	0.785	4.243	4.243	Pass
	0.000	0.000	193.000	193.000	Pass
	-0.785	-0.785	4.243	4.243	Pass
	0.000	0.000	193.000	193.000	Pass
5	-1.571	-1.571	153.000	153.000	Pass
	-0.588	-0.588	3.606	3.606	Pass
	1.571	1.571	152.000	152.000	Pass
	0.785	0.785	4.243	4.243	Pass
6	-1.571	-1.571	152.000	152.000	Pass
	0.000	0.000	193.000	193.000	Pass
	1.571	1.571	152.000	152.000	Pass
	0.000	0.000	193.000	193.000	Pass
7	-1.571	-1.571	35.000	153.000	**Fail**
	-0.785	-0.785	4.243	4.243	Pass
	1.571	1.571	35.000	153.000	**Fail**
	0.785	0.785	4.243	4.243	Pass

Table B.2 Vector-Based Verification 2

Polygon No.	Angle		Length		Result
	Original	Destination	Original	Destination	
8	-1.571	-1.571	**35.000**	**153.000**	**Fail**
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	**35.000**	**153.000**	**Fail**
	0.000	0.000	11.000	11.000	Pass
9	-0.785	-0.785	4.243	4.243	Pass
	0.000	0.000	10.000	10.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
10	-1.571	-1.571	52.000	52.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	52.000	52.000	Pass
	0.644	0.644	5.000	5.000	Pass
11	-1.571	-1.571	52.000	52.000	Pass
	0.000	0.000	10.000	10.000	Pass
	1.571	1.571	52.000	52.000	Pass
	0.000	0.000	10.000	10.000	Pass
12	-0.644	-0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
	0.785	0.785	4.243	4.243	Pass
	0.000	0.000	10.000	10.000	Pass
13	-1.571	-1.571	78.000	78.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	78.000	78.000	Pass
	0.644	0.644	5.000	5.000	Pass
14	-1.571	-1.571	78.000	78.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	78.000	78.000	Pass
	0.000	0.000	11.000	11.000	Pass
15	-0.644	-0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass

Table B.3 Vector-Based Verification 3

Polygon No.	Angle		Length		Result
	Original	Destination	Original	Destination	
16	-1.571	-1.571	47.000	47.000	Pass
	-0.785	-0.785	4.243	4.243	Pass
	1.571	1.571	47.000	47.000	Pass
	0.785	0.785	4.243	4.243	Pass
17	-1.571	-1.571	47.000	47.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	47.000	47.000	Pass
	0.000	0.000	11.000	11.000	Pass
18	-0.785	-0.785	4.243	4.243	Pass
	0.000	0.000	10.000	10.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
19	-1.571	-1.571	66.000	66.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	66.000	66.000	Pass
	0.644	0.644	5.000	5.000	Pass
20	-1.571	-1.571	66.000	66.000	Pass
	0.000	0.000	10.000	10.000	Pass
	1.571	1.571	66.000	66.000	Pass
	0.000	0.000	10.000	10.000	Pass
21	-0.644	-0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
	0.785	0.785	4.243	4.243	Pass
	0.000	0.000	10.000	10.000	Pass
22	-1.571	-1.571	80.000	80.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	80.000	80.000	Pass
	0.644	0.644	5.000	5.000	Pass
23	-1.571	-1.571	80.000	80.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	80.000	80.000	Pass
	0.000	0.000	11.000	11.000	Pass

Table B.4 Vector-Based Verification 4

Polygon No.	Angle		Length		Result
	Original	Destination	Original	Destination	
24	-0.644	-0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
25	-1.571	-1.571	153.000	153.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	153.000	153.000	Pass
	0.644	0.644	5.000	5.000	Pass
26	-1.571	-1.571	153.000	153.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	153.000	153.000	Pass
	0.000	0.000	11.000	11.000	Pass
27	-0.644	-0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
28	-1.571	-1.571	59.000	59.000	Pass
	-0.785	-0.785	4.243	4.243	Pass
	1.571	1.571	59.000	59.000	Pass
	0.785	0.785	4.243	4.243	Pass
29	-1.571	-1.571	59.000	59.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	59.000	59.000	Pass
	0.000	0.000	11.000	11.000	Pass
30	-0.785	-0.785	4.243	4.243	Pass
	0.000	0.000	10.000	10.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
31	-1.571	-1.571	77.000	77.000	Pass
	-0.464	-0.464	4.472	4.472	Pass
	1.571	1.571	76.000	76.000	Pass
	0.644	0.644	5.000	5.000	Pass

