UNIVERSITY OF CALGARY

# Tool Support for Automatic Performance Testing of Java3D Graphics Applications

by

Xueling Shu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2007

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Tool Support for Automatic Performance Testing of Java3D Graphics Applications" submitted by Xueling Shu in partial fulfilment of the requirements of the degree of Master of Science.

*Supervisor,* Dr. Frank Oliver Maurer, Department of Computer Science

*Co-Supervisor,* Dr. Christoph Wilhelm Sensen, Department of Biochemistry and Molecular Biology

Dr. Victoria Mitchell, Haskayne School of Business

Dr. Jonathan Sillito, Department of Computer Science

*Date*

## Abstract

Java3D$^{TM}$ is used on 3D software such as computer games, medical imaging and information visualization. One common concern regarding these applications is performance which is a key attribute for the quality of all 3D programs and is represented as image rendering speed. With automated and effective testing, one is able to regularly evaluate the performance of Java3D applications Thus performance issues can be exposed in a timely manner. However, existing tools and strategies offer limited support for such an evaluation, resulting in a call for tool support to conduct effective Java3D performance testing. To fulfill this request, a new testing framework, J3DPerfUnit, was developed by the researcher. The tool allows Java3D developers to write and automatically run Java3D performance test cases. Developers can check test results on the command line and in test report files. A preliminary evaluation confirmed the tool concept, and uncovered shortcomings in the tool's current version.

## Publications

Some content, ideas and figures from this thesis have appeared in the following peer-reviewed publication:

Xueling Shu, Frank Maurer; A Tool for Automated Performance Testing of Java3D Applications in Agile Environments; Proc. of the Second International Conference on Software Engineering Advances (ICSEA 2007), IEEE 2007 (6 pages).

## Acknowledgements

Thank you, my dear father and mother. It was you two that brought me up and taught me how to be independent. With your support, I learned how to fight with the most unacceptable and cruel fact that life gave me one year ago. Love you forever!

Thanks, Lawrence! You were always with me whatever happened for which I felt so lucky.

Thank you, Frank! With your guidance, I learned a lot during this two-year master study.

Thanks for your support, too, Christoph!

Hi, ladies and gentlemen in EBE group! I very much enjoyed studying with you guys. Good luck!

## Dedication

To my family and friends who cheered me up all the way along.

## Table of Contents

## List of Tables

## List of Figures and Illustrations

## List of Symbols, Abbreviations and Nomenclature

| Symbol | Definition |
|---|---|
| 3D | Three-Dimensional |

# Chapter One: Introduction

This chapter is divided into seven parts. At the beginning, I briefly discuss the significance of performance testing for 3D applications. Then, I explain the research motivation which arose from my experience with a Java3D application: the difficulties in effectively evaluating the application performance due to the restrictions in current testing practices and tools. Next, I confine the research scope to the area of automated performance testing for Java3D applications with tool support. Then, I point out the goals set for my work and the hypothesis stated for verification. Finally, an overview of the key terminology used in this work is provided and the thesis structure is presented.

## 1.1 Research Background

3D implementations are increasingly adopted in a vast variety of realms. For example, in the area of bioinformatics and computational biology, researchers use Java3D$^{TM}$, a 3D graphics technique, to visualize protein structure, availing areas like drug design and protein modelling [Can et al.03, Meloan01].

The quality of 3D applications is dependent on image display performance. Display performance is defined as a reasonable time for an object to display on a screen, or a smooth rendering during mouse/keyboard interactions. Without acceptable performance, 3D systems are inclined to be useless, just like many other software projects have become [Sommerville04]. For instance, it would be onerous to study a protein structure with a 3D application which cannot render the structure in a reasonable time or cannot produce a quick response during mouse/keyboard interactions. To achieve high 3D performance, repeated and continuous testing needs to be conducted. In this thesis, **repeated and continuous performance testing** means: whenever source code is changed (a new feature is developed, a bug is fixed, or code refactoring is done), developers should either write new test cases or execute old test cases to see if the change introduces any performance problems. This regular performance inspection allows for a timely exposure of problems and the developers' quick responses. Thus, performance issues will not accumulate to an extent where they are difficult to resolve [Chung et al.00, Clements96, Smith et al.02].

## 1.2 Research Motivation

Notwithstanding the importance of frequent performance testing for 3D applications, current tool support and strategies provide very limited assistance. This was particularly highlighted and then became my research motivation when I worked as a tester in a bioinformatics project, called "WEPA" [WEPA07], from September 2005 to September 2006,

WEPA is running at the Sun Center of Excellence for Visual Genomics at the University of Calgary, and is funded by the Governments of Canada and Alberta. The team is developing a new tool that allows medical researchers to view a complete image of disease mechanisms in a 3D environment. System implementation is based on both Java2D [Knudsen99] and Java3D [Chen et al.06]. One can use 2D menus to trigger events such as loading/unloading organs into/from a human body. Using mouse and keyboard, a user can interact (rotate, translate and zoom) with 3D scenes and observe organs from various perspectives. Good system performance is required in order to make sure the software responds well to users. As part of my testing role, I carried out frequent performance examinations that, in the end, reflected a need for tool support that could not be fulfilled at that point.

The project development followed Scrum [Schwaber et al.01], one of the incremental and evolutionary development processes. Such processes promote that software should be iteratively developed and continuously tested until a full system is implemented.

To test WEPA, I used two test automation tools, JUnit [JUnit07] and Jemmy [Jemmy07]. With the tools' help, the user stories coded in plain Java and Java2D were effectively tested. However, on the Java3D level, with no appropriate testing tools available, we were forced to follow a tedious and error-prone manual regression testing process. This laborious process negatively affected the testing efficiency, especially when we needed to continuously tune and test for emerging performance problems, such as an unreasonably long execution time to load an organ with large dataset, and a frozen 3D scene during mouse/keyboard interactions. After having seen the benefits that automated Java2D testing brought, the team looked forward to automating Java3D performance tests. Therefore, I was motivated to develop J3DPerfUnit, the tool that is able to automatically run performance tests for Java3D applications.

## 1.3 Scope

My research focuses on the area of Java3D performance test automation with tool support. This focus is shown as the bold italic part in **Figure 1-1**. Starting from the outermost box in **Figure 1-1**, *software testing* intends to examine the status of a software system, uncover defects, ensure that the software does what it is supposed to do and guarantee that the user expectation is satisfied [Watkins01]. *Performance testing* is one type of software testing that validates system performance against requirements, measures system capacity, and discloses performance issues and bottlenecks [Gao03]. *3D* *application performance testing* exclusively emphasizes evaluating 3D systems' rendering efficiency. In particular, my work addresses the research problem of automated performance testing tool support for applications programmed in Java3D.

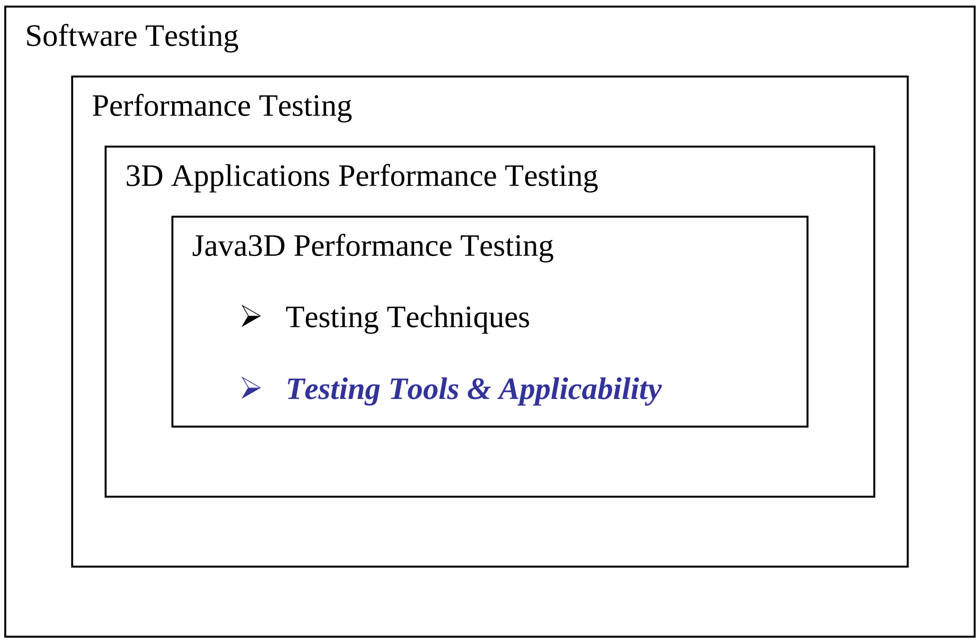

Software Testing
Performance Testing
3D Applications Performance Testing
Java3D Performance Testing
Testing Techniques
Testing Tools & Applicability

**Figure 1-1: Scope of research – the center part**

## 1.4 Goals

In order to efficiently and effectively test the performance of 3D applications during incremental development, I defined the following research goals:

- To thoroughly explore Java3D performance issues;
- To investigate current tool support and approaches for Java3D performance testing in detail, demonstrating their restrictions in effectively testing Java3D performance regularly and continuously;
- To present requirements for the new tool development;

- To present a proof of concept of tool support for automated Java3D performance testing;
- To evaluate the tool's impact on the effectiveness and efficiency of Java3D performance testing.

## 1.5 Hypothesis

The hypothesis of my work is: compared to current testing strategies and tools, it is effective and more efficient to use our new tool support called J3DPerfUnit to conduct Java3D performance testing. To be effective means: (1) performance test cases can be properly specified, and (2) developers can get accurate and helpful test results. To be efficient means the tool can reduce the effort when performance tests need to be continuously run. The hypothesis will be verified in Chapter Six.

## 1.6 Key Terminology

Software development involves several indispensable practices. One of them is software testing. The goal of **testing** is to find errors that could occur during application runtime [Myers et al.04]. [Burnstein03] coins this term as a process that has two purposes:

- Uncover defects. Defects are the result of an error in software that produces incorrect behaviour.
- Examine the reliability, security, usability and correctness in software. Reliability suggests how well the required functions are performed under the affirmed conditions for a period of time. Usability indicates how much effort, such as time, a user needs to put in order to manage software smoothly. Correctness highlights if the software does as it is supposed to do.

It is recommended that testing be conducted on different levels and in an incremental way [Myers et al.04]. Those levels are:

- **Unit testing:** Examinations of "individual hardware or software units or groups of related units" [IEEE02, 79] are carried out on this testing level. A unit is the smallest element that can be tested [Tamres02], for example, one single method in a Java class. The objective of unit testing is to pinpoint logical and implementation errors in each unit [Binder00].
- **Integration testing:** This testing assembles individual software units, hardware units, or both and evaluates "the interaction between them" [IEEE02, 41]. Irregularities among units' coordination are expected to be discovered in this testing phase.
- **System testing:** When a complete and integrated system is ready, system testing assesses "the system's compliance with its specified requirements" [IEEE02, 74]. This testing guarantees that the system functions adequately against the affirmed requirement.

Both functional and non-functional requirements are examined in each of the above testing levels. One kind of examination on non-functional requirements is called **performance testing.** This testing checks if users' expectations on "memory use, response time, throughput and delays" [Burnstein03] can be met or not. Conducted after performance testing, **performance tuning** seeks solutions for the discovered performance issues [Buzen76].

**Test metrics** are used to define input and output test data. One can interpret the output represented by a numeric value as the degree to which a system quality is affected by the input, the given software attribute [Kaner et al.04]. For instance, in 3D performance testing, execution time is a commonly used metric that indicates how fast a system runs.

To reduce the time and human resources required by the traditional manual testing, **test automation** allows developers or users to set up tests with a tool. The tool then executes the tests, compares the actual output to the asserted one, and reports test results [TestAutomation07]. This activity improves the overall testing efficiency by evaluating software quality automatically and repeatedly. **Testing efficiency** relates to the use of resources that produce effective testing results. The fewer resources required, the more efficient the testing is. In a testing process, resources can refer to people, equipment and time invested.

## 1.7 Thesis structure

This work is organized into eight chapters:

Chapter 2 presents the literature overview. First, some basics of 3D applications are introduced. This includes a brief description of how Java3D applications are executed and an introduction to the graphics libraries Java3D relies on. Next, major causes of 3D performance problems are discussed. Then the limitations in current strategies and tools for Java3D performance testing are detailed. Finally, the benefits and risks involved in test automation are presented.

Chapter 3 presents and analyzes a survey that looks into the state of art of the current tool support for Java3D performance testing. It also describes the most important 3D performance metrics a testing tool should measure from the perspective of 3D developers. At the end of this chapter, I discuss the survey limitations.

Chapter 4 analyzes the tool requirements and explains the tool design.

Chapter 5 shows the usage of our new testing tool, J3DPerfUnit. I will use WEPA as an example to explain the test case design and test report details, and illustrate concrete test code written for different user scenarios.

Chapter 6 discusses the impact of J3DPerfUnit on the effectiveness and efficiency of Java3D performance testing by presenting and analyzing a preliminary tool evaluation.

Chapter 7 outlines suggestions for future work.

Chapter 8 concludes my work by summarizing the research achievements.

# Chapter Two: Literature Overview

This chapter[1] gives background information in four areas:

- Java3D development: I briefly discuss Java3D applications' working mechanism, explain 3D rendering with this Java technology, and introduce two low-level graphics libraries this technology depends on;
- Java3D performance issues and causes;
- Java3D performance testing: Metrics commonly used, and existing tools and strategies are presented;
- Advantages and concerns surrounding test automation.

## 2.1 Java3D Development

Hiding the complexity of low-level rendering details, Java3D provides a high-level and object-oriented view of 3D graphics. Hence, Java developers without much low-level graphics programming experience are able to build 3D applications using the language and design pattern they are familiar with [Day07]. Deriving benefits from this, Java$^{TM}$ has been increasingly adopted in diverse 3D implementations "including mechanical CAD, molecular visualization, scientific visualization, animation previews, geographic information systems, business graphics, 3D logos, and educational offerings" [Sowizral et al.99]. This section presents a short overview of this technology.

[1] Section 2.1 & 2.3 reference the following publication:
Xueling Shu, Frank Maurer; A Tool for Automated Performance Testing of Java3D Applications in Agile Environments; Proc. of the Second International Conference on Software Engineering Advances (ICSEA 2007), IEEE 2007 (6 pages).

### *2.1.1 Background*

Early versions of Java missed important pieces supporting graphics implementation. To extend Java usage, Sun and its partners launched projects developing 2D and 3D APIs. From 1996 to 2004, the 3D part gradually evolved into an open source project called Java3D [Java3D07]. Until now, this technology has been applied in a variety of domains, such as molecular visualization, business graphics, 3D logos, educational offerings and geographic information systems [Sowizral et al.99].

Basically, two critical factors contribute to the growth of Java3D applications. First, it is the platform-independent feature that attracts researchers and business to employ Java based technology. Second, in comparison with traditional 3D APIs, such as OpenGL [OpenglArchitectureReviewBoard04] and DirectX [Bargen98], Java3D requires a much lower programming effort by offering a high-level and object-oriented view of 3D graphics, and by hiding low level hardware rendering details. The high-level and object-oriented view of 3D graphics is represented by a tree structure called scene graph. The scene graph specifies information about 3D objects, including shapes, locations, orientations, and hierarchical relationship [Java3DAPITutorial07]. Through manipulating nodes on this tree, one can implement visualization, animation and interaction of 3D objects.

### *2.1.2 Rendering 3D objects with scene graph*

**Figure 2-1** illustrates loading a 3D human body onto a black background using the WEPA application. Figure 2-2 shows how the corresponding scene graph changes before (left side) and after (right side) the body is loaded.

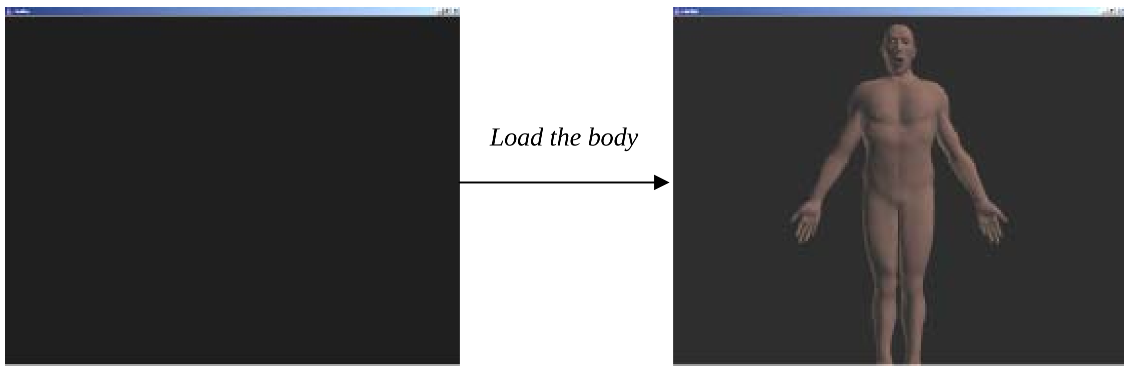


**Figure 2-1: Load a human body onto the background**

**Scene graph: before the body is loaded** **Scene graph: after the body is loaded**

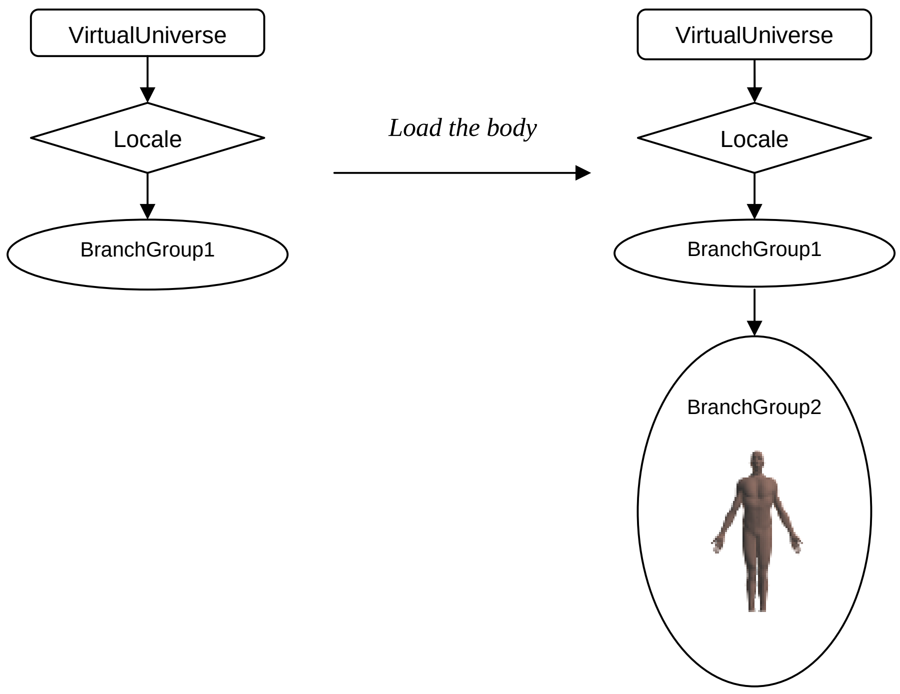


**Figure 2-2: the changing scene graph**

In **Figure 2-2**, one scene graph object needs an explanation: A BranchGroup object "serves as a pointer to the root of a scene graph branch" [Java3DAPISpecification07] and maintains the information of one or more 3D objects. To describe a complete scene, one BranchGroup object can be attached by other BranchGroup objects and forms a hierarchy. A 3D object becomes visible or invisible after its equivalent BranchGroup object is added or removed from the scene graph.

In our case BranchGroup1 stores the common information that all organs have (such as the capability to change the organ data in the program), while BranchGroup2 specifically depicts the data for the human body. BranchGroup2 is attached to BranchGroup1, which forms a hierarchy. This hierarchy describes the scene which is on the right side of **Figure 2-1** After BranchGroup2 is added to the scene graph, the body appears.

A typical object loading pipeline in Java3D is illustrated in **Figure 2-3**.

