INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Design and Evaluation of Java Game Programming Environment

Ying Dong

A Major Report

in

The Department

of

Computer Science

Submitted in Partial Fulfillment of The Requirements for The Degree of Master of Computer Science

February 2002

© Ying Dong, 2002

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72934-6

Canada

ABSTRACT[1]

This report is an exploration of basic principles of game programming with Java. In this report, a basic framework is discussed and provided for programming Java games. Some issues related to Java technology are also researched and placed forward. On the base of the basic principles for Java game programming, an online game is designed and implemented. As this is a project jointly with Ye Zhu, in this report the design of the game is only briefly described.[1] Some common suggestions on writing Java programs are also provided for making fast and efficient Java games. All in all, this report not only covers knowledge on Java game programming, but also some concepts related Java technology and object-oriented programming.

[1] For the detail of the design and implementation of this game, please refer to the collaborated work Ye Zhu's report [13].

Table of Contents

Chapter 1: Introduction 1

Chapter 2: Introduction of Java and the reasons choose this topic 3
2.1 What is Java? 3
2.1.1 The History of Java 3
2.1.2 About the Java Technology 6
2.2 What Can Java Technology Do? 11
2.3 Can Java be used for Games? 13
2.4 Why choose this topic as a graduate level work? 15

Chapter 3: Java Game Programming 17
3.1 Graphics 17
3.1.1 Creating a Graphics Engine 17
3.1.2 Movable Object Blocks 18
3.2 Construction of a Graphics Engine 20
3.3 Painting Images from a List 23
3.4 Installing the Graphics Engine 24
3.5 Improvements 27
3.6 Double Buffering 31
3.7 Invisibility and Other Possible Extensions 32
3.8 Sound Effect 33
3.9 Java-Specific Game Design Issues 33
3.9.1 User Interface 34
3.9.2 Limiting Factors 37

Chapter 4: Case study -- *CrazyRoad1.0* (Online Game) 42
4.1 Brief Description 42
4.2 Design 43

Chapter 5: Improvement and fallacies on Java 48
5.1 Some suggestions for Java game programming 48
5.2 Some fallacies on Java 51
5.3 Java's improvements over C and C++ 52

Chapter 6: Summary 56

References: 58

List of Figures

Figure 1. How Java program works 8

Figure 2. Java Virtual Machine (JVM) 9

Figure 3. A Java program running on the Java platform 10

Figure 4. Java 2 SDK 13

Figure 5. The interface of the implemented game – CrazyRoad1.0 42

Figure 6. Class diagram for the implemented game – CrazyRoad1.0 46

Chapter 1: Introduction

These days the word *Internet* is practically synonymous with the word *computer*. More and more people are buying new computers just so they can surf the World Wide Web or keep in touch with friends and family through e-mail. Increased connectivity among users, as well as an exponential increase in the power of modern hardware and software, has created a gray area between where the personal computer ends and the network begins.

The cyber culture of the Internet has undoubtedly impacted our daily lives more than we imagine. Most of the people who live with Internet browse the Web daily looking for information and news. Some of us use the Internet for academic research. Some use it to get some useful information such as road maps, ads or news etc. And for some, the magic of the Web lies in the ability it affords one to play games online.

The video game industry is a multi-billion-dollar-per-year business. Most of this revenue comes from games for home consoles or shrink-wrapped PC games. [1] This large market affords anyone the perfect opportunity to develop more powerful and interesting applications in computer gaming. Whether it is a fresh new gaming genre geared towards online play or a new marketing scheme to help existing online games make money, opportunity abounds. We can say that there is still plenty of room for people to make their marks on the gaming industry. According to some statistics, Java games often appeal to the 85% of computer users who are considered the real gamers [1], because Java games are usually simple, fun games that require minimal hardware requirements from the user. People can use this fact to spread their games to all types of gamers.

This report will show the basic principles to use Java language to create robust, flexible

gaming applications. It covers the fundamental ideas of creating games using Java with a brief description of a case study (an online game implementation – CrazyRoad1.0). This report also describes the work carried out in collaboration with YE Zhu [13].