Table B.5 Vector-Based Verification 5

Polygon No.	Angle		Length		Result
	Original	Destination	Original	Destination	
32	-1.571	-1.571	77.000	77.000	Pass
	0.000	0.000	10.000	10.000	Pass
	1.571	1.571	77.000	77.000	Pass
	0.000	0.000	10.000	10.000	Pass
33	-0.464	-0.464	4.472	4.472	Pass
	0.000	0.000	11.000	11.000	Pass
	0.588	0.588	3.606	3.606	Pass
	0.000	0.000	10.000	10.000	Pass
34	-1.571	-1.571	35.000	35.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	35.000	35.000	Pass
	0.644	0.644	5.000	5.000	Pass
35	-1.571	-1.571	35.000	35.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	35.000	35.000	Pass
	0.000	0.000	11.000	11.000	Pass
36	-0.644	-0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass
37	-1.571	-1.571	54.000	54.000	Pass
	-0.785	-0.785	4.243	4.243	Pass
	1.571	1.571	54.000	54.000	Pass
	0.785	0.785	4.243	4.243	Pass
38	-1.571	-1.571	54.000	54.000	Pass
	0.000	0.000	11.000	11.000	Pass
	1.571	1.571	54.000	54.000	Pass
	0.000	0.000	11.000	11.000	Pass
39	-0.785	-0.785	4.243	4.243	Pass
	0.000	0.000	10.000	10.000	Pass
	0.644	0.644	5.000	5.000	Pass
	0.000	0.000	11.000	11.000	Pass

Table B.6 Vector-Based Verification 6

Polygon No.	Angle		Length		Result
	Original	Destination	Original	Destination	
40	-1.571	-1.571	75.000	75.000	Pass
	-0.644	-0.644	5.000	5.000	Pass
	1.571	1.571	75.000	75.000	Pass
	0.644	0.644	5.000	5.000	Pass
41	-1.5708	-1.5708	75.000	75.000	Pass
	0	0	10.000	10.000	Pass
	1.570796	1.570796	75.000	75.000	Pass
	0	0	10.000	10.000	Pass
42	-0.6435	-0.6435	5.000	5.000	Pass
	0	0	11.000	11.000	Pass
	0.785398	0.785398	4.243	4.243	Pass
	0	0	10.000	10.000	Pass

APPENDIX C

In this section, developed tool operation is succinctly explained.

Requirement to run the tool:

Operating System: Microsoft Windows 2000 (Build 2195)

Invoking:

Invoking a tool, Figure C.1's screen shows.

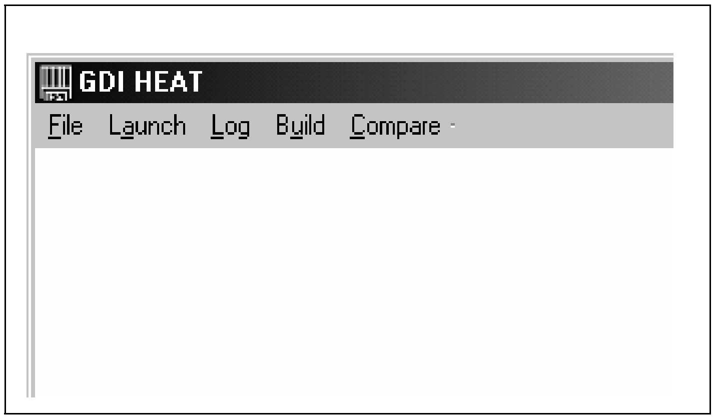


Figure C.1: Main Screen

File Saving

- **File/Save as Original:** Save a file as original (Figure C.2)
- **File/Save as Destination**: Save a file as destination