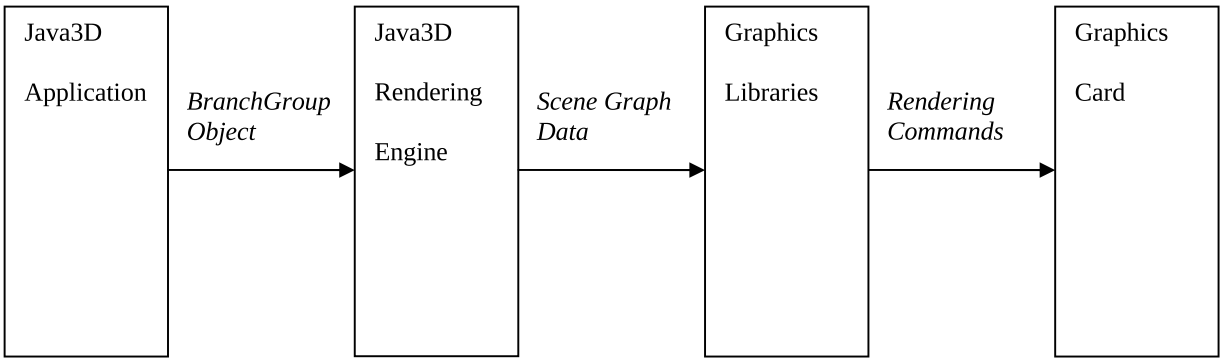


**Figure 2-3: A typical object loading pipeline in Java3D**

The loading pipeline involves four stages:

- First stage: a Java3D application analyzes the object's geometry data stored in files or memory, and generates a corresponding BranchGroup object that handles the information describing what the object should look like. Then, the BranchGroup object is forwarded to the Java3D rendering engine through Java3D API calls.
- Second stage: the Java3D rendering engine processes and optimizes the data, and places the BranchGroup onto the scene graph. Then the engine sends the scene graph data to the underlying graphics libraries.
- Third stage: The graphics libraries receive the scene graph data. Then they produce and dispatch rendering commands to the graphics card.
- Forth stage: The graphics card receives the rendering commands, and finally renders the scene graph.

Knowledge of time spent on each stage gives developers a better idea of where performance problems stem from.

A typical object unloading pipeline in Java3D is illustrated in **Figure 2-4**.

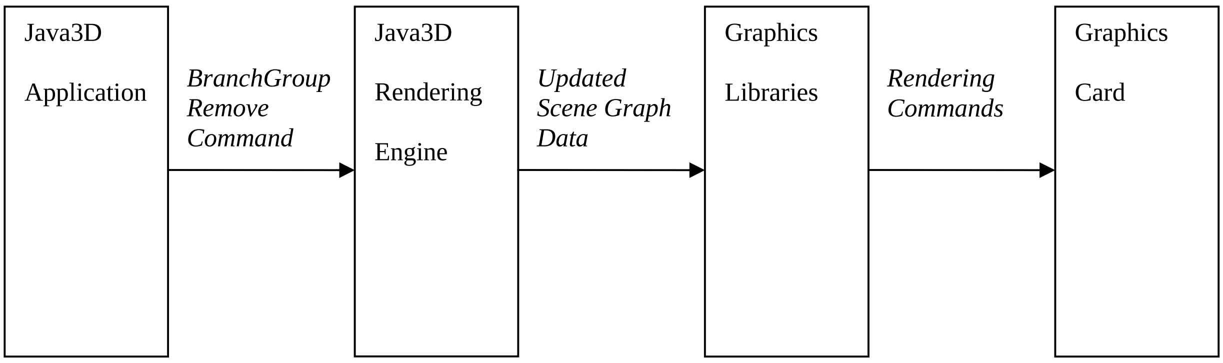


**Figure 2-4: A typical object unloading pipeline in Java3D**

The unloading pipeline also consists of four phases:

- First phase: The Java3D application calls the API which requests the Java3D engine to remove the object's BranchGroup.
- Second phase: The rendering engine unloads the BranchGroup from the scene graph, and sends the updated scene graph data to graphics libraries

What happens in the third and forth phases are the same as that in the third and forth stages of object loading pipeline.

### *2.1.3 Low-Level Graphics Libraries*

Graphics libraries contain graphics commands or functions. These commands are low level implementations because they only implement simple shapes like points, lines, and polygons. To create a complex drawing, a higher level implementation based on low-level graphics libraries is needed [Clingman et al.04]. Java3D is such a high-level implementation. It binds two graphics libraries: OpenGL and DirectX.

As the most widely used graphics library, OpenGL is device-independent and gives developers a good control over the entire 3D implementation. Intended for real-time, high-performance applications, OpenGL grants the power of hardware acceleration [Clingman et al.04]. "In computing, *hardware acceleration* is the use of hardware to perform some function faster than is possible in software running on the normal (general purpose) CPU" [HardwareAcceleration07]. In particular, hardware acceleration utilizes the functionalities already implemented by graphics cards to improve applications' performance.

DirectX is another graphics library specifically developed for the Windows platform. It provides similar graphics commands to those in OpenGL. DirectX is also supported by PC graphics card vendors.

## 2.2 Java3D Performance Issues and Causes

Through building a communication layer in its framework, Java3D forwards all the low level rendering details to the graphics libraries it relies on. Because these libraries empower hardware acceleration, a high performance is therefore expected in Java3D applications. However, a common concern with these applications is a slow rendering speed even with good graphics devices installed on a machine [Can et al.03].

Usually, a performance drop is manifested as an unreasonably long execution time for user interactions. For example, in WEPA, loading an arm into the human body cost more than 20 seconds, and using mouse and/or keyboard to move the human body carrying large organs ended up with a frozen application. Consequently, the whole system became unresponsive, causing difficulty in doing efficient medical research.

Major factors contributing to a performance drop can be categorized into three aspects: a heavyweight scene graph, inefficient 3D file loaders, and limitations in Java3D itself. In the following discussion, each aspect will be explained.

### *2.2.1 A heavyweight scene graph*

This type of scene graph is the most common cause of low performance. It either represents an inappropriately built tree structure or holds complicated 3D objects with large datasets:

- **Unsuitable scene graph structure:** One 3D scene can be represented by several differently constructed scene graphs. When determining which structure should be adopted, developers must be very cautious. A wrong decision may lead to a complex scene graph which takes considerable time to be processed and optimized by the Java3D engine. To develop a proper structure, one needs to have a deep understanding of various Java3D objects on the scene graph, called nodes. Types and organizations of these nodes largely affect scene graph efficiency. This is clearly reflected by [Can et al.03]. To speed up real time interaction and rendering of large molecules, authors of [Can et al.03] modified their scene graph structure three times through changing types and organization of nodes. Such a modification impressively brought "up to eight times improvement in rendering speed and could load molecules three times as large as the previous systems could" [Can et al.03].

- **Complicated 3D objects:** In Java3D, geometric datasets, such as polygons and pixels, are examined and optimized by the APIs and 3D file loaders which converts the datasets stored in files into a Java3D object on the scene graph. Carrying thousands or even millions of polygons, intricate 3D objects add a heavy burden onto the file loaders and APIs. A huge computing effort is required for the analysis and optimization of large datasets. Such an effort could induce an extended application response time, or even a memory leak causing the application to crash. However, to generate images that can precisely depict some 3D objects, dealing with large datasets is not rare. **Figure 2-5** gives an example from the WEPA application.

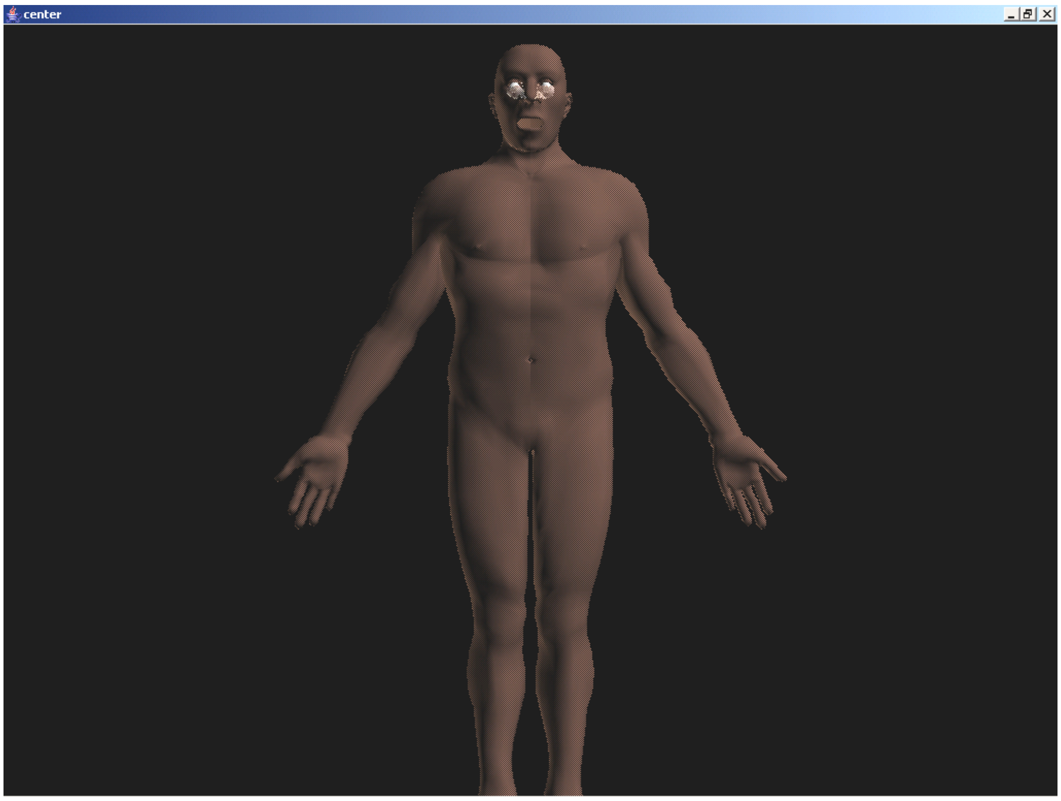


**Figure 2-5: A human body with eyes**

Despite the simple appearance, the body holds thousands of polygons; while the eyes are described by millions of polygons although they are much smaller than the body. A large amount of polygons is used to enhance real-life resemblance, so that medical researchers could effectively run their studies on simulated organs as they are working on real ones. Unfortunately, although the visual precision was enhanced, large datasets resulted in the deprivation of a quick system response during user interactions. For instance, displaying **Figure 2-5** needs 7 seconds. Discovering how to refine the number and distribution of polygons in large 3D objects demands a profound knowledge in mathematics.

### *2.2.2 Inefficient 3D File Loaders*

Besides describing geometry data of a 3D object in source code, developers can use scene files [Java3DAPITutorial07] to store the data. Since there are a variety of scene file formats, different file loaders came into existence. Many times, the selection of loaders has a critical impact on application performance. An inefficient loader lengthens the data analysis process and wastes a considerable amount of system time.

As one solution, developers sometimes create their own loaders tailored to their system's specific needs.

Another solution is to change the scene file format. For example, after doing a comparison of two file loaders, WEPA developers planned to substitute the old "j3f" file to "xml" as they discovered that existing loaders for the first one were not efficient enough while loaders for the second one seemed to be faster.

### *2.2.3 Java3D limitations*

Although the high level design in Java3D largely reduces the programming effort, it also has downsides which confine the room for performance improvement. The three leading constraints which may trigger performance drops are [Day07]:

- **Rendering-pipeline details are unavailable for developers:** In order to flatten the learning curve for Java developers who have no extensive low-level 3D programming experience, Java3D provides high-level APIs, hides low-level rendering details, and does the rendering optimization by itself. The drawback is the developers' incapability to touch low-level details when they are important for investigating causes of performance problems.

- **Limitations in the Java3D implementation:** Certain limitations in the Java3D implementation restrict performance optimization. [Can et al.03] gives an example. As a scene graph would become heavyweight with a large amount of nodes on it, the authors wanted to reduce the amount. He put those atoms which have the same appearance, but different coordinates under a single Sphere [Java3DAPISpecification07] object (***A sphere object is one of the scene graph nodes describing a 3D object whose shape is a sphere, for example, the shape of an atom***). Hence, when the atoms' number boosts, the number of Sphere nodes will not increase quickly. However, it turned out that one Sphere node can only describe one single atom. Consequently, when the number of atoms increases dramatically, many Sphere nodes are required, and a big performance problem may occur.
- **Achievement of hardware acceleration is challenging:** Usually, graphics card vendors optimize drivers like OpenGL and DirectX on their own. **"**It is in each vendor's best interest to build proprietary extensions and make proprietary optimizations to sell more of its own hardware. As with all hardware optimizations, you must use accelerator-specific OpenGL optimizations with the understanding that each optimization for one platform diminishes portability and performance for several others" [Day07]. Therefore, Java3D's more general-purpose optimizations make it hard for applications to reach uniform performance on dissimilar graphics cards. As a result, the same Java3D application may run at a totally different speed under a changed hardware setting.

While I focused on three major kinds of factors that may affect Java3D performance, there are definitely other causes influencing an application's execution speed, such as hardware selection.

## 2.3 Performance Testing for Java3D Applications

Throughout the development of 3D applications, performance needs to be tested and tuned regularly. This section reviews two metrics most widely used for 3D performance testing, and discusses the existing tool support and strategies.

### *2.3.1 Testing Metrics*

There are two popular metrics adopted in performance testing for 3D applications. One is execution time, the other frame rate [FrameRate07].

As the clearest sign of how fast a system runs, execution time is an ideal standard for 3D performance assessment. Through measuring the execution time, one can easily tell if a user interaction, such as loading a 3D object, runs longer than expected. Nevertheless, due to a lack of appropriate tools, execution time is usually calculated manually, which is not only inefficient, but also inaccurate.

Frame rate, or FPS (frames per second), or frame frequency, represents the number of images, called frames, produced by a graphics device every second. A higher frame rate indicates a smoother rendering. In general, 30fps is equivalent to movie performance. Frame rate can be measured with existing tool support. Nonetheless, such support does not fully automate the measurement process. Also the support has a few restrictions.

In the next part, I will discuss current tools and methods utilized in today's 3D performance testing, and their limitations which motivated my research.

### *2.3.2 Existing Tools and Strategies*

To carry out effective and efficient performance testing, it is essential to have tools that not only determine meaningful metrics which fit various system needs, but also can automatically execute test cases. However, at present, no such tool support exists in the area of Java3D performance testing. This section takes a look at this situation through an in-depth introduction of today's popular testing frameworks and tools.

#### 2.3.2.1 JUnit and JUnitPerf

JUnit is a regression testing framework for Java developers to write unit tests. In this framework, a Java class defines a test case, and in each test case a method defines a test scenario. With the help of predefined JUnit assertions, each test case assesses if some features behave as expected. All test cases can run together through being organized into a test suite. Test results are reported right after one test case or test suite is executed, clearly indicating whether tests pass or fail. The tool shows the test result either with text descriptions in command line, or with green (a test passes) or red (a test fails) bars shown in an IDE window. The automation and evident test results make JUnit the standard regression testing framework for Java programs.

Extending its previous focus on only evaluating an application's functional requirements, JUnit4 [JUnit4In10Minutes07] gives a new parameter called "timeout", with which one can write a test case not only assessing the correctness of application features, but also investigating if the test case can pass in a certain amount of time. This new feature combines examinations for both performance and functional requirements.

JUnitPerf [JUnitPerf07] is an extension of JUnit. It is purposely designed to test the performance and scalability of functionality included within current JUnit tests. To do so, JUnitPerf allows a definition of execution time as well as a specification of the expected system load.

In spite of the power and ease delivered by these two frameworks, they are not suitable for Java3D performance testing. This is because the frameworks cannot calculate an accurate time spent on the Java3D rendering process. Let us look at an example for WEPA. A test case is created to calculate how long it takes to load a skull into the human body. **Figure 2-6** shows the test code written in JUnit.

```
.......

public class LoadOrganTest {
        private WepaMain wepaAppHandler;
        private SceneGraphContentManager sceneManager;
        .......

        @Before
        public void setUp() throws Exception {
                wepaAppHandler              =              new              WepaMain
(“file:./config/desktop_behavior.cfg”);

                sceneManager = wepaAppHandler.getManager();
        }

        //JUnit4 timed test: only passes if the test finishes in 1000 ms.
        @Test(timeout=1000)
        public void loadSkullTest() {
                // add the skull onto the human body
                AtlasManager.addAtlasObject (sceneManager, “skull.j3f”);
        }
        .......
```

**Figure 2-6: Skull loading testing with JUnit**

```
......

public class AtlasManager {

        ......

        public static void addAtlasObject(SceneGraphContentManager sgcManager,
                                        String orgNameInxStr) {

                BranchGroup branchGp = null;
                ......
                if (ext.equalsIgnoreCase("j3f")) {
                        /*
                         * read an organ's geometry data from a scene file, and
                         * create a BranchGroup object for the organ.
                         */
                        branchGp = loadJ3fFile(filename);
                }

                // process the BrachGroup object
                .......

                /*
                 * request the Java3D engine to add
                 * the BrachGroup object onto the scene graph
                 */
                objPos.addChild(branchGp);
                ......
        }
```

**Figure 2-7: Function "addAtlasObject()"**

**Figure 2-7** shows the function "addAtlasObject()" in class "AtlasManager" which has the responsibility of loading an organ into the body.

In the JUnit test, execution time of the method "AtlasManager.addAtlasObject ()" is calculated. This execution time only includes the period from reading geometry data of the skull (method "loadJ3fFile(filename)" in **Figure 2-7**) to requesting the Java3D engine to add the skull's corresponding BranchGroup object onto the scene graph (method "addChild(branchGp)" in **Figure 2-7**). Therefore, this JUnit test cannot calculate how long the Java3D engine spends in successfully adding the BranchGroup object to the scene graph, and how long it takes for the graphics card to render the actual image of skull. This is because a Java3D internal thread takes over the rendering task after "addChild(branchGp)" is called. No further rendering information can be directly acquired by the JUnit test. As a result, for Java3D applications, JUnit provides very limited help in detecting and locating performance problems. JUnitPerf handles timed tests based on the existing JUnit tests, thus it has the same drawback.

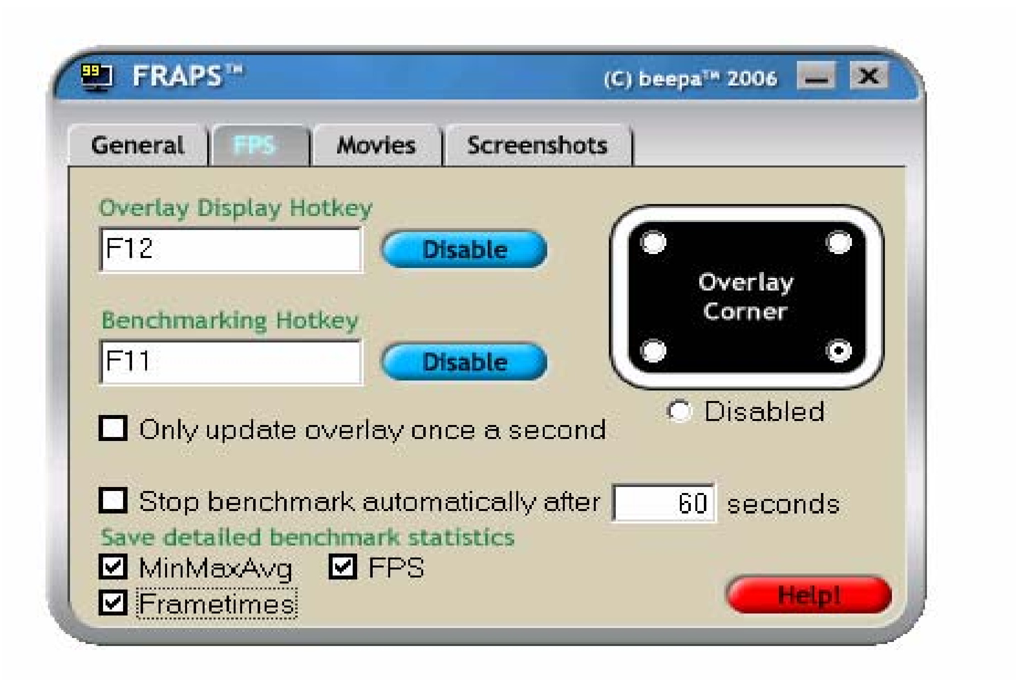
Moreover, because the two frameworks do not provide APIs to automatically trigger mouse or keyboard events, interaction tests such as zoom in a 3D scene cannot be automated. Therefore, the execution time for these tests cannot be reported.