Chapter 2: Introduction of Java and the reasons choose this topic

2.1 What is Java?

2.1.1 The History of Java

Essentially the Java story begins in about 1990 when a programmer at Sun Microsystems (a computer company best known for the manufacture of powerful and expensive *Unix Workstations*), Patrick Naughton told his boss Scott McNealy that he was quitting to join a rival company (NeXT Computers). Being highly regarded, McNealy was not too happy about this; therefore, he asked Naughton a favor - namely asking him to write a report on what Sun was doing wrong.

Without going into too much detail, Naughton produced a document which was a series of gripes boiling down to a plea for Sun to concentrate on less disparate activities. This proposal was then circulated too much of the management of the company, producing feverish statements of agreement, and ended up in Sun creating a small team to pursue Naughton's ideas, shielded from the more cautious elements of the company who might have been wondering where the money was going.

They decided to focus on consumer electronics applications first of all. And this led to a demo to McNealy in 1992 where from a small hand held Psion type contraption, it was made possible to program a video recorder all at the touch of a button. The idea was to integrate the smallest electronic device with cyberspace using a standard language all of their own that would have Internet connectivity at its very heart. At the beginning its code name was "Oak". However, back then it was a bit far ahead of its time.

Consequently, there were a series of dispiriting false dawns. A possible team up with Mitsubishi & French Telecom was floated but it never came to anything. In fact Sun

scrapped the team (by now named *FirstPerson*) in 1994, and just plugged away at their traditional *high-end server* market. However very shortly after this, the Mosaic Web Browser emerged (the first web browser to be able to view pictures) and the sudden growth of the World Wide Web made the folk at Sun reconsider. Having seen the potential of Mosaic, and its commercial elaboration by Netscape, a senior manager at Sun, Bill Joy, took on the challenge of turning Java into an explicitly Internet language. This indeed was the first real public incarnation of Java: as applets appearing on web pages. An applet, as its name might suggest, is a mini application, a small entity of interactivity that executes on our machine within a web browser. An experimental web browser written in Java, called WebRunner and later renamed as HotJava, ready by the end of 1994 was the host in which those applets "ran".

Although we are wizened now by the amount of interactive frivolity we have chanced upon in our surfing, it is important not to underestimate how revolutionary the concept seemed back then. Karl Jacob, CEO and chief technologist at Dimension X Inc, a San Francisco company creating 3-D websites using Java said at the time: "Java allows us to do the things that advertisers and studios are asking us to do...Until now, everything on the Web was fizzling, not sizzling."

So what made Java suitable for the Internet then? A mixture of the following things:

- **Simple and Familiar:** that is to say, it looks quite similar to C and C++ for programmers of those languages, but with many of the elements which sometimes prove troublesome (pointers, memory de-allocation) removed
- **Object-oriented:** a program created in Java is essentially the weaving together of various objects (mixtures of code and data) that are instantiations of various

"classes". However, Java is very strictly so.

- **Architecture-neutral:** because it is an interpreted language, it requires only an Interpreter or Just in Time Compiler otherwise known as VM (***The Java Virtual Machine***) in order to run. So as long as we have the VM for the machine we want to run it on, then all Java programs should theoretically run.
- **Portable:** because of the above, in theory we should only have to write it once, and it should work on all machines, though this is still not true in practice all the time.
- **Distributed:** a program can create objects from classes stored on different web servers - and only download the ones appropriate for a particular group of actions at a particular time.
- **Secure:** a whole layer of Java's architecture is devoted to security, so that it formalizes the way to deal with possible security problems (since we are allowing a program to run on our machine from an internet site).

In fact Sun posted the first Internet ready version of Java on an obscure Internet server in 1994 and invited Netscape and some select others to look at it. Marc Andreeson, chief luminary of Netscape was impressed and told the San Jose Mercury News: "What these guys are doing is undeniably, absolutely new. It's great stuff." This was very auspicious: indeed Kim Polese, who later went on to become CEO of the company Marimba, and was at that time Senior Product Manager for Java said: "That quote was a blessing from the god of the Internet".