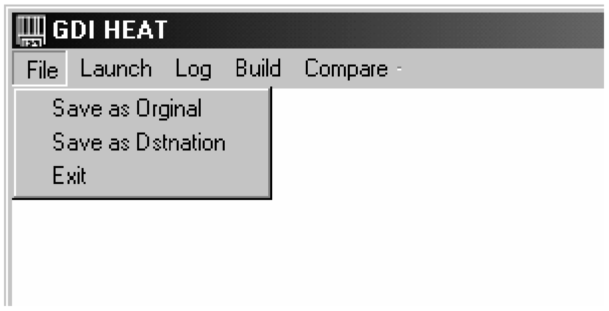


Figure C.2: File Saving

Logging

- **Log/Execute:** Forcibly redraw target drawing object and log GDI information
- **Log/Show Result:** Show the logging result
- **Log/Delete Log:** Delete the log file.

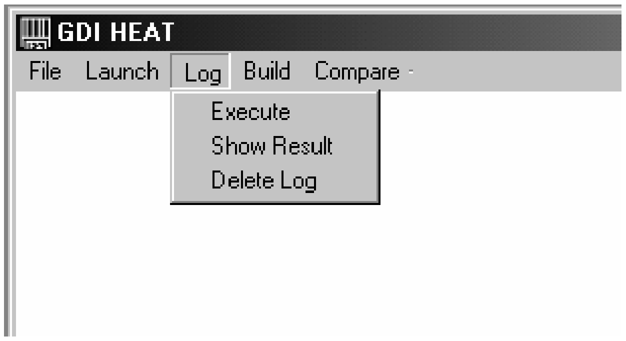


Figure C.3: Logging

Building

- **Build/Bind Files:** Bind the saved file and prepared C file.
- **Build/Show C Source File:** Show the bound C file, which can be compiled
- **Build/Build All:** Build bound C file
- **Build/Recover the Picture:** Regenerate and show saved drawing object.

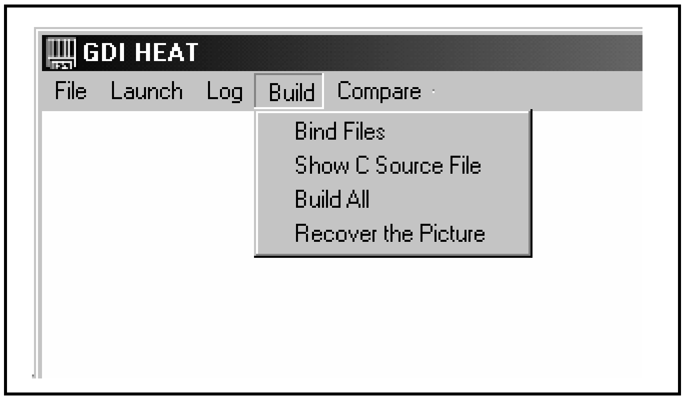


***Figure C.4:* Build a File**

Comparing

- **Compare/Bind Org and Dst**: Bind the original and destination file.
- **Compare/Build**: Build the bound file
- **Compare/Compare and Show Result**: Compare original and destination object and show the result.
- **Compare/Error Rate (Optional)**:

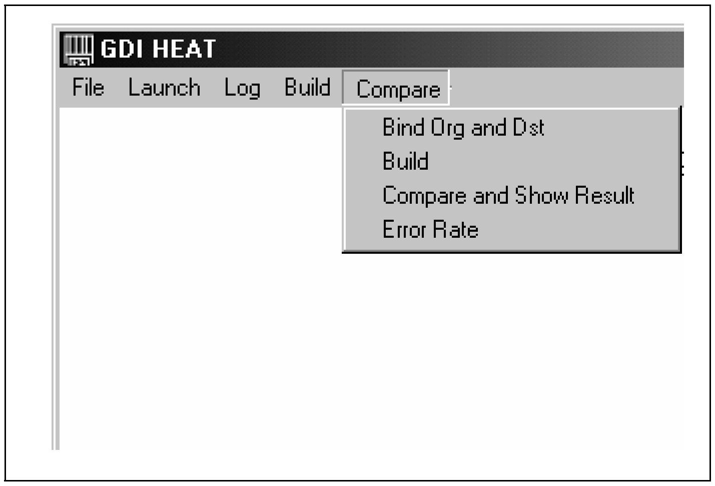


Figure C.5: Compare Objects

Thesis

Version 1.0

Printed Date: 5/28/2003

Word Count: 10509