#### 2.3.2.2 Fraps

Many times, Fraps [Fraps07] is mentioned by Java3D developers as a tool that can be used to evaluate rendering performance. It is applicable to applications using DirectX or OpenGL graphics technology and was originally designed for benchmarking computer games. With the frame rate being shown in one corner of the application's screen, one can check, at any time, how smoothly 3D objects get rendered. Basically, the higher the frame rate, the smoother the rendering is.

To get a frame rate, a user needs to press hot keys to **manually** determine starting and ending points of a capture during the run of an application. Then the user can obtain results from the generated reports. The following demonstrates an example similar to the one in section 2.3.2.1: evaluate performance for skull loading. This time, we use Fraps to investigate the frame rate during the skull loading.

**Figure 2-8** illustrates the configuration dialog in Fraps, in which we set key "F11" to set starting and ending points of a capture, and adjust the print position of the frame rate to the lower right corner of the screen. Three boxes are checked in order to examine the test results from various perspectives: “MinMaxAvg” reports the average frame rate; “FPS” lists current frame rates during testing; and “Frametimes” enumerates all the time points when a new frame is rendered.



Overlay Display Hotkey: F12
Benchmarking Hotkey: F11
☐ Only update overlay once a second
Disabled
☐ Stop benchmark automatically after 60 seconds
Save detailed benchmark statistics
☑ MinMaxAvg ☑ FPS
☑ Frametimes

**Figure 2-8: Fraps configuration dialog**

Now, we start WEPA, and perform two steps after the WEPA application window appears, which is illustrated in **Figure 2-9**:

- First step: Load a body onto the window, and press “F11” the first time to define a starting point for the test. As the sign for a successful starting point definition, a frame rate of “39” is printed by Fraps inside a green square.
- Second step: Load skull into the body, and press “F11” a second time to define the ending point. As the sign for a successful ending point definition, a frame rate of “6” is shown inside a red square.

Because the frame rate drops from “39” to “6”, we can say that the rendering becomes less smooth after loading the skull.

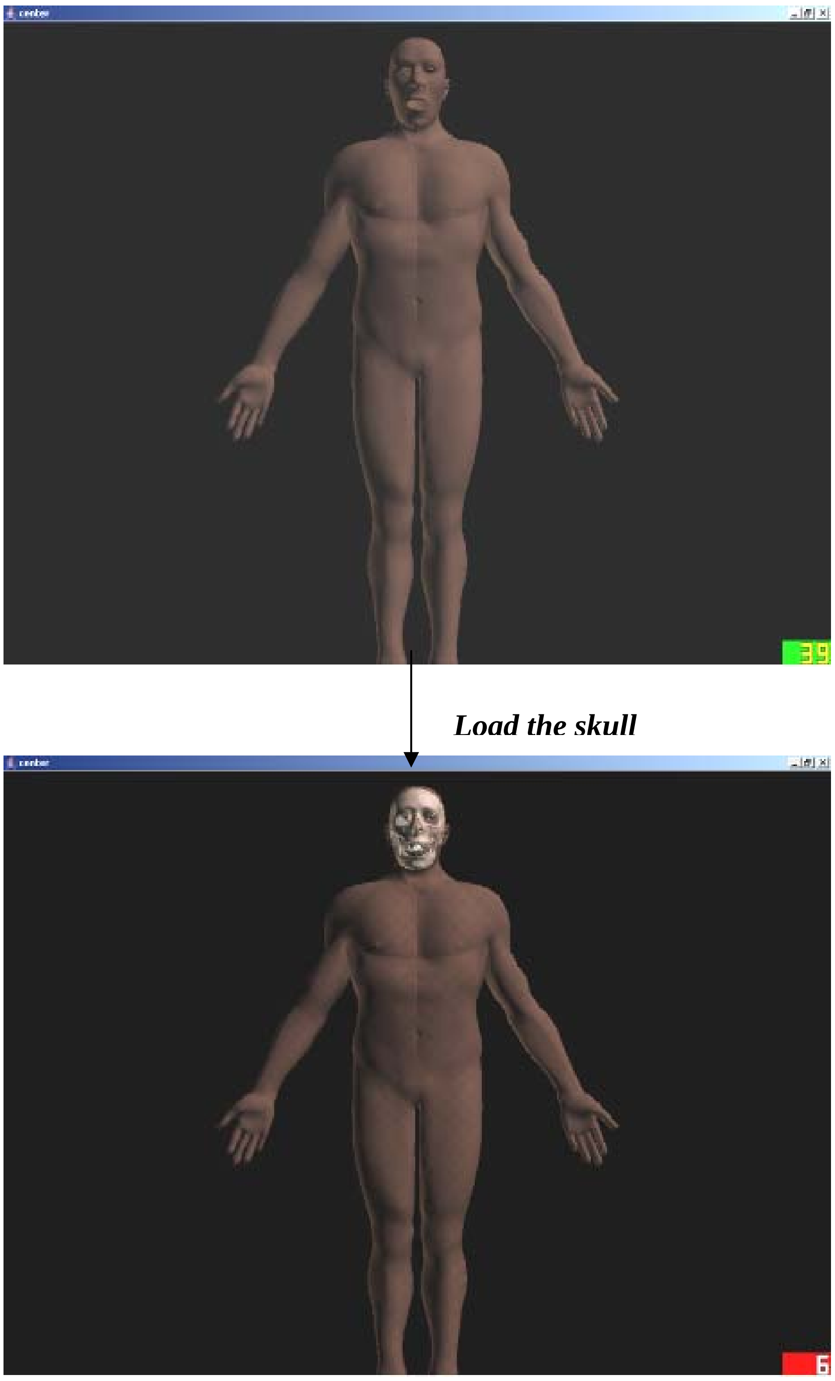


**Figure 2-9: Loading the body into the window and then the skull into the body**

To examine the test details, we can review the reports in Excel files. Table 2-1 shows the content of such a file.

**Table 2-1: Average frame rate during the skull loading**

| Frames | Time (ms) | Min | Max | Avg |
|---|---|---|---|---|
| 104 | 18503 | 0 | 39 | 5.621 |

From this table we can get the following information: during 18503 milliseconds, 104 frames are rendered; the maximum frame rate is 39, while the minimum is 0; and the average frame rate is 5.621.

Although Fraps can show and record the frame rate during application runtime, the above example clearly indicates that Fraps relies largely on user intervention (i.e. pressing hotkeys twice). This makes the accuracy of the testing reports for measuring from the start to the end of an operation highly dependent on hand-eye coordination. Delays in human reactions usually result in somewhat inaccurate testing blocks.

Besides manual operation and inaccurate test report, another major limitation is that Fraps is Windows-based only. It cannot be used on Linux or Solaris, the operating systems for many Java3D applications (and WEPA as well). Besides, with the information provided by Fraps, there is no way to tell where a performance problem originates.

As analyzed above, the testing process based on Fraps is manual and non-repeatable. Test report generated by Fraps might be inaccurate sometimes. Meanwhile, Fraps itself is platform-dependent.

2.3.2.3 Customized Testing Programs

Without a suitable tool in hand, it is not uncommon for developers to build a customized testing program for their 3D applications. However, with limited support available from Java3D, this customization requires a lot of effort and has many restrictions. For instance, the following shows a customized code fragment that calculates rendering time.

```
import java.awt.GraphicsConfiguration;
……

class MyCanvas extends Canvas3D {
        private long exeTime = 0;
        private long startTime = 0;
        private long endTime = 0;

        public MyCanvas(GraphicsConfiguration graphicsConf) {
                super(graphicsConf);
        }

        // override “preRender()”
        public void preRender(){
                startTime = System.currentTimeMillis();
        }

        // override “postRender()”
        public void postRender(){
                endTime = System.currentTimeMillis();
                exeTime = endTime - startTime;
        }
}
```

“MyCanvas” is a class extending “Canvas3D” [Java3DAPISpecification07] which is a drawing canvas for 3D rendering. “MyCanvas” overrides two methods: “preRender()” and “postRender()”. A timer starts in “preRender()”, and stops in “postRender()”. This way, the rendering time can be calculated. Unfortunately, this solution only deals with static 3D objects. It cannot be deployed to any 3D applications with animation. An animation continuously triggers “preRender()” and “postRender()”. Thus, there is no distinction if the said methods are prompted by the animation or by loading/unloading static 3D objects. Moreover, this solution may require substantial programming effort as the complexity of the application grows.

As a customized performance testing process may require a substantial coding effort in addition to the actual development, it is usually replaced by visually benchmarking and manually interacting with only one or several specific objects. Thus, only a limited scope of the application gets tested. Furthermore, in crunch time, these manual performance tests are then often neglected in the same way that manual regression testing is skipped under time pressure.

In a word, the absence of proper tools can make performance testing of Java3D applications too much effort to be continuously and consistently performed.

## 2.4 Test Automation

In that testers need to manually record testing scenarios and assess software quality, traditional software testing is labour-intense and error-prone. The emergence of test automation gave a rise to remarkable improvements, such as: reduced testing time and effort, and the convenience to run tests repeatedly. Meanwhile, inherent risks also exist. This section presents a critical discussion on good and bad aspects concerning test automation.

### *2.4.1 Benefits*

There are quite a few benefits in automating tests. The most important four are:

- **Reducing the effort to execute test suites:** Traditionally, companies hire experts to conduct extensive testing on their software system. Huge effort, such as a large number of testers and a large amount of time, is spent on creating complete test documentation, effectively coordinating test resources and on the correction of human mistakes made during testing. Not only the testing cost is considerable, but also the overall efficiency is problematic. With the support of test automation, all tests can be automatically run without human intervention, thus much less testing time is required for an extensive system testing.

- **Facilitating repeatable and continuous regression testing:** In regression testing, tests that have already passed can be reused and executed. In this way, one can check if features that have been newly implemented would trigger the bugs that previously occurred but were fixed. When performed in a repeatable and continuous manner, regression testing could provide a higher chance to regularly detect serious problems, and develop solutions in time. However, to implement this methodology using the traditional manual testing, companies need to invest more budgets into testing resources, such as testers [EET07]. Test automation changed the situation. Involving no human participation, automated regression tests can be executed as many times as wanted, send out alerts on significant problems at a much higher frequency than in manual testing, and allow developers to respond quickly to test situations.
- **Improving the code coverage:** [CodeCoverage07] summarizes the definition and categorization of code coverage: In software testing, code coverage, expressed in percentage, represents the degree (in %) to which an application source code has been tested. Generally, there are four major ways used to evaluate code coverage: S*tatement coverage* checks if each line of the source code has been executed by the tests; C*ondition coverage* examines if each evaluation point (such as a true/false

decision) has been tested; *Path coverage* investigates if every possible route through a given part of the code has been tested; and *Entry/exit coverage* checks if every possible call and return of a function has been tested. The goal of testing is to achieve higher code coverage. Nonetheless, due to the complexity involved in test case design and recording, and the large effort requied to frequently run high volume tests, manual testing evaluates only a very limited portion of the system, resulting in low code coverage. By using automation tools to record and continuously execute thousands or millions of tests, test automation allows testers to spend most of their time in generating more test suites. This leads to a higher percentage of tested application code, which improves code coverage.

- **Allowing for high volume performance and load testing:** For projects whose key success factor is performance, a high volume performance and load testing plays a vital role. 3D applications, for example, are required to keep stable when numerous 3D objects or user interactions are being processed. Being automated, existing test cases can be conveniently repeated several times to simulate a rising number of 3D objects or user interactions. This provides information on how an application can still perform well under increasing workload.

### *2.4.2 Risks*

Regardless of the mentioned benefits, there are also significant risks that comes along with test automation [Tamres02]:

- **Getting familiar with a new testing tool may require a steep learning curve:** Each testing tool has it own requirements on how to design test cases, how to organize test cases into test suites, how to execute test suites and how to interpret test results. In addition, test scripts in some tools are written in programming languages, which requires some programming background. As such, there may be a steep learning curve for testers who are not acquainted with coding, or who have never used a particular tool before. Also, if high employee turnover occurs, frequent training for new testers is not only time-consuming but also expensive [Pettichord01].

- **Testing tools have bugs and limitations:** Testing tools are software in nature. Therefore bugs and limitations are inevitable. These defects may impact the testing quality [Tamres02]. For instance, when a tool fails to report serious bugs, we miss an opportunity to fix errors. As a result, serious problems may accumulate to an extent where they are difficult to resolve. Besides, if test execution is too slow, developers may prefer to delay the current testing and write more code, especially when there is a deadline. Unfortunately, many code changes may have taken place during this postponement of testing. Even when testing is finally done, "debugging becomes more difficult because if tests fail after a series of changes, identifying the guilty change is difficult" [Smith et al.01], and it is very possible that developers forget what changes they made. "Murphy's Law says that this forgotten change will most likely be the source of the failing tests" [Smith et al.01].
- **Maintaining test suites may demand a big effort:** Tools can automate test execution, but they cannot define or create test scripts without human participation. Significant workload is needed to develop the tests. The maintenance effort would grow fast if code keeps changing. This is because not only the existing test suites need revised but also new tests need to be developed. Even experienced programmers or testers need considerable time for these tasks.

## 2.5 Chapter Summary

This chapter covered some background on Java3D and test automation. Namely, the working mechanism involved in Java3D was introduced; how to build a scene graph to render 3D objects was explained; an extensive discussion on Java3D performance issues was given, and the current tool support for its testing was reviewed. Based on this discussion, it was found that the existing testing tools and approaches offer very limited help in effectively conducting a repeated and continuous performance testing on Java3D applications. This motivated my work; at the end of this chapter, a critical overview into the pros and cons of test automation was given.

The next chapter delves into the details of my pre-development survey with graphics experts, uncovering more about the state of art in Java3D performance testing and the most desired testing metrics for new tool support.

# Chapter Three: A Survey on 3D Performance Testing

We carried out a survey to improve our understanding of the needs for performance testing of Java3D applications. The main results are:

- The survey responses pointed out that at present only insufficient tool support exists for Java3D performance testing (confirming our own investigations into the subject);
- The survey provided a guideline that resulted in a set of basic requirements for J3DPerfUnit development.

This chapter[2] discusses the survey details. I first present objectives of the survey and how it was designed. Next, participants' backgrounds are discussed. Then, survey results are revealed and analyzed. Finally, I discuss the survey limitations.

## 3.1 Objectives and Survey Design

On the whole, the intention behind this survey was to acquire broader insight into current status in 3D performance and its testing, and to collect fundamental tool requirements.

Specifically, the survey tried to solicit answers for four questions:

- What are the performance issues commonly encountered during 3D applications development?
- What test cases have been built to detect these issues?
- What are the metrics recommended for 3D performance testing?

[2] Section 3.2 & 3.3 reference the following publication:
Xueling Shu, Frank Maurer; A Tool for Automated Performance Testing of Java3D Applications in Agile Environments; Proc. of the Second International Conference on Software Engineering Advances (ICSEA 2007), IEEE 2007 (6 pages).

- Lastly, how is the testing executed, manually or with tools? If with tools, what are they, and their features and limitations?

Answering the above questions would help gather essential information that would generate valuable input for the J3DPerfUnit development, and to reassess the usefulness of my research.

The questionnaire (Appendix A.2.) includes multiple choice and open ended questions. Some questions have "N/A" (Not Applicable) as an option because the questions are only suitable for certain situations. For instance, participants were asked to provide names of the testing tools if they did use tools. In this case, no information should be expected if testing was done manually. In case of a need for a clarification of their replies as well as a possible instigation of more information, all the participants were also asked to decide whether they would like to have a follow-up interview or not.

Starting from September 2006, questionnaires, along with an introduction offering a brief overview of my work (0 Survey Introduction), were emailed to Java3D forums, the WEPA team (four developers), and the Interactions Lab (four developers) and the EBE [EBE07] lab (one developer) on campus. Over the next two months, from September to October, I collected all of the responses, conducted follow-up interviews, and then performed the analysis. For the follow-up interviews, I emailed questions to the participants. All of them except one replied.

## 3.2 Participants

The survey was sent to team members of the WEPA project, Master and PhD students in the Interactions Lab and EBE lab on campus, and developers from two major Java3D forums. Scarce interest from the two Java3D forums unfortunately resulted in no involvement from them, but we still received useful information from the first two resources. They were also good representatives of the population developing 3D applications. The reason is that these participants had been programming with either Java3D or OpenGL for 1 to 13 years.

In total, nine participants received the questionnaire. All provided valid responses. In this chapter, these responses will be used to show the results. Note that each survey reply was presented from an individual perspective, and therefore is subjective in nature. Likewise, because it was I who analyzed the entire survey, the analysis made in this chapter is also personal. However, I would argue that the survey analysis is still valid. This is because the analysis is based on the feedback from a group of developers who were working on 3D applications. These developers had knowledge about various performance issues and had experiences in performance testing.

We took two 3D programming experiences into account: OpenGL and Java3D. There are two reasons for this. Firstly, OpenGL is one of the graphics libraries used by Java3D. Secondly, OpenGL developers tend to be knowledgeable about low level rendering details, and would therefore have a good knowledge of 3D performance. Involving developers from both areas is more likely to produce a complete set of ideas concerning 3D performance issues and testing. Although the other version of Java3D, DirectX is also a robust software interface for graphics devices, we did not count experience with it as a prerequisite for participation. DirectX only works on the MS Windows operating system, and thus has a narrower application scope and is less popular within the Java3D programming community.

Table 3-1 outlines the participants' experience with Java3D and OpenGL in years.

**Table 3-1: Years of Experience with Java3D and OpenGL**

| Developers | Java3D Experience | OpenGL Experience |
|---|---|---|
| Developer 1 | 0 | 1 |
| Developer 2 | 0 | 11 |
| Developer 3 | 0 | 13 |
| Developer 4 | 1 | 0 |
| Developer 5 | 1 | 0 |
| Developer 6 | 1 | 3 |
| Developer 7 | 3 | 1 |
| Developer 8 | 3 | 0.5 |
| Developer 9 | 4 | 2 |

With respect to Java3D programming, among the nine survey respondents, three had no experience. Three spent a solid one-year period developing applications with Java3D. The other three respondents had three to four years experience working with Java3D technology.

Concerning OpenGL programming, two survey respondents had no experience; three had either half of or one year experience using OpenGL for 3D implementations. Another two respondents had two to three years of experience working with OpenGL technology; the remaining two had eleven and thirteen years experience.

In addition, four participants had a combined experience with Java3D and OpenGL. Among them, one was experienced with both languages; the others were just getting started with either language.

## 3.3 Results

### *3.3.1 Performance Issues*

The commonly identified issues in the survey are:

- Loading and unloading complicated 3D objects, such as those with a high polygon count and textures, tend to be very slow, memory costly, and generate a low frame rate. For example, one reply emphasized the influence of a high polygon count as this:

  *"Loading large 3D objects tend to be slow, and subsequent operations become rather cumbersome… Many surface-based 3D models are made with the anatomical precision in mind, and therefore require a very high polygon count, which affect their performance"*— Developer 4 (see Table 3-1)

- Interacting with a 3D scene has a long lag time and delivers a low frame rate. Usually, this situation happens when a 3D scene has objects described by a large set of geometry data.

- Various hardware cofigurations result in diverse application performance. For instance, one respondent remarked that:

  *"The same Java3D application runs faster on a Windows machine than on SunRays"*— Developer 6 (see Table 3-1)

- Lastly, implementation limitations in OpenGL and Java3D also cause performance drops. For example, one developer quoted that:

  *"Java3D does not do the computations in hardware, so it is slower compared to OpenGL"*— Developer 6 (see Table 3-1)

### *3.3.2 Performance Requirement and Testing*

Table 3-2 reports the responses on how 3D performance requirements are collected.

**Table 3-2: How 3D Performance Requirements Are Collected**

| Requirement Types | Number of Responses |
|---|---|
| No Requirements | 2 |
| Formal (i.e. requirement specification) | 2 |
| Informal (i.e. user discussion) | 5 |

Table 3-3 and Table 3-4 show the responses about how 3D performance testing was conducted.