Suddenly everything was looking right. By May 95 Netscape had licensed Java from Sun and built a version of it into Navigator 2.0. By the autumn of that year, the first books

about Java were being written and published. On Dec 4 1995 Business Week ran a cover story on "Software Revolution --- The Web Changes Everything" and promoted Java as a breakthrough force in the expanding the Web and the Internet. In the following week, Silicon Graphics, IBM, Adobe, Macromedia and finally Microsoft adopted and licensed Java from Sun.

In all of these things, Sun used a two pronged software strategy to increase interest in Java. a) They produced their web browser written in Java called *Hot Java* b) The freely distributed "Java Development Kits" JDKs which contained a standard compiler, an applet viewer, all the documentation for code, and a documentation generator (Javadoc) which produced documentation in one standard form.

The Java revolution first seemed to really solidify with the release of Netscape and Internet Explorer 3.0, and also introduced some much needed competition in the realm of Java virtual machines (since Microsoft's was very superior to Netscape's on the PC at least). This version of Java was 1.0.x and became the standard for a while.

Not long after 1.0.x a new version then came out 1.1.x which has a number of significant improvements: a more sophisticated support for User Interface components, together with so-called Java Beans, things that behave rather like OLE objects on Windows (i.e. applications that can host or reside in other applications - for example Equation Editor inside Word) - however these would be totally cross platform objects. These years Java improves very fast and several new versions have come out. These new versions include Java 1.2, 1.3 and 1.4 and all of them improve significantly.

2.1.2 About the Java Technology

Java technology is both a programming language and a platform.

2.1.2.1 The Java Programming Language

According to Java's language specification by Sun, the Java programming language is a high-level language that can be characterized by the following words:

Simple, object-oriented, distributed, robust, secure, architecture neutral, portable, high performance, multithreaded and dynamic.

Apart from those been described above, the other properties are:

- **High performance**: *Performance* is always a consideration. The Java platform achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The *automatic garbage collector* runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance. Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform. In general, users perceive that interactive applications respond quickly even though they're interpreted.
- **Multithreaded and dynamic:** The Java platform supports multithreading at the language level with the addition of synchronization primitives: the language library provides the *Thread* class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java technology's high-level system libraries have been written to be *thread safe*: the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution. While the Java Compiler is strict in its compile-time static checking, the language and run-time system are *dynamic* in their linking stages.

> Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network. In the case of the HotJava Browser and similar applications, interactive executable code can be loaded from anywhere, which enables transparent updating of applications. The result is on-line services that constantly evolve; they can remain innovative and fresh, draw more customers, and spur the growth of electronic commerce on the Internet.

With most programming languages, we either compile or interpret a program so that we can run it on our computer. The Java programming language is unusual in that a program is both compiled and interpreted. With the compiler, first we translate a program into an intermediate language called *Java bytecodes*, the platform-independent codes interpreted by the interpreter on the Java platform. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed. The following figure illustrates how this works.

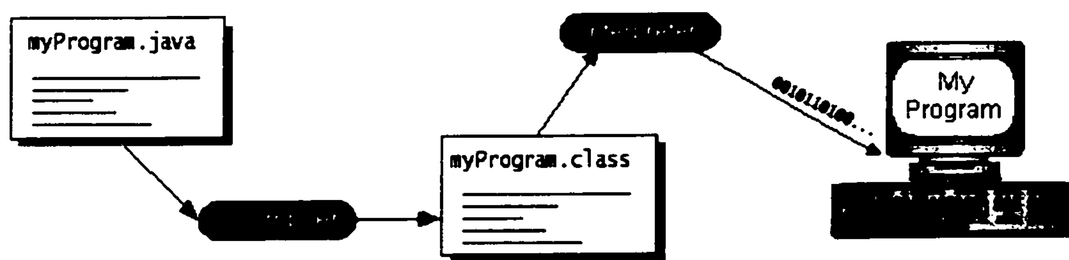

Figure 1. How Java program works

We can think of Java bytecodes as the machine code instructions for the *Java Virtual*

Machine. Every Java interpreter, whether it's a development tool or a Web browser that can run applets, is an implementation of the Java VM.

Java bytecodes help make the saying by the sun people "write once, run anywhere" possible. We can compile our program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java VM. This means that as long as a computer has a Java VM, the same program written in the Java programming language can run on Windows 2000, a Solaris workstation, or on an iMac.

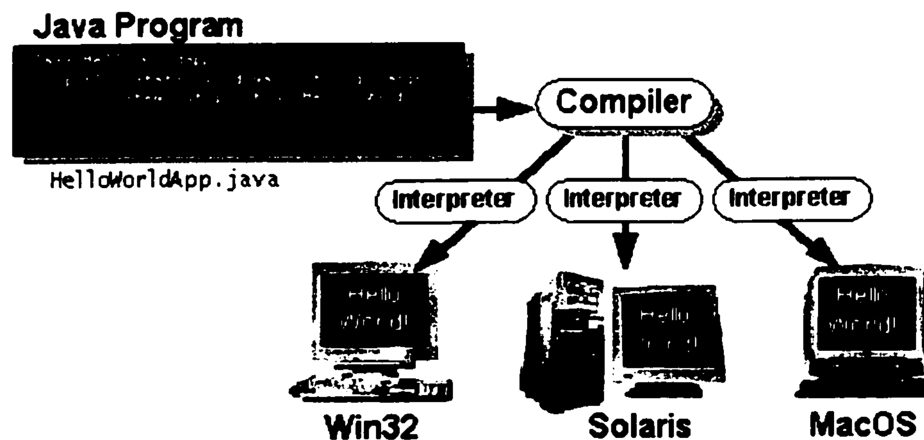

Figure 2. Java Virtual Machine (JVM)

2.1.2.2 The Java platform

A platform is the hardware and software environment in which a program runs. We have already mentioned some of the most popular platforms like Windows 2000, Linux, Solaris, and MacOS. Most platforms can be described as a combination of the operating system and hardware. The Java platform differs from most other platforms in that it is a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

- The *Java Virtual Machine* (Java VM)
- The *Java Application Programming Interface* (Java API)

Java VM is the base for the Java platform and is ported onto various hardware-based platforms.

The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.

The following figure depicts a program that is running on the Java platform. As the figure shows, the Java API and the virtual machine insulate the program from the hardware.

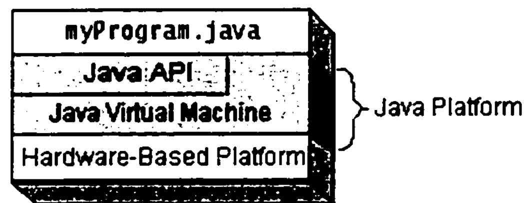

Figure 3. A Java program running on the Java platform

Native code is the compiled machine code that runs on a specific hardware platform. As a platform-independent environment, the Java platform can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time bytecode compilers can bring the performance close to that of native code without threatening portability.

2.2 What Can Java Technology Do?

The most common types of programs written in the Java programming language are *applets* and *applications*. An applet is a program that adheres to certain conventions that allow it to run within a Java-enabled browser.

However, the Java programming language is not just for writing entertaining applets for the Web. The general-purpose, high-level Java programming language is also a software platform. Using the generous API, we can write many types of programs.

An application is a standalone program that runs directly on the Java platform. A special kind of application known as a *server* serves and supports clients on a network. Examples of servers are Web servers, proxy servers, mail servers, and print servers. Another specialized program is a *servlet.* A servlet can almost be thought of as an applet that runs on the server side. Java Servlets are a choice for building interactive web applications, replacing the use of CGI scripts. Servlets are similar to applets in that they are runtime extensions of applications. Instead of working in browsers, though, servlets run within Java Web servers, configuring or tailoring the server.

How does the API support all these kinds of programs? It does so with packages of software components that provide a wide range of functionality. Every full implementation of the Java platform gives us the following features:

- **The essentials**: Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on.
- **Applets**: The set of conventions used by applets.
- **Networking**: URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses.

- **Internationalization**: Help for writing programs that can be localized for users worldwide. Programs can automatically adapt to specific locales and be displayed in the appropriate language.
- **Security**: Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.
- **Software components**: Known as JavaBeans™, can plug into existing component architectures.
- **Object serialization**: Allows lightweight persistence and communication via Remote Method Invocation (RMI).
- **Java Database Connectivity (JDBC™)**: Provides uniform access to a wide range of relational databases.