**Table 3-3: Unit Testing Types**

| Unit Testing Types | Number of Responses |
|---|---|
| No Unit Testing | 3 |
| Manual (i.e. debugging) | 4 |
| Automated | 1 |
| Manual & Automated | 1 |

**Table 3-4: User Acceptance Testing Types**

| User Acceptance Testing Types | Number of Responses |
|---|---|
| No User Acceptance Testing | 3 |
| Manual | 6 |

Among the nine participants, only two stated they had formal requirements, and officially performed either unit or user acceptance testing. In particular, one of them mentioned that their unit testing was automated. Another five respondents claimed requirements were solicited from informal user discussions, and either manual unit or user acceptance testing were conducted thereafter. Nonetheless, a later interview uncovered that in the case of one of these five respondents, the statement was only a future plan the participant thought of. The actual user discussions and performance testing never happened. The remaining two participants reported that neither formal nor informal performance requirements existed for their applications. However one of them said both manual and automated unit testing had been performed on the developer’s own initiative.

### *3.3.3 Test Cases*

To develop a tool that can effectively measure 3D system performance, understanding its test case design has a significant value. Therefore, we asked the participants to list one or more performance test cases that were used to evaluate their applications. Originally, only three participants listed their test cases. In the follow-up interviews, it was uncovered that among the six participants who did not provide test cases three had not done any formal performance testing yet. The remaining three did the testing, but had no formal test cases. After further discussion, these six developers affirmed that their answers to “testing metrics” (Question 13) and “desirable features” (Question 14) could describe what they would include in the test case design. For this reason, I considered the responses to Questions 13 and 14 when analyzing the responses that address test cases.

Typically, the participants would like to define their performance test cases as this: evaluate execution time or frame rate during two operations. One is loading/unloading 3D objects. The other is interacting with 3D objects, such as zooming, rotate and translate. Here is one participant’s reply:

*“A specific data set is used with the required load and various interaction is attempted on a typical users machine.”*— Developer 3 (see Table 3-1)

To be clear, the examples given in the survey could be summarized as:

- Apply a scenario on specific test scenes with a required load. For example, load the heart, which has 3000 polygons, into the human body. A test scene can refer to a 3D view, which in this case is the one displaying the human body.

- Attempt various interactions with the scene. For instance, rotate the scene which displays the human body with its heart.
- Run the same application under different operating systems and hardware environments. Compare their results.
- Check specific performance metrics, such as: investigate what the frame rate is during the rotation; or examine the time to load an organ image onto a screen.

Several words in the examples appeared repeatedly and thus grabbed my attention: “load”, “interactions” and “statistics”. They provided a clue for what parameters should be used when designing a performance test case. A further investigation into the developers’ responses disclosed a similarity in the meanings of these words: “load” refers to the dataset size of objects, for instance, polygon counts or pixel numbers; “interactions” means zooming, rotating or translating a scene or 3D objects with mouse and/or keyboard; and “statistics” mainly covers two metrics: execution time and frame rate. In the following section, we will list other metrics cited by participants.

### *3.3.4 Testing Metrics*

Participants suggested seven important metrics to evaluate 3D performance. They are listed in Table 3-5.

**Table 3-5: Performance metrics**

| Metric name | Number of affirmative responses |
|---|---|
| Loading/unloading time of objects | 6 |
| Interaction or lag time | 6 |
| Frame rate | 4 |
| Memory usage | 3 |
| Maximum Polygon & Object Counts at X | 1 |
| Scene graph size | 1 |
| Milliseconds/frame | 1 |

The column “Number of affirmative responses” highlights number of participants asking for a specific measurement. As the table shows, “loading/unloading time of objects”, “interaction time or lag time” and “frame rate” are three top most identified metrics.

“Loading time of objects” refers to the time spent from the point when an application starts reading a 3D object data from files or memory to the point when the object is displayed.

“Interaction time or lag time” means the time from triggering an interaction, such as: rotation, to observing the final view. A longer lag time indicates a larger system delay to handle mouse/keyboard commands, which is directly reflected as a less smooth rendering.

“Frame rate” represents the number of frames produced by a graphics device every second. In general, the higher the frame rate, the smoother the rendering is.

“Memory usage” is another broadly cited standard of measurement. Using too much memory dramatically reduces the system response capability and introduces performance drops.

The remaining metrics in Table 3-5 are more individual preferences as each one was only called for by a single respondent.

More like a system load test, “maximum polygon & object counts at X fps (frame per second)” investigates what the maximum polygon and object counts an application can handle while its rendering quality is required to keep at a certain frame rate.

“Scene graph size” informs developers of how big a scene graph is after loading a 3D object. The larger the scene graph, the more likely the application runs slowly.

Lastly, “milliseconds/frame” is to examine frame rate from another perspective.

### *3.3.5 Testing Techniques*

As demonstrated in Table 3-3, out of nine participants only two stated that their performance unit testing was automated. The rest claimed either no testing or manual testing. I was very interested in responses from the two respondents who reported that their performance testing was automated. I tried to conduct a follow-up interview to dig out more details about the performance testing tools used by the two participants.

The first developer stated JUnit is their tool to conduct performance testing. It turned out that the developer actually refered to Jemmy, a Java2D testing framework. As their application was a hybrid of Java2D and Java3D, they automated 3D test cases by triggering 2D events. Because Jemmy was originally designed to trigger events on 2D controls to simply examine functional requirements for Swing applications, it provides limited power to explore Java3D performance. For example, Jemmy can automatically select and open a 3D file from a dialog, but it cannot offer any further information on 3D performance, such as: how long the file is analyzed, when the corresponding BranchGroup object is created, when the Java3D engine attaches this BranchGroup to the scene graph, and how long the graphic device spends in producing the actual image. Furthermore, one cannot use Jemmy to automate mouse/keyboard interactions and get performance statistics.

The second developer shed light on the power of their auto-tester.

*“We used a custom written auto-tester that started a sample-app and recorded a bunch of statistics for test scenes. This worked well as long as we tested enough scenes and the scenes did not change from one test to the next”*. – Developer 1 (see Table 3-1)

Nevertheless, the developer did not give any further response, thus no details could be obtained. Yet based on the description, this auto tester could only benefit the implementation the respondent worked on and yielded other restrictions. For example, instead of directly testing the real application, a “sample-app” needs to be built first and the scenes were required not to change from one test to the next.

## 3.4 Analysis

The survey disclosed four realities of participants’ 3D performance testing:

### *3.4.1 Testing is not routinely conducted*

Regular and continuous performance testing is critical to the quality of 3D applications. However, as uncovered by the seven respondents in section 3.3, either no performance testing was performed (two responses), or it happened “passively” (seven responses). “Happened passively” means that instead of on the development teams’ own initiative, performance testing was conducted only after users detected performance problems.

One of the possible reasons why a routine performance testing rarely happened might be the fact that most tests were done manually. One respondent’s answer supports this assumption:

*Manual testing takes a lot of time, also it is hard to remember at the end of a session exactly when the frame rate dropped, lag time increased, etc.* — Developer 7 (see Table 3-1)

An automatic testing tool, if designed appropriately, can help solve these difficulties. Another reason for a irregular testing might be people’s attitudes towards whether or how the testing should be done. This part cannot be handled by tool automation.

### *3.4.2 Test cases include two major aspects*

One test case is to investigate the rendering speed when loading/unloading 3D objects onto/from 3D scenes. The other is to examine how smoothly images are displayed when using mouse/keyboard to perform interactions, such as zooming, rotating, and translating, on a 3D scene or individual 3D objects. Covering most of the aspects examined in 3D performance testing, these two test cases can effectively evaluate how well 3D applications respond to user actions.

In reality, automating interactions with individual objects are much more complicated than interactions with a 3D scene. For this reason, interaction tests with a 3D scene, along with objects loading/unloading tests can be the starting point of a testing tool.

### *3.4.3 Testing involves two major metrics*

They are execution time and frame rate.

Showing the rendering speed straight, execution time becomes the most cited standard for evaluating performance in 3D applications. Through measuring this metric, one can easily tell if it lasts too long to perform an action.

Calculating how many frames, or images, are rendered every second, frame rate also reflects 3D applications performance. In general, 30fps is equivalent to movie performance. A lower frame rate value indicates a non-smooth rendering.

For a performance testing tool, these two measurements should be taken into account when test cases are defined.

### *3.4.4 There is very limited tool support*

According to the survey, participants' performance testing, if done, was conducted manually. Manual testing is time-consuming, and tends to introduce errors, such as incorrect input data or test scenario. Consequently, it makes testing inefficient and ineffective. Although some respondents claimed automated testing was adopted, their tools were either not designed for 3D performance testing, or had big limitations.

This confirmed my own research result that there was very limited tool support for automated 3D performance testing. At the same time, this also strengthens the significance of my work.

## 3.5 Limitations in the survey

The responses from graphics experts provided a richer context for the current state of 3D performance testing than my own experiences alone. The survey responses also presented guidelines for J3DPerfUnit development, and emphasized the importance of my research. However, it may suffer from a few limitations:

- The sample size was small. Initially, I posted my invitations on two major Java3D forums and expected participants from there, but in the end no replies were received. The final data includes 9 respondents and the analysis only incorporates them. Thus, the information collected may be insufficient, and my conclusion may not be generalizable.
- One third of the participants had no programming experience with Java3D and another one third had only 1 year experience. They either did not have a clue of or were not confident in how Java3D performance testing should be done. Therefore, the value of their responses was somewhat limited. Furthermore, because they covered most of the survey population, the overall survey quality was influenced. Also, no concrete analysis was done on the variances of their restricted experience and how it affected the value of their answers.

- Data on some follow-up interviews was not taken. Some participants were not willing to be interviewed in person or by telephone. Thus, the way I tried was to use emails. Unfortunately, some interview data was accidentally deleted from the mailbox. Or, I filtered out the data that was not considered useful at that point, but later I determined it was important in terms of writing my thesis.
- The participants' opinions and my analysis may be biased. Each participant provided their feedback from an individual perspective, and thus is subjective in nature. Similarly, because the survey was analyzed by me alone, the analysis is also personal. All these personal opinions may generate biases.

## 3.6 Chapter Summary

This survey was conducted to explore the current status of 3D performance testing, and help generate development guidelines for J3DPerfUnit. University of Calgary granted an ethics approval and nine subjects were involved. The result revealed important 3D performance test cases and metrics, and affirmed my own investigation that existing tools very limitedly support a fully automated 3D performance testing. Limitations of the study are recognized as the size of the survey population, the familiarity with Java3D programming, and the incomplete data set from subjects.

# Chapter Four: Requirements Analysis and Tool Design

As shown in Chapter Three, there exists very limited tool support for the automation of Java3D performance testing. To improve the situation, a new testing tool called J3DPerfUnit was developed. In this chapter, I will first analyze the tool requirements. Then I will explain the tool design by showing the high-level architecture and demonstrating the execution process.

## 4.1 Requirements Analysis

Based on the survey analysis in Chapter Three, the current support provided by Java3D, and basic functionalities of a unit testing tool, the following tool requirements were generated:

- **Reporting the execution time to load a 3D object onto a scene graph.**

  For 3D object loading, two kinds of execution time are needed: (1) the total time spent on the object loading pipeline demonstrated in **Figure 2-3**; and (2) time spent on each stage of the loading pipeline. The second kind of execution time helps developers locate where a performance problem comes from.

  For the third and fourth stage of the pipeline, Java3D does not offer any related time information (see Table 4-1).

**Table 4-1: Time not calculated by J3DPerfUnit (For the object loading pipeline)**

| 3rd stage of the loading pipeline | 4th stage of the loading pipeline |
|---|---|
| how long it takes for graphics libraries to receive a rendering command | how long it takes for a graphics card to render the actual image of a scene graph |

As a result, this requirement was changed to: (1) calculate the total time spent on the first two stages of the object loading pipeline; and (2) respectively calculate the time spent on the first and second stage of the loading pipeline (see Table 4-2).

**Table 4-2: Time calculated by J3DPerfUnit (For the object loading pipeline)**

| 1st stage of the loading pipeline | 2nd stage of the loading pipeline |
|---|---|
| how long it takes for a Java3D application to analyze geometry data and generate a corresponding BranchGroup object | how long it takes for the Java3D rendering engine to process and optimize the data, and place the BranchGroup onto the scene graph |

- **Reporting the execution time to unload a 3D object from a scene graph.**

For 3D object unloading, two kinds of execution time are needed: (1) a total time spent on the object unloading pipeline demonstrated in **Figure 2-4**; and (2) time spent on each phase of the unloading pipeline.

Similar to the first requirement, Java3D does not give any related time information for the third and forth phase of the unloading pipeline (see Table 4-1).

Thus, this requirement was changed to: (1) calculate the total time spent on the first two phases of the object unloading pipeline; and (2) respectively calculate the time spent on the first and second phase of the unloading pipeline (see Table 4-3).

**Table 4-3: Time calculated by J3DPerfUnit**

**(For the object unloading pipeline)**

| 1st phase of the unloading pipeline | 2nd phase of the loading pipeline |
|---|---|
| how long it takes for a Java3D application to send a BranchGroup remove command to the Java3D Rendering Engine | how long it takes for the Java3D rendering engine to remove the BranchGroup from the scene graph |

- **Reporting FPS (frames per second or called frame rate) during user interactions with a 3D scene.**

  For example, when rotating a 3D object with mouse, the tool should calculate the average FPS from the point when a mouse button is pressed, and the time during which the mouse is moved, to the point when the mouse button is released. A low frame rate value indicates a non-smooth rendering during the rotation. Execution time is not selected to measure the interaction performance. This is because Java3D does not provide any API that helps get such time information.

- **Reporting the time to build a Java3D scene graph.**

  This can expose problems with scene graph design. When a scene graph is not constructed properly, displaying a 3D scene can be extremely slow or simply fail.

- **Generating a summary on the command line after each test runs.**

The summary should include the number of passing and failing test cases, and the names of failed test cases.

- **Providing a detailed report for each test**.

  This report lists the detailed result for each test case. The result not only indicates whether a test case passes or fails, but also shows asserted and actual value of the metric (either execution time or FPS) used to evaluate the test case.

## 4.2 Tool Design

J3DPerfUnit consists of five major components: J3DPerfEngine, J3DMonitor, AWTMonitor, SceneGraphMonitor, and TestThread. They are introduced respectively below. Meanwhile, to give readers a better understanding of how these five components work together, an example is presented afterwards.

### *4.2.1 The High Level Architecture*

**Figure 4-1** illustrates the tool's high-level architecture.

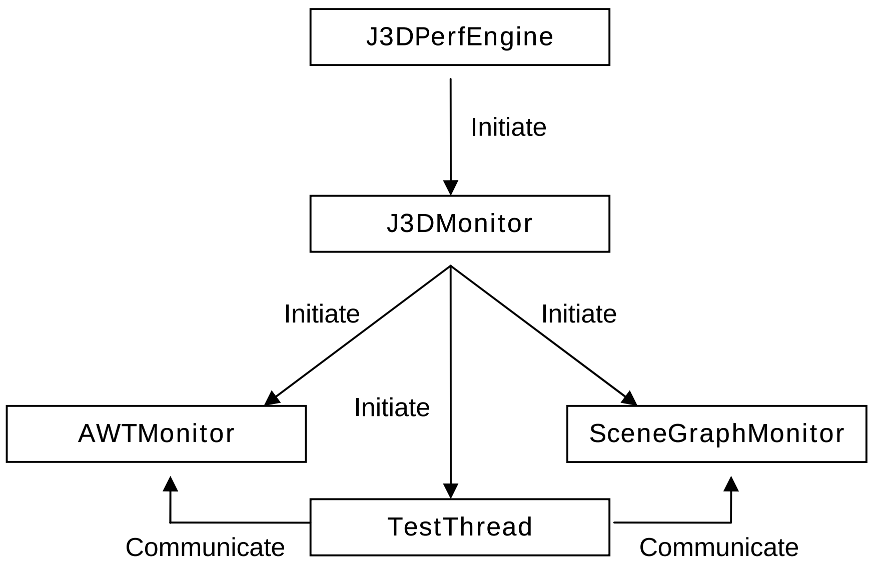


**Figure 4-1: The five components in J3DPerfUnit and their relationship**

J3DPerfEngine has five responsibilities:

- Set up a test environment, for example, record a test class name. The name will be taken as file name of the corresponding test report.
- Instantiate a new test case.
- Trigger J3DMonitor which registers a newly created test case.
- Start and stop the frame rate counter. The counter generates frame rate statistics for interaction tests.
- Terminate the test execution when the last test case is reached.

J3DMonitor implements two aspects:

- Test case registration. In this process, a new test case will be put into the test execution queue and begins to run. The test execution queue holds all the test cases that are handed over to J3DMonitor after being initiated by J3DPerfEngine.
- Startup of three components: AWTMonitor, SceneGraphMonitor and TestThread.

AWTMonitor observes the events pertinent to window opening and closing. A window instance is kept when the window is open, and is destroyed when the window is closed. Holding the instance of an opened window is important as J3DPerfUnit needs it to locate scene graphs and perform mouse/keyboard automation.

SceneGraphMonitor captures two kinds of user events on the opened windows:

- Adding or detaching 3D objects onto/from scene graphs.
- Mouse/keyboard press, release and scroll on 3D canvases on scene graphs.

Lastly, TestThread performs two duties:

- Coordinates and executes test cases in the test execution queue by communicating with both AWTMonitor and SceneGraphMonitor.
- Reports test result and generates test reports.

### *4.2.2 Tool Execution Process: An Example*

To demonstrate J3DPerfUnit execution process, we present a sample test case which evaluates the frame rate during a scene zooming through mouse scroll. We assume this sample is the first test case specified in its test class.

**Figure 4-2** illustrates the execution process composed of nine steps. Detailed explanations for each step are provided right after the figure.

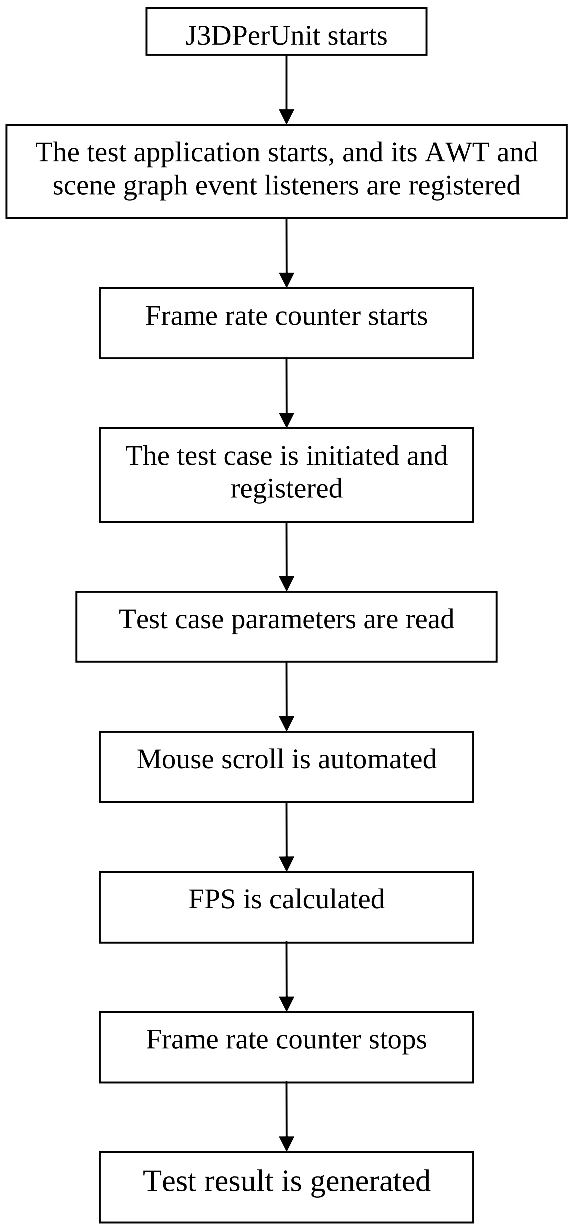

J3DPerUnit starts
The test application starts, and its AWT and scene graph event listeners are registered
Frame rate counter starts
The test case is initiated and registered
Test case parameters are read
Mouse scroll is automated
FPS is calculated
Frame rate counter stops
Test result is generated

**Figure 4-2: Nine steps in the execution process**

- **Step One: J3DPerfUnit starts**

  J3DPerfEngine starts, and records name of the test class. Then J3DMonitor, AWTMonitor, SceneGraphMonitor and TestThread begin running.

- **Step Two: the test application starts, and its event listeners are registered**

  The java class ClassRunner is used to execute the test application automatically. When windows of the application open, AWTMonitor captures the events and records window instances. Next, AWTMonitor notifies SceneGraphMonitor to register scene graphs event listeners for 3D object adding/detaching and mouse/keyboard operation.

- **Step Three: frame rate counter starts through J3DPerfEngine**
- **Step Four: the test case is initiated and registered**

  When all the setups are ready, J3DPerfEngine instantiates the test case which then is put into the execution queue in TestThread by J3DMonitor.

- **Step Five: test case parameters are read**

  TestThread reads the parameters, such as: on which canvas of which window this test case executes, where on the test canvas the mouse is placed, how many notches the mouse scroll should trigger, etc.

- **Step Six: the mouse scroll is automated**

  TestThread finds the test window and canvas by contacting AWTMonitor and SceneGraphMonitor. After this, TestThread automatically places and scrolls the mouse to zoom the test scene.

- **Step Seven: FPS is calculated**

TestThread captures the start and end point of the testing period by communicating with SceneGraphMonitor. After zooming is finished, TestThread checks the data sent by the frame rate counter, and calculates the frame rate.

- **Step Eight: frame rate counter is stopped by J3DPerfEngine**
- **Step Nine: test result is generated**

  Test result is printed on command line, and a test report is created.

All the test cases have a similar execution process.

# Chapter Five: J3DPerfUnit Usage

J3DPerfUnit is a tool that automates Java3D performance testing. Performance test cases are created with the tool's predefined APIs. Test results are displayed right after a test class is executed. Test details can be investigated using generated reports. Moreover, an ant task is produced to facilitate running J3DPerfUnit tests in Apache Ant, a Java-based build tool [Ant07]. What follows in this chapter[3] includes five sections. Section one takes the reader through an overview of the tool features. Section two and three demonstrate how to conduct Java3D performance testing with J3DPerfUnit. Section four looks at how to write a J3DPerfUnit ant task. The state of the tool implementation is briefly discusses in Section five. Finally, the chapter concludes with a summary.

## 5.1 An Overview of Tool Features

My tool development was carried out in Eclipse, an open source integrated development environment created by IBM [Eclipse07]. The reason for this choice as the development environment was: 1) it offers a user-friendly interface; 2) it provides a SVN plug-in [Subclipse07] to allow checking in/out my project conveniently. J3DPerfUnit source code is stored under the MASE project on SourceForge [Sourceforge07] , an open source projects repository [MASE07].

This section gives an overview of the tool features, including

- Java3D performance metrics evaluated by the tool
- How to set up performance test cases

[3] This chapter reference the following publication:
Xueling Shu, Frank Maurer; A Tool for Automated Performance Testing of Java3D Applications in Agile Environments; Proc. of the Second International Conference on Software Engineering Advances (ICSEA 2007), IEEE 2007 (6 pages).

- How to read test results and reports.

All the code examples in this chapter are extracted from tests for the WEPA application.

### *5.1.1 Metrics*

J3DPerfUnit can evaluate four aspects of Java3D application performance:

- Execution time to build a scene graph;

  This aspect describes how long it takes to display a scene graph right after the application starts.

- Execution time to load a 3D object onto a scene graph;

  As discussed in Chapter Four, this execution time includes two parts: (1) a total time spent on the first two stages of object loading pipeline demonstrated in **Figure 2-3**; and (2) the individual time spent on first and second stage of the loading pipeline

- Execution time to unload a 3D object from a scene graph;

  This execution time also includes two parts: (1) a total time spent on the first two phases of the object unloading pipeline demonstrated in **Figure 2-4**; and (2) the time spent respectively on the first and second phase of the unloading pipeline

- Frame rate during user interactions.

  From this perspective, one will know how smooth the rendering is when a 3D scene is rotated, zoomed or translated with mouse/keyboard.

### *5.1.2 Test case definition and execution*

There are three steps a user needs to follow to define and execute a new test case:

- Create an instance of the class “ClassRunner”. Depending on the type of test, this class runs an application automatically before or after the next three steps described below. For example, when testing the performance of loading a 3D object, “ClassRunner” should be started before creating the test case. This is because object loading usually happens after an application starts. When testing scene building time, the runner should be started afterwards. This is because scene building takes place right within the starting process of an application.
- Create an instance of the class “TestCase”. This class is where a Java3D performance test case is defined.
- Specify test parameters for methods in “TestCase” (Table 5-1 gives a brief explanation of the parameters used to describe test cases).
- Define the method whose name starts with “assert” after all the other methods. An “assert” method states an assertion for either execution time or frame rate. The test case will be executed whenever its corresponding “assert” method is called.

**Table 5-1: Test parameters**

| **Parameter** | **Meaning / Definition** |
|---|---|
| Test case name | The name of a test case. This name will be displayed in test reports which are local files generated by J3DPerfUnit. When the test case fails, the name is also printed out on command line. |
| The title of a test window | The title of a window where a test scene is contained. This title is displayed in the window’s border. A test scene |

| | |
|---|---|
| | refers to the 3D scene described by a Java3D scene graph. |
| Index of test canvas | Test canvas index on the test window. A canvas is a blank rectangular area on a window. In this rectangular area, one can make drawings or trigger UI events. A 3D scene is always held by one canvas on a window. Because in Java3D multiple canvases can exist on a window at the same time, we need to point out which canvas is under testing. |
| BranchGroup identifier | Either the name or the user data of a "BranchGroup" instance. The Java3D class called "BranchGroup" describes one or more 3D objects. In Java3D, the only way to identify different "BranchGroup" instances is to specify particular names or user data for these instances. |
| Time assertion | Maximum time for a successful test |
| Frame rate assertion | Minimum frames produced per second for a successful test |
| Interaction data | The data used to trigger events in a 3D scene. For example, a tester can define which mouse or keyboard button to press, or specify start and end point for mouse movement, etc. The interaction data allows a test case to well simulate the way a human operates mouse/keyboard. |

### *5.1.3 Test Result*

Execution result of a test class is available in two formats. One is command line messages, which are similar to those printed by JUnit. The other is local files, which are generated by J3DPerfUnit to store more details of the test outcome.

Command line messages include three pieces of information. Firstly, the name of the executed test class is shown. Secondly, the number of failed test cases is reported. Lastly, a name list of the failed test cases is displayed.

In local files, developers can examine more test result details. Those details encompass the information such as the user events that triggered the test, execution time or frame rate, and probable causes of the performance breakdown for reference.

## 5.2 Evaluate Execution Time

### *5.2.1 Build a 3D Scene Graph*

As the soul of a Java3D application, a scene graph must be constructed properly. Otherwise displaying a scene can be extremely slow or simply fail. In order to expose probable problems with scene graph design, I provide a test case that can explore execution time to build (display) a scene.

The black screen in **Figure 5-1** is the empty 3D scene displayed right after WEPA is started. The display time will be investigated by a scene build test. To define such a test, we need two parameters:

- The title of the test window, which, in our case, is “center” specified in the method “testSceneBuild”. J3DPerfUnit uses this parameter to locate the scene where the test case should execute;

- The maximum execution time for passing the test which is 2000 milliseconds for this example. This time assertion is specified in the method “assertExeTimeEquals”.

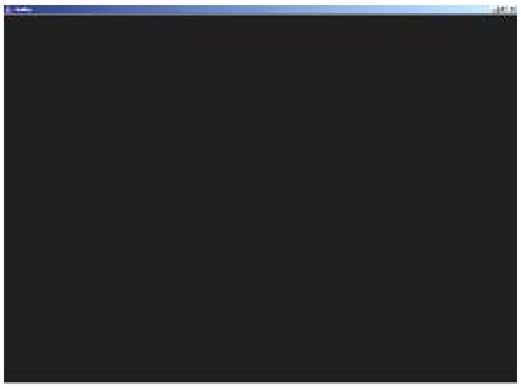


**Figure 5-1: The 3D scene in the WEPA application**

Concretely, the test code is:

```
TestCase testcase1 = j3dTest.getTestCase(secenBuildForWEPA);
testcase1.testSceneBuild ("center");
testcase1.assertExeTimeEquals (2000);
classRunner.runApplication ("WepaMain");
```

The result is displayed on the console as:

```
[J3DPerfUnit] Running 1 test in “ca.ucalgary.wepa.tests.gui.SnBuildTest”
[J3DPerfUnit] Fail (0 tests)
```

This indicates the WEPA 3D scene was successfully shown within 2 seconds.

### *5.2.2 Load a 3D Object*

Intricate 3D objects always have a complicated geometry structure or rich textures resulting in a high polygon count. Loading these objects requires a large data set to be processed, which tends to produce performance issues

The first test in this section checks whether loading a skeleton into a human body can be finished in 2 seconds or not. Screenshots are provided in **Figure 5-2**, demonstrating the scenes before and after the skeleton is loaded.

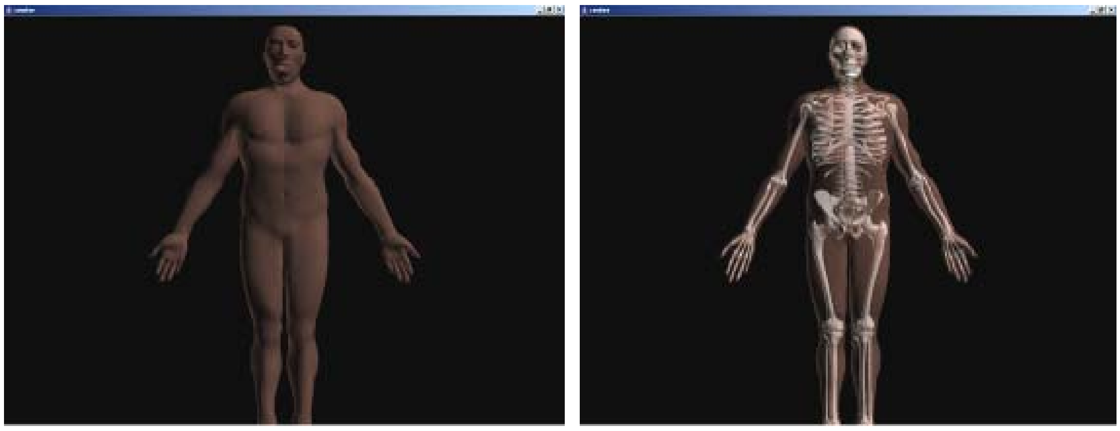


**Figure 5-2: Screenshots --- before (left) and after (right) the skeleton is loaded**

For this test, two parameters are used to locate the test canvas:

- The title of the test window (“center”);
- The test canvas index (“0”), indicating it is the first canvas on the test window that our test will be executed on.

Two parameters are used to specify the BranchGroup to be loaded:

- The BranchGroup name or user data. In this test, we use the name of the BranchGroup representing the skeleton (“skeleton”);
- An identifier indicating whether we use name or user data of the BranchGroup to be loaded. J3DPerfUnit defines two constants to represent this identifier: BranchIdentifier.BRANCH_NAME and BranchIdentifier.USER_DATA

Lastly, one parameter holds the value for the maximum execution time allowed (2 seconds).

Concretely, test code is:

```
TestCase skeletonTest = j3dTest.getTestCase (“load skeleton”);
skeletonTest .setUI (“center”, 0);
skeletonTest .testLoadBranch ("skeleton", BranchIdentifier.BRANCH_NAME);
skeletonTest.assertExeTimeEquals (2*1000);
```

Test result sends out a fail message shown as below:

```
[J3DPerfUnit] Running 14 tests in “ca.ucalgary.wepa.tests.gui.WEPAPerfTest”
[J3DPerfUnit] Fail (1 test)
[J3DPerfUnit] Failed Test Cases Are (Please check the report file for details):
[J3DPerfUnit] Load skeleton
```

This means the skeleton was not loaded in two seconds. The detailed report reads this:

```
Test case "load skeleton" fails:
  The BranchGroup was not attached to the scene graph in the required
  period
  Event – loading a BranchGroup
  BranchGroup Name -- skeleton
  BranchGroup User Data -- 0*17*
  Time Expecation -- 2000ms
```

To simply learn the execution time exhausted on each stage of the loading pipeline, developers can define a test case without specifying a time assertion. For example, to create a test case for the above skeleton loading test, one only needs to substitute “skeletonTest.assertExeTimeEquals (2*1000)” with “skeletonTest. assertNoTimeExpt()”.

Such tests are always treated as “Pass” by J3DPerfUnit. The following are the report details:

```
Test case "load skeleton" passes: (no time assertion is specified)
        General Information:
           Event -- loading a branch group
           BranchGroup Name -- skeleton
           BranchGroup User Data – 0*17*
           Time Expectation -- 5000ms
        Time spent on the loading pipeline (1st and 2nd stages):
           Total Execution Time-- 3406ms
           Time spent on the first stage -- 3360ms
           Time spent on the second stage -- 46ms
```

### *5.2.3 Unload a 3D Object*

Code and command line messages for 3D object unloading test are very similar to what is described in the previous section. Only the report details read differently. For instance, if the skeleton unloading test with a 1-second time assertion succeeds, the report reads this:

```
Test case "unload skeleton" passes:
        General Information:
           Event -- unloading a branch group
           BranchGroup Name -- skeleton
           BranchGroup User Data – 0*17*
           Time Expectation -- 1000ms
        Time spent on the unloading pipeline (1st and 2nd stages):
```

```
Total Execution Time -- 109ms
Time spent on the first stage -- 109ms.
Time spent on the second stage – 0ms
```

### 5.3 Evaluate Frame Rate

Interaction with a scene involves rotating, translating and zooming with only mouse, only keyboard, or a combination of mouse and keyboard. All of these interactions have been fully automated by J3DPerfUnit which uses Java Robot [J2SE1.5.0API07]..

Reporting "interaction time" was dropped because I cannot generate accurate reports for this metric using Java3D as the API does not provide access to this information. Therefore, J3DPerfUnit uses frame rate to evaluate interaction performance. A higher frame rate indicates a smoother interaction.

#### *5.3.1 Rotate a 3D Scene*

Rotating a scene allows for observing 3D objects from multiple angles. For instance, in a medical study viewing data of different parts of an organ might unveil critical information about a disease. The example below is a test for rotation performance on a human body with no organs loaded.

Six parameters must be defined for this rotation test:

- The title of the test window ("center");
- The test canvas index ("0").
- A start point for the mouse movement (in this example, the middle point of the test window);
- An end point for the mouse movement (in this example, the lower right corner of the test window);

- The number of steps the mouse movement will be divided into ( “5”);
- The time delay in milliseconds (“200”) before each step mentioned above. This parameter and the previous one are used together to better simulate the mouse movements produced by a human hand.
- The mouse button that is pressed while dragging (“InputEvent.BUTTON1_MASK” to denote the left button is pressed).
- The frame rate assertion (“30”).

**Figure 5-3** presents two screenshots of the scenes before and after the body is rotated by the test.

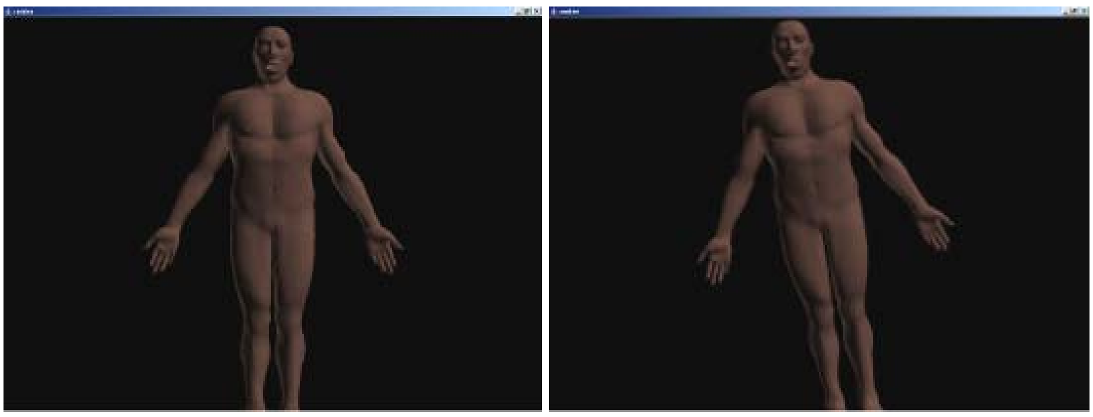


**Figure 5-3: Rotate an empty human body**

Concretely, the test code is:

```
/* start a frame rate counter */
j3dTest.setUI("center",0);
j3dTest.startFPSCounter();

/* rotate view */
TestCase bodyRotateTest = j3dTest.getTestCase (“rotate the empty body”);
```

```
bodyRoateTest.setUI(“center”, 0);
bodyRotateTest.testRotateView (
        new Point(windowXCenter, windowYCenter),
        new Point(windowXEnd , windowYEnd),
        5,
        200,
        InputEvent.BUTTON1_MASK);
bodyRotateTest.assertFrmRateEquals (30);
```

This test passes, which indicates a smooth rotation. The report details read the following:

```
Test case "rotate the empty body" passes:
        Event -- rotate a 3D scene;
        Frame Rate Assertion – 30 frames/second;
        Actual Frame Rate – 122 frames/second
        The rendering is smooth.
```

### *5.3.2 Translate a 3D Scene*

All the parameters for a translate test have the same meaning as those for a rotate test.

**Figure 5-4** presents two screenshots showing the scenes before and after running a translate test on an empty human body. The test succeeds, too.

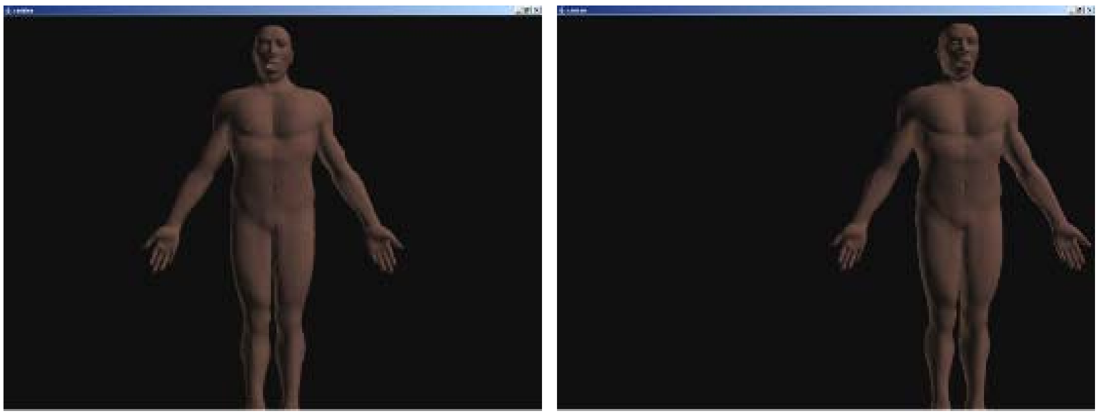


**Figure 5-4: Translate an empty human body**

### *5.3.3 Zoom a 3D Scene*

Zooming in can present more details of a 3D object for closer observation. Zooming out allows for an overview of the whole object. In medicine, an organ can be carefully examined by zooming in its texture.

Test examples for zooming in on a human body that has organs with complex geometric data are given below. These examples use two different ways to automatically implement the zoom: with mouse scroll and with a combination of mouse left click and Alt key press.

For tests using a combination of mouse and keyboard needs, an extra parameter “key press” in comparison to those for rotate and translate tests is needed.

For tests using mouse scroll, four parameters need to be specified:

- A predefined start point to position the mouse.
- Symbols representing the mouse wheel.

- The number of “notches” to move the mouse wheel (negative values indicate movement up/away from the user; positive values indicate movement down/towards the user).
- The frame rate assertion.

**Figure 5-5** shows two screenshots demonstrating the scenes displayed before and after the body is zoomed in using a combination of mouse left press and Alt key press.