The Java platform also has APIs for 2D and 3D graphics, accessibility, servers, collaboration, telephony, speech, animation, and more. The following figure depicts what is included in the Java 2 SDK.

Figure 4. Java 2 SDK

2.3 Can Java be used for Games?

Can Java be used to write games? As a language for games, it should have comparatively large set of rich libraries and good ability to deal with graphics, and also from the modern point of view, it better be object-oriented. We know that Java is object-oriented, and up to now it has a large set of rich packages (libraries), and after its 2D and 3D packages came to the world, now it has the necessary ability to deal with graphics. So Java has all the necessary features for writing games. But we may still ask that why choose Java for games. The decision whether to use Java over languages such as C or C++ is not always clear. There are situations where people may want to use one language and situations where another might be more appropriate. For instance, using Java to create today's latest and greatest first-person 3D shooter may not be such a wise choice – at least not until more development tools are created. On the other hand, it can be guaranteed that object-oriented programmers can get that 2D done faster and more efficiently with Java than with practically any other language out there. So for the people who like to write their

2D games, Java is really a good choice. Following are some main reasons why people might choose Java for a particular game:

- **Platform Compatibility**

Practically all Java programs written on one system will automatically run on any other system that supports the Java platform. Therefore, the exact same code can be executed on virtually any system, without any code changes needed. A game created with Java can be sold to someone running Windows XP just as it can to someone running an older vertion of Linux.

- **Internet Capabilities**

Much of the Java language is tailored for the Internet. Therefore, we can easily create and run games for the Web using Java. This includes everything from downloading simple games like checkers or black jack and running them directly through the web browsers to playing multiplayer role-playing games online.

- **The Java API**

Java comes with a rather impressive number of data structures and other packages, saving the programmer from having to perform many menial, low-level tasks. The Java Abstract Window Toolkit (AWT), Java 2D and Java project Swing are just a few Java technologies available to help people write better code faster. In short, the Java API allows people to focus on the problem at hand, rather than worrying about many of the underlying details.

- **Object-Oriented Design**

Object-oriented design makes programming faster and easier. Large tasks can be broken down into smaller ones, and object-oriented programs are also generally easier to follow

and manage than the ones written using procedural languages.

- **A Shallower Learning Curve**

Game programming requires knowledge of mathematics, physics, graphics programming, artificial intelligence, and so on. This can be quite difficult for many people who like to write their own games. Using a language with so many built-in constructs (these constructs can deal with the calculations, image processing or other computing which are some critical required functionalities for game programming) allows newcomers to the field of game programming to feel comfortable more quickly, allowing them to break down the projects into small, manageable steps.

2.4 Why choose this topic as a graduate level work?

Since the occurrence of Java, Java has improved and spread so fast that nowadays more and more software developers and programmers are using Java technology to design and develop software programs. More and more companies are using Java to bind or implement their products, especially in e-commerce business field. With every new version or new product of Java, it often brings us new conception, new thinking and new technologies. So to some degree catching Java technology make people not behind the world of modern computer science. From the above description we can know that Java is a high-level programming language that has a large set of packages, powerful functionalities in many fields and a lot of good features, and game programming requires so much knowledge of mathematics, physics, graphics programming, artificial intelligence, especially good understanding of object-oriented concept and so on, so I think this is quite a good choice for a graduate level work

In this project we designed a game programming environment in Java. This environment has been evaluated by the collaborated work of Ye Zhu [13] by designing a Java game applet for this environment.

Chapter 3: Java Game Programming

The technologies used when creating games with Java is a lot like creating games with other languages. There are usually some basic components that consist a game, such as movable objects, some backgrounds, friendly graphical user interface etc. In general, dealing with graphics is the most important task in game programming, and modern games are much more advanced and powerful than before, Sound is becoming the second most important task in game programming. Usually almost in any kind of languages, the basic idea for game programming is that we need to deal with the design of movable objects since a game usually contains lot of movable objects in it, we need to deal with the design of a graphic engine to keep track of the movable objects, that is, we need some methods to control those movable objects, and we also need to deal with double buffering to make the movement of the movable objects look smooth.