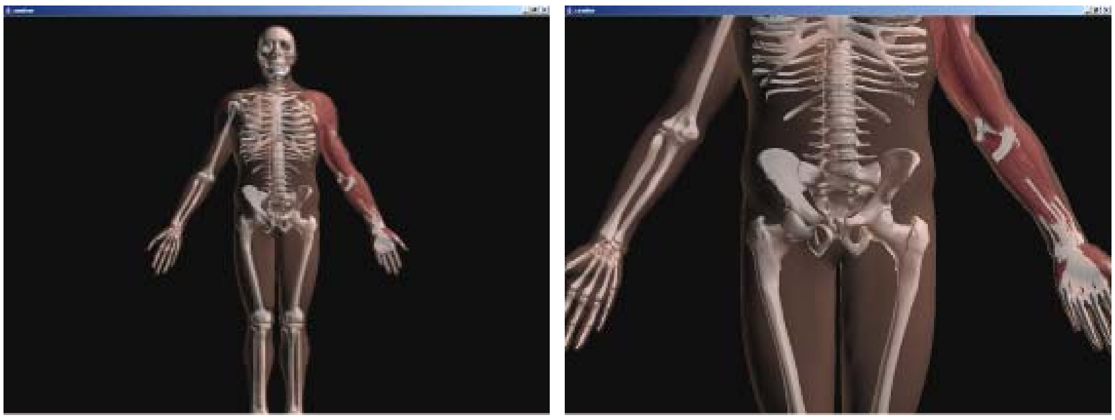


**Figure 5-5: Zoom in a human body with organs**

Both tests failed.

Messages for the second failed test are printed in the report like this:

```
Test case "rotate the empty body" fails:
        Event -- rotate a 3D scene;
        Frame Rate Assertion – 30 frames/second;
        Actual Frame Rate – 4 frames/second
        The rendering is not smooth.
```

The message highlights a dramatic drop of frame rate during zooming. Carrying organs with complex geometric data, the human body yields a very low interaction performance. As a result, developers either need to optimize the implementation to make the test pass or discuss a change of the performance requirements with their customers.

## 5.4 J3DPerfUnit Ant Task

Apache Ant is a Java-based build tool. It builds a Java application with XML-based configuration files which describe “a target tree where various tasks get executed” [Ant07]. Whenever changes are made to the source code, Ant tasks can be executed to check if an application has problems. These ant tasks can be compiling and running an application, wrapping up an application package, executing test cases, and so on.

To facilitate testing Java3D performance with Ant, J3DPerfUnit offers an Ant task to run performance tests. Defining ant tasks for J3DPerfUnit test is very similar to that for JUnit tests.

The following demonstrates how to include a performance test case (“ca.ucalgary.wepa.tests.gui.WEPAPerfTest”) into the build script of the WEPA project:

```
<j3dperfunit maxmemory = “512M”>
        <jvmarg value="-Dj3d.configURL=file:${configFile}"/>
        <classpath refid="unittest.class.path" />

        <!-- start batch tests, generate reports -->
        <batchtest todir="${j3dReports.dir}">
                <fileset dir="${src.dir}">
                        <include name="**/tests/gui/WEPAPerfTest.java" />
                </fileset>
        </batchtest>
</j3dperfunit>
```

The test output and detailed test reports are the same as shown in the previous sections.

## 5.5 Status of Implementation

J3DPerfUnit is implemented with Eclipse 3.1.0. Java Development Kit 1.5.0 is the underlying java compiler. In total, three java archive files, called jar files, are required to be on the build path. What follows is the explanation for these jar files: Jakarta-regexp-1.4.jar handles regular expressions; ant.jar provides the classes related to the J3DPerfUnit ant task; junit-4.1.jar is used for compiling and executing JUnit tests in the project.

J3DPerfUnit contains seven packages and 53 classes. All the functions are formally commented according to "Requirements for Writing Java API Specification" published on Sun website [Javadoc07]. Thus, the project API document can be easily generated by Javadoc Tool [JavadocTool07].

## 5.6 Chapter Summary

This chapter started with a general introduction on the tool features, including the tool's examination scope, the creation of test cases and the examination of test results and reports. Presenting source code and result of several test cases from the WEPA project, the chapter demonstrated how J3DPerfUnit is used to conduct Java3D performance testing. A customized ant task was exhibited to show J3DPerfUnit tests can also be conveniently executed in a continuous build process. The tool architecture was given, identifying the responsibilities of and the relationship among five key components. These components are used to setup and tear down a test environment, capture AWT events and scene graph events, conduct test cases, and produce test results and reports. This chapter concluded with a brief look at the status of tool implementation.

# Chapter Six: Tool Evaluation

J3DPerfUnit is the first proof-of-concept tool that automates Java3D performance testing. In order to assess the tool's usage and gather suggestions on improvements, a tool evaluation was performed with the WEPA development team. The evaluation has received the ethics approval from the Conjoint Faculties Research Ethics Board at the University of Calgary. The letter of approval can be found in Appendix B.

The evaluation involves an on-site evaluation session and a follow-up interview. In the beginning of this chapter, I introduce the objectives and the design of the evaluation. Then, I unveil the participants' Java3D programming background. Followed by this part is respondents' feedback pertaining to the tool usage. An analysis on the feedback is presented afterwards. Finally, I close this chapter by inspecting the limitations of this evaluation.

## 6.1 Objectives

The objectives are:

- To assess if J3DPerfUnit can conduct an effective Java3D performance testing;
- To investigate if J3DPerfUnit can enhance the efficiency of Java3D performance testing.

To evaluate the tool properly, it is essential to understand what effective and efficient testing is. Being effective means having the power to create an effect [Effective07]. Java3D performance testing, when determined as effective, should produce two effects:

- Detecting performance problems. For example, if a 3D object is not loaded in the required period, an effective testing is supposed to reveal this issue in a timely manner.
- Pinpointing the source of a performance problem. This could shorten the bug fixing time.

Efficiency can be represented by "the ratio of the effective or useful output to the total input" [Efficiency07]. Regarding testing, efficiency primarily concerns the use of resources that produce effective testing results. The less the resources are consumed, the more efficient testing is. In a testing process, resources can refer to people, equipment and time invested. In this thesis, testing efficiency particularly refers to the time invested. This is due to my research motivation: WEPA's manual testing on the Java3D component took considerable time and negatively affected the testing efficiency.

**6.2 Evaluation Design**

Originally, the evaluation was planned to last one month, from June 1, 2007 to June 30, 2007 during which J3DPerfUnit would be used by the WEPA team. At the end of the study, developers would provide their feedback on a questionnaire. If necessary, a follow-up interview would be conducted to further understand some responses. However, a severe bug in Java Development Kit (JDK) [BugRequestFocus07] disabled J3DPerfUnit from executing automated 3D interaction tests on the Solaris platform [Solaris07] where WEPA was developed. As a result, the initial plan became infeasible, and the following strategies were then delivered as a replacement.

*Evaluation Steps*

Step 1: Preparation. I read to the seven WEPA developers the recruiting statement submitted with the ethic approval. Next, I handed out a questionnaire and two pieces of blank paper to each participant. The blank papers were for participants to note their initial comments during the tool demonstration.

Step 2: General introductions. First, I introduced the questionnaire structure. This helped the respondents to be familiar with: (1) how the questions were categorized; and (2) what each question was about. Then, I presented the goal of this study. Lastly, I provided an overview of my research background.

Step 3: Tool demonstration. To begin, I presented how to install J3DPerfUnit into a Java3D project. Then, I exhibited the tool through running **three tests designed for WEPA**: (1) execution time to build the scene graph; (2) execution time to load a heart into the human body; and (3) frame rate produced when rotating and translating the human body. For each test case, I explained the source code, the result shown on the command line, and the generated test report. Following each test case demo was a five to ten minute question session. Respondents raised questions, and put down their comments on the blank papers passed out in Step 1. Finally, I demonstrated how to use J3DPerfUnit Ant task to run a Java class with **14 performance tests written for WEPA**.

Step 4: Collecting feedback. First, an open discussion was conducted on the overall impression about J3DPerfUnit. Next, each respondent completed the questionnaire, and indicated if he/she was willing to be interviewed for a further understanding of his/her feedback. In the end, I collected all the questionnaires.

Step 5: Follow-up interviews. When some responses needed to be clarified further, follow-up interviews were performed with the particular respondents.

*Data Collection*

Data was gathered from three places: a questionnaire of 12 questions, notes for the on-site open discussion, and the record for the follow-up interviews (conducted over email).

In the questionnaire, both quantitative and qualitative questions are included. The responses to quantitative questions are measured by a 5 point Likert Scale [LikertScale07] (5 ”strongly agree” to 1 “strongly disagree”). The qualitative questions are multiple-choice and open-ended questions. The use of Liker Scale is to investigate whether participants favor a statement or not, while multiple-choice and open ended questions assess respondents’ perceptions of J3DPerfUnit and also gather their suggestions on the tools’ future work. At the end of the questionnaire, all the participants can make a choice of whether they would like to have a follow-up interview or not.

## 6.3 Participants

Seven WEPA developers planned to participate in the evaluation on June 22, 2007. However, two developers could not make it. For them, I arranged another evaluation session on July 13, 2007. Both sessions followed the evaluation steps listed in Section 6.2.

One participant did not complete the questionnaire because the participant had no experience with Java3D performance testing. Therefore, only the other six responses will be presented as the evaluation results and used for an analysis. Note that each questionnaire was filled in from each respondent’s personal view, and therefore is subjective in nature. Meanwhile, because I interpreted the responses alone, the evaluation analysis performed in this chapter is also discretionary.

Among the six respondents, five are summer students. Table 6-1 outlines the participants' experience with Java3D in months.

**Table 6-1: Months of Experience with Java3D**

| Developers | Java3D Experience |
|---|---|
| Developer 1 | 1 |
| Developer 2 | 1.5 |
| Developer 3 | 24 |
| Developer 4 | 2 |
| Developer 5 | 1 |
| Developer 6 | 2 |

In regard to Java3D programming, one had a 24-month programming experience with Java3D. Two had a two-month experience working with Java3D technology. One spent one and a half months in developing WEPA with Java3D. The remaining two respondents programmed with Java3D for one month.

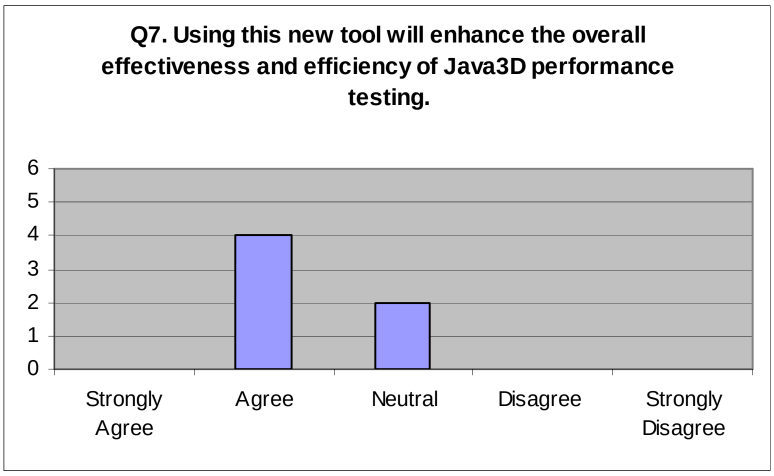
## 6.4 Results

All respondents (100%) accepted a follow-up interview. For each interview, a question list was created in order to guide the overall process.

### *6.4.1 J3DPerfUnit in general*

To solicit the participants' overall impression on J3DPerfUnit, I used two methods. One is: I designed Question 7, 11, and 12. The other is: after analyzing the result for all twelve questions asked in the questionnaire, I emailed every respondent an analysis summary for a confirmation. Developers' answers to the three questions and to the analysis summary are presented below.

Firstly, **Figure 6-1** illustrates the responses for Question 7 and 11. For Question 7, four respondents believe that using J3DPerfUnit enhances the overall effectiveness and efficiency of Java3D performance testing. The other two respondents chose “Neutral” as the answer. For Question 11, four out of six respondents suggested that they either agree or strongly agree to use J3DPerfUnit in their future Java3D performance testing. The remaining two respondents were on neutral ground.



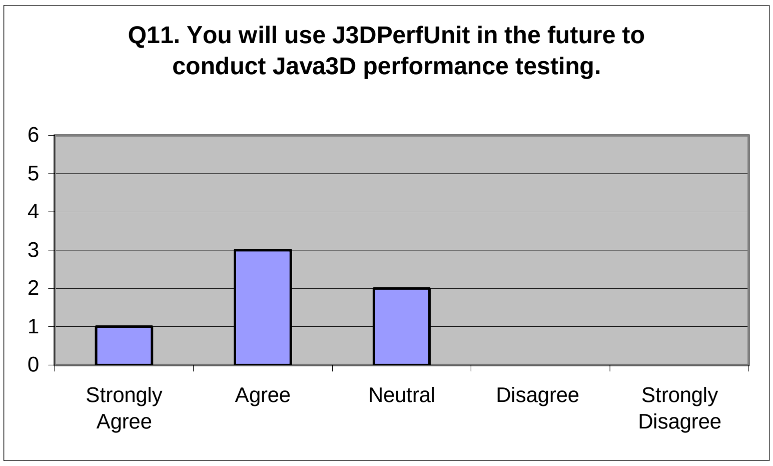


**Figure 6-1: J3DPerfUnit general perceptions distribution by the WEPA team**

In the follow-up interviews, the developers who answered “Neutral” to Question 11 gave these explanations. One said “The problem that I am currently facing is that I wont be able to  use J3DPerf as long as it is subjected to Windows”. The other expected the test report to be much more specific so that the problem source could be spotted faster and easier.

Secondly, regarding Question 12, each respondent submitted at least one of the comments below: (1) J3DPerfUnit would increase test-coverage; (2) the tool would increase their overall development efficiency; and (3) the tool can point out inefficiencies in the Java3D Engine. In the open field for this question, one respondent also commented that “automating manual testing is a great improvement”.

Thirdly, all participants confirmed the analysis summary I made. The summary reads below:

- **Research Idea: Automate Java3D performance testing with tool support.**

  The idea itself is good and will benefit Java3D software development. It can save considerable time and effort on testing.

- **Current Tool Development**

  As a tool support for the research concept, J3DPerfUnit has

  *Advantages:*

  (a) Can detect performance problems;

  (b) Regarding problem detection only, the tool is more efficient than manual testing, especially when a test needs to be executed repeatedly. This is because J3DPerfUnit can reduce testing time and effort.

  *Disadvantage:*

  The tool cannot offer more specific and detailed information on where a problem comes from. For example, time spent on smaller block of the application code, and frame rate variation during an interaction.

### *6.4.2 Test case design*

"In software engineering, the most common definition of a test case is a set of conditions or variables under which a tester will determine if a requirement or use case upon an application is partially or fully satisfied" [TestCase07].

In this evaluation, the respondents were asked to use the following standards when deciding if J3DPerfUnit defines test cases appropriately or not:

- Does J3DPerfUnit define scope of the test cases based on the pre-development survey result from Chapter Three? According to the survey, these test cases are commonly used for Java3D performance testing.
- Does J3DPerfUnit offer a set of parameters that accommodate various execution conditions? For example, the variables defined for automating user interactions should allow for the triggers from mouse, keyboard, or a combination of both.

**Figure 6-2** shows that an overwhelming majority of the six developers consider J3DPerfUnit test cases to be defined properly. One respondent commented that the definitions are "simple and logical".

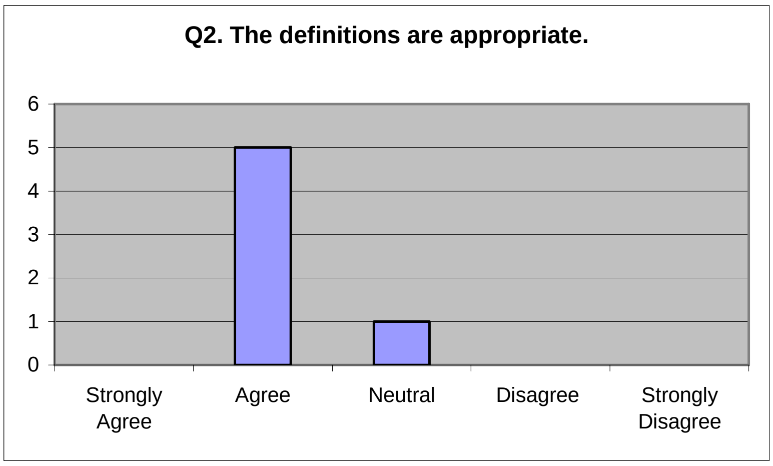


**Figure 6-2: Test case definition perceptions distribution by the WEPA team**

The one participant who answered "Neutral" explained as follows:

*"I think that the definitions of the test-cases as such are clearly-structured. I am not so much into the various types of unit tests (JUnit, Jemmy) that I could direcly compare the methods of defining test-cases used in J3DPerf to those."* – Developer

### *6.4.3 Test report accuracy*

A test report is accurate when it correctly shows the result against the assertion set in a test case. **Figure 6-3** illustrates that four developers thought the feedback in the test report is accurate. The other two replied "Neutral". Among these two respondents, one claimed that besides showing the average frame rate, an accurate report should also reflect the change of frame rate during user interactions because "Frame rate could vary wildly during a test execution" (the developer). The other thought the report did not effectively point out the performance problem source from the application code.

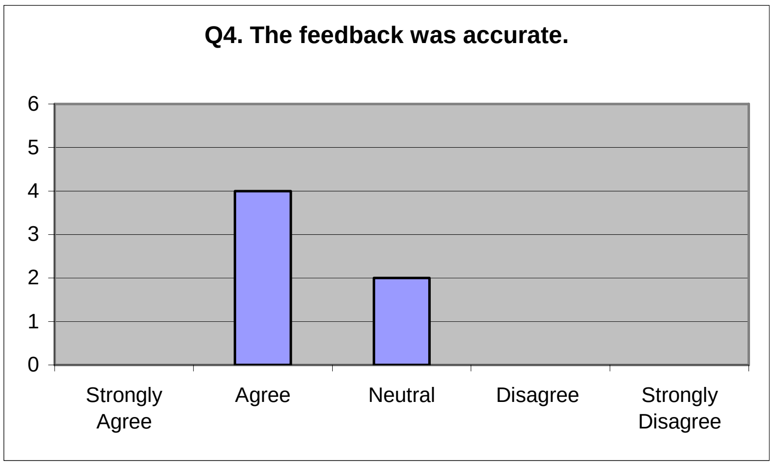


**Figure 6-3: Test report accuracy perceptions distribution by the WEPA team**

### *6.4.4 Ability to locate performance problems*

As demonstrated in **Figure 6-4**, two respondents agreed that "the feedback in the test report helped locate performance problems in the application". One respondent held a neutral attitude while the remaining three disagreed.

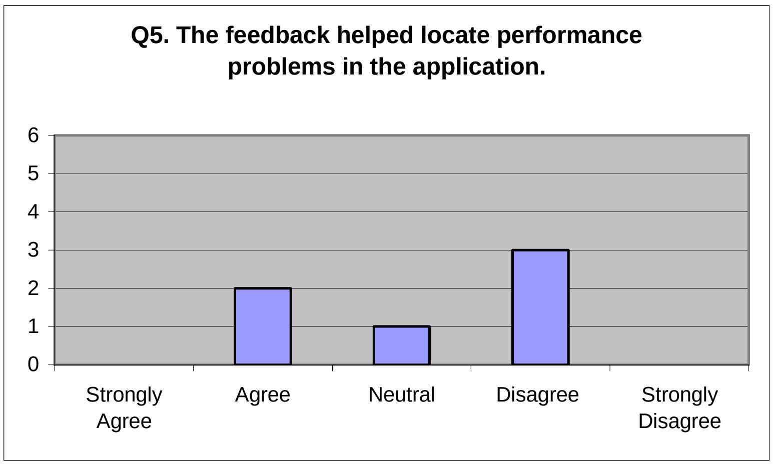


**Figure 6-4: Test result perceptions distribution by the WEPA team**

In the follow-up interviews, the four developers who answered either "Neutral" or "Disagree" detailed their responses. Although the current test report can point out whether a problem originates from the source code or Java3D engine, this information is still too general. One developer quoted "I disagree because of the lack of specificity in locating the problems, not merely detecting them in the program". All the developers anticipated the tool can report more detailed performance information, such as time spent on creating a BranchGroup, during application runtime.

### *6.4.5 Ease of application*

As a unit testing tool, J3DPerfUnit tries to make itself to be easily employed through three efforts. First, the tool installation only requires copying jar files into a project's library folder. Second, the process to write a test case is very simple. Basically, the creation of each test case comprises only three to four code statements. Lastly, test report is generated with a description that is clear to read.

After demonstrating the above three features, a staggering majority of participants (83.3%) agreed or strongly agreed that J3DPerfUnit is easy to apply (**Figure 6-5**).

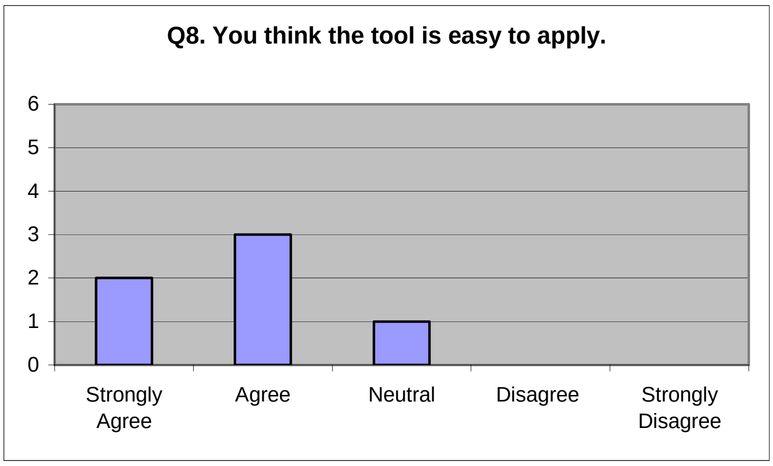


**Figure 6-5: "Easy to apply" perceptions distribution by the WEPA team**

The only developer who answered “Neutral” elaborated “The tool can be hard to apply if the scene already contains a Behaviour node… Also…If I try to use the mouse while the test suite is using the mouse to move stuff around, will I be interfering with it? If yes, then it could be a problem”. He drew this conclusion due to two facts. First, to calculate frame rate, J3DPerfUnit needs to start a counter which uses a particular API “WakeupOnElapsedFrames”. If the tested Java3D application requires an animation that calls the same API, a conflict will rise. This is because “WakeupOnElapsedFrames” cannot be called by two places at the same time. Second, to automate user interactions, J3DPerfUnit uses the Java Robot package [JavaRobot07]. To properly automate mouse or keyboard actions with the package, one must not physically move the mouse or press the keyboard when the automation is running.

### *6.4.6 A comparison*

All six developers were doing manual performance testing. As shown in **Figure 6-6**, five developers claimed that currently they had no preference for testing done manually versus the one performed with J3DPerfUnit. One developer favoured manual testing. All developers agreed that J3DPerfUnit makes detecting performance problems more efficient than manual testing, but as indicated in 5.4.1 and 5.4.4, they expected more detailed information in the test report to facilitate their problem location.

Among the developers who replied “Neutral”, one explained “J3DPerfUnit does not report very specific info, so at this point, there is no preference. However, I would certainly prefer J3DPerfUnit once it can output more specific info pertaining to some of the problem.” Another responded “They would complement each other. I.e. I would still use some manual testing of performance, but J3DPerfUnit is a welcome addition”. The others expressed the similar opinions.

The developer who preferred manual testing explained: “Manual testing is faster for on-the-spot testing. If I spent a lot of time on performance analysis, J3DPerfUnit would be more useful to check how long an action takes.”

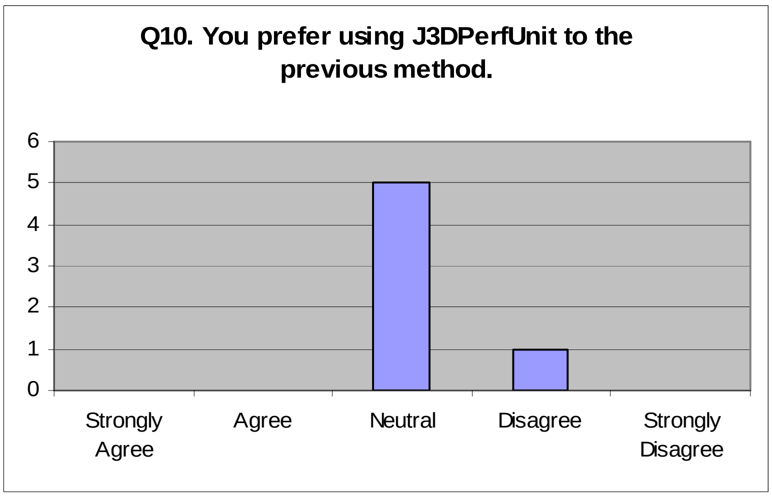


**Figure 6-6: Comparison perceptions distribution by the WEPA team**

## 6.5 Analysis

The tool evaluation pointed out the strengths of and improvements for J3DPerfUnit, the first proof-of-concept tool that automates Java3D performance testing

### *6.5.1 Strengths*

*Testing effectiveness*

J3DPerfUnit effectively detects performance problems by implementing the followings:

- Properly specified test cases. Firstly, test cases are the ones commonly used in 3D performance testing to catch problems. Thus, J3DPerfUnit can be employed to examine most user performance requirements. Secondly, the parameters used in test cases define the various test conditions. This allows performing a test case with different scenarios and criteria.
- Basically, the test report is correct. It clearly shows if a test passes or fails against the assertion made.

*Testing efficiency*

- Compared to manual testing, using J3DPerfUnit is more efficient in performance problem detection. This is because J3DPerfUnit can reduce testing time, especially when a test needs to be continuously executed.

***6.5.2 Improvements***

Listed below are the highlighted improvements for J3DPerfUnit:

- Lack of specific and detailed information on the source of a performance problem. For instance, time spent on smaller modules in the application code and frame rate variations during an interaction are currently unavailable in J3DPerfUnit. These limitations originate from two facts. First, the researcher has only half a year programming experience with Java3D. Without enough time to investigate deeper technical details, the researcher could not provide more specific performance information on smaller code blocks. Second, in the pre-development survey, participants only mentioned an average frame rate as a metric for user interaction tests. Therefore J3DPerfUnit solely implemented this requirement in its current version.
- User interaction tests cannot be executed on Solaris. To automate mouse/keyboard actions triggering user interactions tests, J3DPerfUnit used the Java Robot package in JDK library. Unfortunately, one indispensable API "Component.requestFocus()" in the package does not work on Solaris. Being in the Sun database for four years, this bug still remains unresolved. The researcher did not know about the problem, and only used a Windows machine for the tool development.
- J3DPerfUnit cannot work with some animations. Java3D API "WakeupOnElapsedFrames" is used by J3DPerfUnit to implement the frame rate counter. A conflict will arise if the tested Java3D application has an animation triggered by the same API call. The improvement on this issue might be difficult.

Based on the researcher’s current knowledge, to produce a frame rate counter for general Java3D applications, “WakeupOnElapsedFrames” must be called.

## 6.6 Limitations

This evaluation assessed the value of a proof-of-concept tool J3DPerfUnit on the effectiveness and efficiency of Java3D performance testing. Several limitations may affect validity of the evaluation result:

- Solely based on a tool demonstration, this short-term evaluation could not offer developers a hands-on trial of the tool. Accordingly, the participants might not have a deep understanding of the tool’s implementation, such as how to use J3DPerfUnit to create effective test cases, etc. The good thing is that, during its development, J3DPerfUnit used WEPA as one of the test applications. Thus, the WEPA test cases prepared for the case study should look similar if they were written by WEPA developers themselves. Yet it is still possible that other test scenarios the researcher did not think of may be created.
- The sample size was small. Lacking the time and resources, the researcher could only conduct the tool evaluation with a development team in a co-operator’s lab. The six participants might not be representative of the target population. Thus, the interpreted results could suffer from this limitation and my conclusion may not be strong.
- Five out of six participants used Java3D for only one to two months. They did not have substantial experience with Java3D performance testing. As a result, the value of their responses needs to be taken with a grain of salt. Since they covered most of the survey population, the overall survey quality was influenced.

- No information was collected regarding how often WEPA performance testing was done. If the testing was regularly executed, the test automation process might have been more recognizable to the participants. Thus, more valuable feedback might have been submitted.
- Some respondents knew the researcher. This could bias their responses.

## 6.7 Chapter Summary

An evaluation is performed to assess the value of J3DPerfUnit on Java3D performance testing. Altogether, 17 tests written for WEPA were demonstrated during the evaluation. The evaluation disclosed that it is efficient and effective to use J3DPerfUnit to detect performance problems when performance testing needs to be conducted continuously. Meanwhile, to make performance testing more effective, the future development of J3DPerfUnit should focus on making locating problem source easier by detailing much more performance information in reports.

# Chapter Seven: Future Work

## 7.1 Tool Enhancement

J3DPerfUnit is a proof-of-concept of automating Java3D performance testing. For a use in industry, the tool needs to make several enhancements:

*Locate Problem Source*

The current version of J3DPerfUnit is not good at pinpointing a performance problem source. This can be improved by generating more specific information for a test. According to the evaluation feedback, the specific information could encompass:

- Time spent on smaller application code blocks. For example, how long it takes to read the geometry data, and to generate a BranchGroup respectively. Or as a respondent suggested, profiling the source code can be considered as well;
- Frame rate variations during an interaction. The tool should produce a list of frame rates produced every 2 seconds during an interaction test.

*Replace Java Robot*

Because the Java Robot package has a bug, user interaction tests in J3DPerfUnit cannot not work on Solaris. The future development may ponder using the Java3D animation package (com.sun.j3d.utils.behaviors) [Java3DAPISpecification07] to produce the effect of zooming, translating and rotating a 3D scene.

*Revise Frame Counter*

Based on the researcher’s current knowledge, to produce a frame rate counter, the API call “WakeupOnElapsedFrames” must be used. As claimed in Chapter Five, a conflict will rise when the applications under testing use the same API.

Future tool development needs to work out a way to deal with this issue. For now, the researcher cannot give any suggestions.

*Include One More Metric*

Besides the two implemented metrics: execution time and frame rate, memory usage is another widely cited measurement in the pre-development survey. High memory usage usually signals a performance bottleneck. When it is detected, the developers are urged to improve the way they have dealt with the system resources.

*Facilitate System Load Testing*

System load testing is a vital part of performance testing that indicates a system's highest capacity. This kind of testing is often used for tuning web application performance [Nguyen00, Subraya06]. Likewise, the ability to process a large number of objects is crucial in 3D applications. Games, for example, require smooth translation, rotation or zoom over all the objects on a scene.

*Record Test History*

Recording a history for test results can be beneficial to identify possible culprits of a performance breakdown. By comparing data from different periods, the history provides hints as to what changes may affect performance.

## 7.2 Further Background Survey

The background survey experiences several limitations. A further study addressing these limitations is required. Possible improvements in the further study include:

- Increase the sample size. The background survey I conducted had only nine participants. Such a small sample size might affect the validity of survey responses. Also, a valid statistic analysis could not be done on the test metric choices due to the small sample size. How to calculate the sample size that fits a reasonable study depends on several factors. One factor is characteristics of the target population, such as what kind of applications the selected 3D developers implement, how long have these developers been working on 3D applications, and how often the developers conduct performance testing. Another factor could be what kind of analysis would be conducted on the future survey. If the analysis plans to compare subgroups, for example, metric selection difference within three categories of programming experiences, then enough people is required to fill in the three categories [Nardi06].

- Find industry respondents from various areas. In the conducted background survey, all the participants came from the academic area. The future study should consider including the 3D developers with industry background. This is because the industry developers would have more chances to work with the real customers who would ask for 3D performance requirements that can be used in performance testing. Also, it would be better to have the developers who work on different kinds of 3D applications. The reason is these developers would see various performance requirements. The more diversified requirements are collected, the more kinds of test cases and test metrics we can include into J3DPerfUnit.
- Invite the experienced 3D developers.  Here, “experienced” means two things. First of all, the future background survey should collect feedback from the 3D developers who have a long programming experience in 3D applications, say more than two years. Experienced developers will know more performance issues and do performance testing more often than beginners. Second, the selected developers should run their performance tests on a regular base. This kind of developers will

have better suggestions on features of tool support for automated performance testing. Conducting regular performance tests will make it clearer what is the one who does not run performance testing frequently.

## 7.3 Further Tool Evaluation

The tool evaluation also suffers various limitations. The solutions to these limitations are the same as what is discussed in Section 7.2. Meanwhile, I want to add one more suggestion: give the participants a longer attempt time (for instance, two months) on J3DPerfUnit. This way, the participants can have a deeper understanding on how the tool works. Thus, the responses from these participants would be more complete and valid.

# Chapter Eight: Conclusion

Performance plays a vital role for high quality Java3D applications. Without an acceptable performance, Java3D software tends to become useless. To evaluate performance, corresponding testing should be done continuously. Repeated and continuous performance testing, in the context of this work, means: whenever source code is changed (a new feature is developed, a bug is fixed, or code refactoring is done), developers should either write new test cases or execute old test cases to see if the change introduces any performance problems. In this manner, performance problems can be exposed and cleared up in time, and thus will not accumulate to an extent where they are tough to resolve.

In spite of the importance of regular performance testing, existing strategies and tools fall short of the desired support. They take considerable time and effort and yet are still not effective. The research goals set up in this thesis thus are:

- To thoroughly explore Java3D performance issues, confirming the importance of regular and continuous testing for Java3D performance;
- To investigate current tool support and approaches for Java3D performance testing in detail, demonstrating their restrictions in effectively testing Java3D performance regularly and continuously;
- To present requirements for the new tool development;
- To present a proof of concept of tool support for automated Java3D performance testing;
- To evaluate the tool's impact on the effectiveness and efficiency of Java3D performance testing.

The first and second goals are fulfilled in Chapter Two. In this chapter, I extensively discussed the major causes of 3D performance problems. I also detailed the limitations in current strategies and tools for Java3D performance testing.

The third goal is achieved through a survey on 3D performance testing. The participants' responses gave me a guideline that resulted in a set of basic requirements for the new tool development.

To reach the forth goal, I developed J3DPerfUnit which automates Java3D performance testing. J3DPerfUnit looks into four respects of Java3D application performance. First, the tool investigates how long a 3D scene can be displayed after the application starts. Next, the tool calculates the execution time to load 3D objects onto a scene graph. Then, the tool calculates the execution time to unload 3D objects from a scene graph. Finally, the tool reports the frame rate produced during user interactions. Before this work, such a tool support was unavailable for Java3D performance testing.

The fifth goal is implemented through the tool evaluation discussed in Chapter Six. The evaluation was conducted with the WEPA developers. During the evaluation, J3DPerfUnit was demonstrated with real test code for the WEPA application. While the tool needs to improve effectiveness in reporting the problem source, support of the tool and the concept of test automation were confirmed as being effective and efficient in detecting Java3D performance problems.

## References

[Ant07] Apache Ant; http://ant.apache.org/ (Last visited May 07, 2007)

[Bargen98] Bradley Bargen & Terence Peter Donnelly; Inside Directx (Microsoft Programming Series); Microsoft Pr, Redmond; Washington; 1998

[Binder00] R. V. Binder; Testing Object-Oriented Systems: Models, Patterns and Tools; Addison Wesley; 2000

[BugRequestFocus07] Bug: Component.requestFocus() fails on Solaris and Linux; http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4851685; (Last visited June 27, 2007)

[Burnstein03] Ilene Burnstein; Practical Software Testing; Springer, 1st edition; June 24, 2003

[Buzen76] J. P. Buzen; Tuning: Tools and Techniques; ACM SIGMETRICS Performance Evaluation Review; Volume 5, Issue 4; Fall 1976; p63-81

[Can et al.03] T. Can, Y. Wang, Y.-F. Wang and J. Su; FPV: Fast Protein Visualization Using Java 3D; Proceedings of the 18th Annual ACM Symposium on Applied Computing, SAC'03; Melbourne, FL; March, 2003; pp. 88-95

[Chen et al.06] Jim X. Chen & Edward J. Wegman; Foundations of 3D Graphics Programming: Using JOGL and Java3D; Springer; London; 2006

[Chen02] Jim X. Chen; Guide to Graphics Software Tools; Springer; October 2002

[Chung et al.00] Chung, L., B. A., E. Yu, and J. Mylopoulos; Non-Functional Requirements in Software Engineering; Kluwer Academic Publishers; Dordrecht, The Netherlands; 2000

[Clements96] P.C. Clements; Coming Attractions in Software Architecture; Technical Report No.CMU/SEI-96-TR-008; Software Engineering Institute, Carnegie Mellon University; Pittsburgh, PA; 1996

[Clingman et al.04] Dustin Clingman, Shawn Kendall and Syrus Mesdaghi; Practical Java Game Programming; Charles River Media; June 2004;

[CodeCoverage07] Code Coverage; http://en.wikipedia.org/wiki/Code_coverage; (Last visited April 20, 2007)

[Cohn04] Mike Cohn; User Stories Applied: For Agile Software Development; Addison-Wesley Professional; 2004

[Day07] Bill Day; 3D graphics programming in Java, Part 1: Java3D; http://www.javaworld.com/javaworld/jw-12-1998/jw-12-media.html; 1998; (Last visited April 09, 2007)

[Day07] Bill Day; 3D graphics programming in Java, Part 3: OpenGL; http://www.javaworld.com/jw-05-1999/jw-05-media.html;1999; (Last visited April 13, 2007)

[EBE07] EBE Main Page; http://ebe.cpsc.ucalgary.ca/ebe/; (Last visited April 29, 2007)

[Eclipse07] Eclipse Homepage; http://www.eclipse.org; (Last visited May 07, 2007)

[EET07]Empowering Equipment Testers; http://www.esolpartners.com/shared/pdf/Automated_Testing_Breakthrough_11.1.06.pdf; (Last visited Aug 14, 2007)

[Effective07] Effective; http://en.wiktionary.org/wiki/effective; (Last visited June 26, 2007)

[Efficiency07] Efficiency; http://www.answers.com/efficiency?cat=biz-fin; (Last visited June 26, 2007)

[Elliott et al.02] James Elliott, Robert Eckstein, Marc Loy, David Wood and Brian Cole; Java Swing, Second Edition; O'Reilly Media, Inc.; 2002

[FrameRate07] Frame Rate; http://en.wikipedia.org/wiki/Frame_rate; (Last visited April 15, 2007)

[Fraps07] Fraps; http://www.fraps.com/; (Last visited April 17, 2007)

[Gao03] Jerry Zeyu Gao, H.-S. Jacob Tsao and Ye Wu; Testing and Quality Assurance for Component-Based Software; Artech House, Incorporated; 2003

[HardwareAcceleration07] Hardware Acceleration; http://www.answers.com/hardware%20acceleration; (Last visited April 11, 2007)

[IEEE02] IEEE Standard Glossary of Software Engineering Terminology; Standard 610.12-1990 (R2002), IEEE Computer Society Press; 2002

[IterativeAndIncrementalDevelopment07] Iterative and incremental development; http://en.wikipedia.org/wiki/Iterative_and_incremental_development (Last visited April 06, 2007)

[Java3D07] Java 3D; http://en.wikipedia.org/wiki/Java3d; (Last visited April 07, 2007)

[Java3DAPISpecification07] Java3D API Specification; http://download.java.net/media/java3d/javadoc/1.5.0/index.html; (Last visited April 09, 2007)

[Java3DAPITutorial07] Java3D API Tutorial; http://java.sun.com/developer/onlineTraining/java3d/; (Last visited April 09, 2007)

[Javadoc07] Requirements for Writing Java API Specifications; http://java.sun.com/j2se/javadoc/writingapispecs/index.html; (Last visited May 17, 2007)

[JavadocTool07] Javadoc Tool; http://java.sun.com/j2se/javadoc/; (Last visited May 17, 2007)

[JavaRobot07] Java Robot: J2SE 5.0 API Specification; http:// java.sun.com/j2se/1.5.0/docs/api/java/awt/Robot.html; (Last visited July 13, 2007)

[J2SE1.5.0API07] J2SE 1.5.0 API Specification; http://java.sun.com/j2se/1.5.0/docs/api/; (Last visited May 17, 2007)

[Jemmy07] Jemmy: Overview; http://jemmy.netbeans.org/; (Last visited April 04, 2007)

[JUnit07] JUnit; http://www.junit.org/index.htm; (Last visited April 16, 2007)

[JUnit4In10Minutes07] JUnit 4.0 In 10 Minutes; http://www.instrumentalservices.com/index.php?option=com_content&task=view&id=45&Itemid=52; (Last visited April 16, 2007)

[JUnitPerf07] JUnitPerf; http://clarkware.com/software/JUnitPerf.html; (Last visited April 16, 2007)

[Kaner et al.04] Cem Kaner & Walter P. Bond; Software Engineering Metrics: What Do They Measure and How Do We Know? 10th International Software Metrics Symposium, Metrics 2004

[Knudsen99] Jonathan Knudsen; Java 2D Graphics [BARGAIN PRICE]; O'Reilly & Associations, Inc.; Sebastopol; 1999

[Lewis04] William E. Lewis; Software Testing and Continuous Quality Improvement, Second Edition; AUERBACH; 2004

[LikertScale07] Likert Scale; http://en.wikipedia.org/wiki/Likert_scale; (Last visited Aug 16, 2007)

[MASE07] MASE Project; http://mase.svn.sourceforge.net/viewvc/mase/ (Last visited May 07, 2007)

[Meloan01] S. Meloan; Exploring the New Frontier: Java$^{TM}$ Technology Powers the "Post-Genomic: Era"; http://java.sun.com/features/2001/10/genome2.html (Last visited April 03, 2007)

[Myers et al.04] Glenford J. Myers, Corey Sandler, Tom Badgett and Todd M. Thomas; The Art of Software Testing, second edition; Wiley; June 21, 2004

[Nardi06] Peter M. Nardi; Doing Survey Research, second edition; Allyn & Bacon;2006.

[Nguyen00] Hung Q. Nguyen; Testing Applications on the Web: Test Planning for Internet-Based System; Wiley; 2000.

[NTSC07] NTSC; http://en.wikipedia.org/wiki/NTSC (Last visited June 11, 2007)

[OpenglArchitectureReviewBoard04] Opengl Architecture Review Board & Dave Shreiner; OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.4 (4th Edition); Addison-Wesley Professional; 2004

[Pettichord01] Bret Pettichord; Seven Steps to Test Automation Success; http://www.io.com/~wazmo/papers/seven_steps.html; 2001 (Last visited April 19, 2007)

[Schwaber et al.01] Ken Schwaber & Mike Beedle; Agile Software Development with SCRUM; Prentice Hall; 1st edition; 2001

[Smith et al.02] Smith C.U. & L. G. Williams; Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software; Addison-Wesley; Boston, MA; 2002

[Smith et al.01] Shaun Smith & Gerard Meszaros; Increasing the Effectiveness of Automated Testing; http://www.agilealliance.org/system/article/file/994/file.pdf; 2001; (Last visited April 19, 2007)

[Solaris07] Solaris Operating System, http://en.wikipedia.org/wiki/Solaris_(operating_system) (Last visited July 9, 2007)

[Sommerville04] I. Sommerville; Software Engineering 7th Edition; Addison-Wesley; Boston, MA; 2004

[Sourceforge07] Sourceforge Home Page; http://sourceforge.net/ (Last visited Aug 16, 2007)

[Sowizral et al.99] Henry A. Sowizral & Michael F. Deering; the Java 3D API and Virtual Reality; IEEE Computer Graphics and Applications, Volume 19, Issue 3; IEEE Computer Society Press; 1999; p12-15

[Subclipse07] Subclipse: SVN Plug-in for Eclipse; http://subclipse.tigris.org/ (Last visited May 07, 2007)

[Subraya06] B. M. Subraya; Integrated Approach to Web Performance Testing: A Practitioner's Guide; IRM Press; Hershey, PA, USA and London, UK; 2006

[Tamres02] Louise Tamres; Introducing Software Testing; Addison-Wesley Professional; May 9, 2002

[TestAutomation07] Test Automation; http://www.answers.com/test%20automation (Last visited April 02, 2007)

[TestCase07] Test Case; http://en.wikipedia.org/wiki/Test_case (Last visited June 28, 2007)

[Watkins01] John Watkins; Testing IT: An Off-the-Shelf Software Testing Process; Cambridge University Press; 2001

[WEPA07] WEPA Human Body Project; http://cave.ucalgary.ca/wepa/ (Last visited April 04, 2007)

# APPENDIX A: BACKGROUND SURVEY

## A.1. Survey Introduction

Hello my name is Xueling Shu, a graduate student at the University Of Calgary, Department Of Computer Science. I am doing a thesis based Masters Degree. Dr. Frank Maurer is my supervisor.

Our research aims to bring forth more effective ways to conduct performance testing for Java 3D application. We plan to develop a testing tool to perform automatic performance testing for Java 3D applications and will evaluate the tool's effectiveness and efficiency in the end.

Currently, we are in the first stage of the research: investigating performance issues with and performance testing techniques for Java3D applications. By analyzing the results from questionnaires, interviews, and research notes, we expect to collect the requirement for the tool development.

If you volunteer as a participant, you will fill in an initial study questionnaire to collect the current performance issues with and performance testing techniques for Java3D applications. You will be invited to open ended interviews to let us understand more state of art of Java3D performance testing. We won't expose your name in our study. Some interview sessions will be tape recorded with your permission to facilitate collection of information, but being taped recorded is not a required condition to take part in our research.

You may withdraw whenever you want without any negative consequences, and choose whether to have data collected to the point of withdrawal excluded from the study or retained/used by the researchers. At the same time, the decision to participate (or not) will in no way affect students' grades in any course, and that the instructor will not know who has or has not chosen to participate.

The study has received formal ethics clearance thorough Conjoint Faculties Research Ethics Board. Participation in the research is voluntary and there are no known or anticipated risks involved.

If you are willing to participate, please fill out the following consent form and we will be in touch with you.

## A.2. Questionnaire

### Background

1. Programming Experience with 3D applications
   Number of years: _______

2. Programming experience with OpenGL
   Number of years: _______

3. Programming experience with Java 3D
   Number of years: _______

4. For what fields are your ***Java 3D*** applications designed (i.e. bioinformatics, games etc)?

### Requirement

5. How the performance requirements for your 3D applications are collected (any specific method or framework)?

6. Please delineate one or more performance requirement in your 3D applications. To protect confidential information, actual names or titles can be replaced. (For example: xx object should be shown in xx seconds)

### Testing

7. On which levels is ***performance*** tested for the 3D applications you build? (Multiple Choices) ______________
   (a) Automated unit testing
   (b) Manual unit testing (i.e. debugging)
   (c) Automated user acceptance testing
   (d) Manual user acceptance testing
   (e) Other ways, please indicate what they are ______________________
   (f) No testing

8. If unit testing (***for 3D applications performance***) is automated, what are the testing tools you use? Please list their advantages and disadvantages?

9. Please present a test case example of unit testing.

10. If user acceptance testing (***for 3D applications performance***) is automated, what are the testing tools you use? Please list their advantages and disadvantages?

11. Please present a test case example of user acceptance testing.

**Problem Description**

12. Please list and describe in details the performance problems in OpenGL and Java 3D applications you build.

**Metrics**

13. What metrics you would define to describe the performance requirement for a 3D application? ("Execution time", for example. Any others?)

**Desirable Features**

14. What are the desirable features of an automatic performance testing tool for 3D applications?

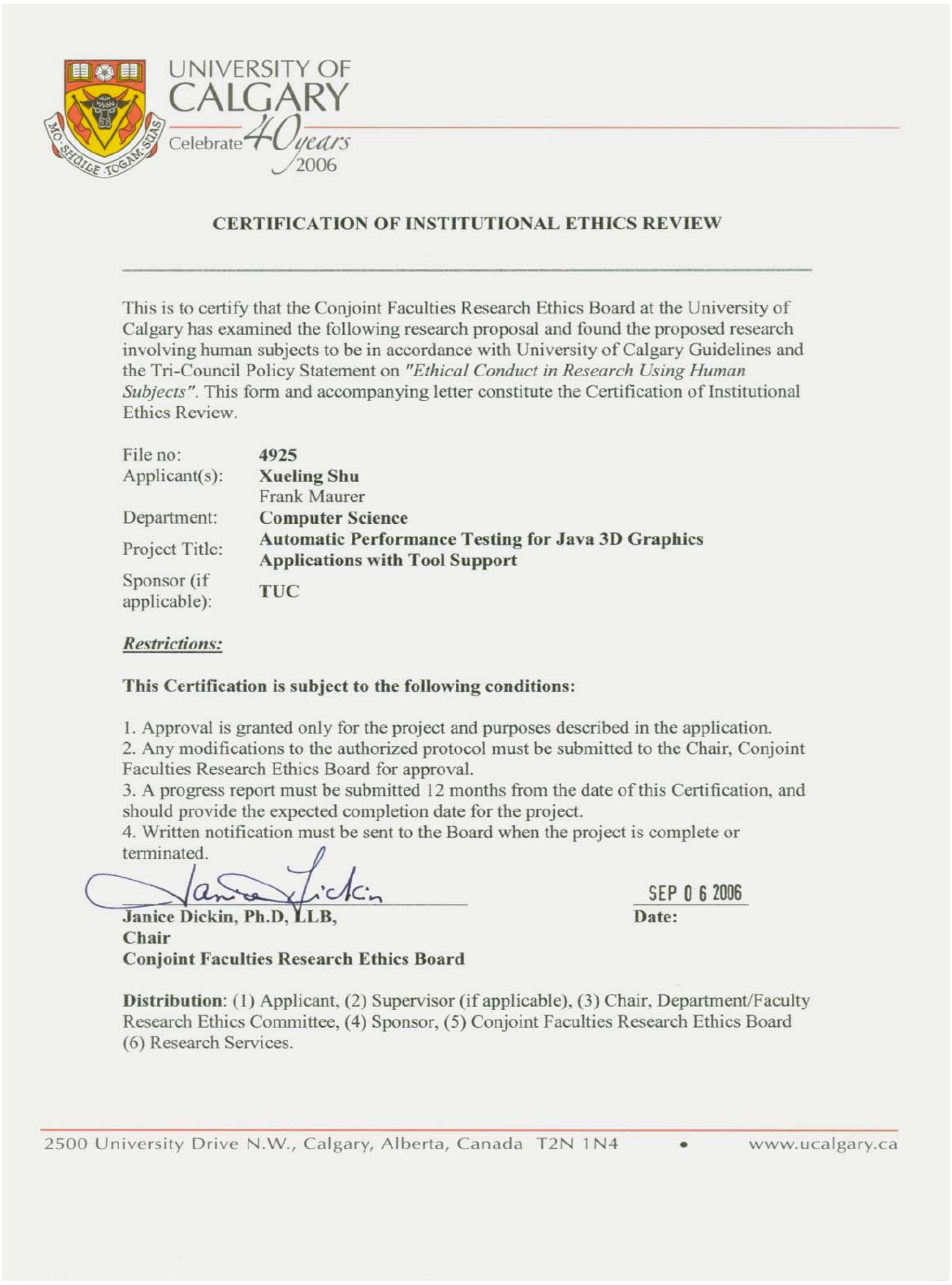
Follow-up interviews and discussions may be conducted to further understand participants' answers.

Would you like to be interviewed (by telephone or face to face)? Yes ___ , No ___. If Yes, Please provide your telephone number _________________ and I will email you to confirm the date and time.

Thanks for your support!

Xueling Shu

## A.3. Ethic Approval



UNIVERSITY OF CALGARY

Celebrate 40 years 2006

**CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW**

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

| | |
|---|---|
| File no: | **4925** |
| Applicant(s): | **Xueling Shu**<br>Frank Maurer |
| Department: | **Computer Science** |
| Project Title: | **Automatic Performance Testing for Java 3D Graphics Applications with Tool Support** |
| Sponsor (if applicable): | **TUC** |

***Restrictions:***

**This Certification is subject to the following conditions:**

1. Approval is granted only for the project and purposes described in the application.
2. Any modifications to the authorized protocol must be submitted to the Chair, Conjoint Faculties Research Ethics Board for approval.
3. A progress report must be submitted 12 months from the date of this Certification, and should provide the expected completion date for the project.
4. Written notification must be sent to the Board when the project is complete or terminated.

Janice Dickin

**Janice Dickin, Ph.D, LLB,**
**Chair**
**Conjoint Faculties Research Ethics Board**

SEP 0 6 2006
**Date:**

**Distribution**: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca

# APPENDIX B: TOOL EVALUATION

## B.1. Evaluation Introduction

Hello my name is Xueling Shu, a graduate student at the University of Calgary, Department of Computer Science. I am doing a thesis based Masters Degree. Dr. Frank Maurer is my supervisor.

Our research aims to bring forth more efficient ways to conduct performance testing for Java 3D applications. We developed a testing tool - J3DPerfUnit - to perform automatic performance testing for Java 3D applications and would like to evaluate J3DPerfUnit's effectiveness and efficiency. The analysis of the results from questionnaires, interviews, and research notes, will provide insights for the research in automatic performance testing on Java3D applications and the tool's further development.

If you volunteer as a participant, you will go through two phases:

1. Participate in an evaluation session including a tool demonstration and introduction.
2. Fill in a questionnaire which lists questions about the tool's effectiveness and efficiency.

You may be invited to open ended interviews to let us further understand your answers on the questionnaire. We won't expose your name in our study. Some interview sessions may be tape recorded with your permission to facilitate collection of information, but being taped recorded is not a required condition to take part in our research.

You may withdraw whenever you want without any negative consequences, and choose whether to have data collected to the point of withdrawal excluded from the study or retained/used by the researchers. At the same time, the decision to participate (or not) will in no way affect students' grades in any course, and that the instructor will not know who has or has not chosen to participate.

The study has received formal ethics clearance thorough Conjoint Faculties Research Ethics Board. Participation in the research is voluntary and there are no known or anticipated risks involved.

If you are willing to participate, please fill out the following consent form and we will be in touch with you.

## B.2. Questionnaire

### Background

1. Programming experience with Java 3D
   Number of months: _______

### J3DPerfUnit

***Test Case Definition***

2. The definitions are appropriate.
   ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

3. Please point out all the aspects you think need to improve regarding test case definitions? (You can make a list based on test case categories, such as: for a building scene graph test, it is better for J3DPerfUnit to do…)

***Test Result and Report***

4. The feedback was accurate.
   ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

5. The feedback helped locate performance problems in the application
   ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

6. Please specify all the difficulties you encountered when understanding test result and report?

***Overall***

7. Using this new tool will enhance the overall effectiveness and efficiency of Java3D performance testing.
   ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

8. The tool is easy to apply during the case study.
   ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

**Comparison**

9. What is your previous method for performance testing? (Manually or using tools? If using tools, please specify what they are)

10. You prefer using the previous method to J3DPerfUnit. Please explain your answer.
    ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

11. You will use J3DPerfUnit in the future to conduct Java3D performance testing.
    ( ) Strongly agree ( ) Agree ( ) Neutral ( ) Disagree ( ) Strongly disagree

**Summary**

12. What are the choices you think can appropriately describe J3DPerfUnit:

    ( ) Increasing test-coverage

    ( ) Improving the quality of Java3D application

    ( ) Test results provided hints on possible performance problems.

    ( ) Executes tests effectively and efficiently

    ( ) Increases your overall development efficiency

    ( ) others: Please fill in ______________________________

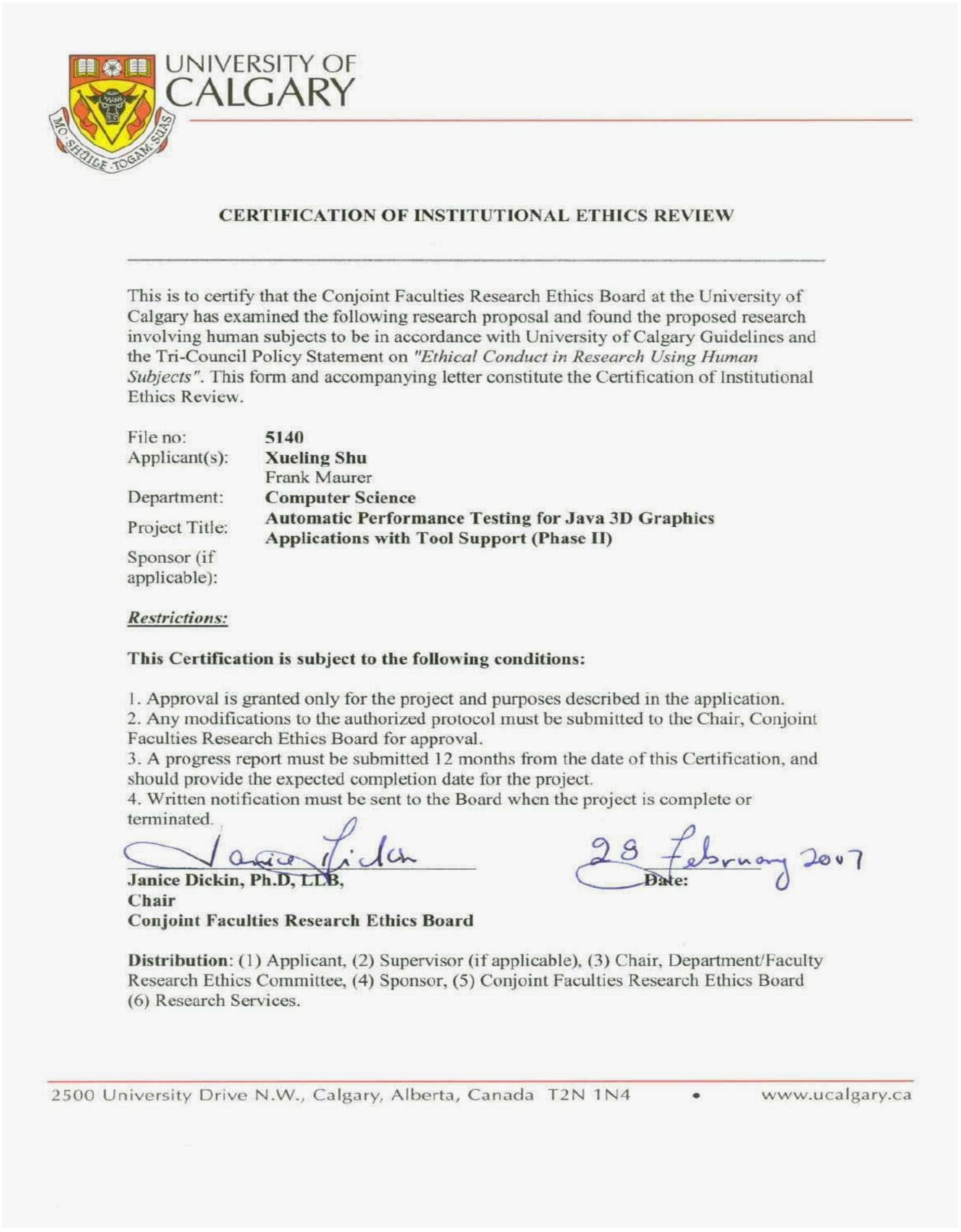
Follow-up interviews and discussions may be conducted to further understand participants’ answers.

Are you willing to be interviewed (by telephone or face to face)?  Yes ___   ,   No ___. If Yes, Please provide your telephone number _________________ and e-mail ____________________ and I will email you to confirm the date and time.

Thanks for your support!

Xueling Shu

## B.3. Ethic Approval



UNIVERSITY OF CALGARY

MO SHÙILE TOGAM SUAS

**CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW**

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

| | |
|---|---|
| File no: | **5140** |
| Applicant(s): | **Xueling Shu**<br>Frank Maurer |
| Department: | **Computer Science** |
| Project Title: | **Automatic Performance Testing for Java 3D Graphics Applications with Tool Support (Phase II)** |
| Sponsor (if applicable): | |

***Restrictions:***

**This Certification is subject to the following conditions:**

1. Approval is granted only for the project and purposes described in the application.
2. Any modifications to the authorized protocol must be submitted to the Chair, Conjoint Faculties Research Ethics Board for approval.
3. A progress report must be submitted 12 months from the date of this Certification, and should provide the expected completion date for the project.
4. Written notification must be sent to the Board when the project is complete or terminated.

Janice Dickin

**Janice Dickin, Ph.D, LLB,**
**Chair**
**Conjoint Faculties Research Ethics Board**

28 February 2007
**Date:**

**Distribution**: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca

## APPENDIX C: MATERIALS AND RAW DATA

1. Pre-development survey questionnaires and email interview notes are in folder "Pre_ Development _Survey"
2. Tool evaluation questionnaires, notes for email interviews and on-site discussions  are in folder "Post_ Development _Survey"
3. Pre-development survey charts and raw data are in file "PreStudy.xls" under folder "Analysis".
4. Tool evaluation charts and raw data are in file "PostStudy.xls" under folder "Analysis".
5. Presentation slides for tool evaluation are located in folder "Post_Development_Survey".
6. J3DPerfUnit source code is in folder "SourceCode".