Fortunately Java takes care of a lot of the low level or tedious work we would have to do if we were writing a game in another language. For example, Java provides built-in support for transparent pixels, making it easier to write a graphics engine that can draw nonrectangular objects. Java also has built-in support for allowing several different processes or tasks to run at once - perfect for creating a word with a lot of creatures, each with its own special methods for acting. And, the easy implementation of multi-threading also makes Java a better way for game application.

3.1 Graphics

3.1.1 Creating a Graphics Engine

A *graphics engine* is essential to a well-designed game in Java. It is an object that is

given the duty of painting the screen. It keeps track of all the objects on the screen at one time, the order in which to draw the objects and the background to be drawn. The most important function of the graphics engine is to maintain the movable object blocks.

3.1.2 Movable Object Blocks

As we usually can see, there are a lot of movable object blocks in a game. These blocks make our life infinitely easier if we are interested in creating a game that combines graphics and user interaction, as most games do. To make the movable object blocks, we have two alternative ways in Java. One is using some methods that are related to drawing graphics in Java to draw those objects. This method is very tedious and usually can only be used in simple animation programs. The other way, which we actually use in game programming, is that we make the object as such an object which contains both a picture that will be drawn on the screen and the information that tells us where the picture is to be drawn on the screen. To make the object move, we simply tell the movable object (more precisely, the graphics engine that contains the movable object) which way to be drawn and we are done -- redrawing is automatically taken care of.

The basic method for making a movable object block (here after it is called **MOB**) in Java is shown in **Listing 1**:

Listing 1. The code creating a movable object block (MOB).

```
import java.awt.*;
public class MOB
{
  public int x = 0;
  public int y = 0;
  public Image picture;
  public MOB ( Image pic )
  {
     picture = pic;
   }
}
```

As we can see from the above code, the movable object consists of an image and a set of coordinates. The constructor takes an Image and stores it away to be drawn when needed. After we have instantiated an **MOB** (that is, after we have called the constructor), we have a movable object that we can move around the screen just by changing its x and y values. The graphics engine will take care of redrawing the movable object in the new position.

One thing to consider is the nature of the picture that is going to be drawn every time the movable object is drawn. Consider the place in which the image probably will originate. In all likelihood, the picture will either come from a *GIF* or *JPEG* file, which has one very important consequence -- it will be rectangular. What does this mean? This means that if the entire movable objects are rectangles, the characters would be drawn, but so would their backgrounds. The chances are, we will want to have a background for the entire game; it would be unacceptable if the unfilled space on character images covered up our background just because the images were rectangular and the characters were of another shape.

When programming games in other languages, this problem is often resolved by examining each pixel in a character's image before drawing it to see whether it is part of the background [2]. If the pixel is not part of the background, it is drawn as normal. If the pixel is part of the background, it is skipped and the rest of the pixels are tested. Pixels that are not drawn usually are referred to as *transparent pixels*. Fortunately, Java has built-in support for transparent colors in images, which simplifies our task immensely. We do not have to check each pixel for transparency before it is drawn

because Java can do that automatically. Java even has built-in support for different levels of transparency. For example, we can create pixels that are 20 percent transparent to give our images a ghostlike appearance. Here we will deal only with fully transparent pixels. Java's capability to draw transparent pixels makes the task of painting movable objects on the screen much easier. But how do we tell Java what pixels are transparent and what pixels are not? Now we have two alternative ways to tell Java which pixels are transparent and which are not. One way is to load the image and run it through a filter that changes the *ColorModel*, but that would be a very hard way because firstly we have to be very familiar with dealing images and secondly we have to pass a lot of information related to the image filter to the image consumer in which way usually it requires lot of coding. The other way is that because Java supports transparent *GIF* files, whenever a transparent GIF file is loaded, all the transparency is preserved by Java. This means our job got a lot easier. Now what we need to do is just use our favorite graphics package to create a *GIF* file (or a picture in some other format that we can eventually convert to a *GIF* file). Select a color that does not appear anywhere in the picture and fill all areas that we want to be transparent with the selected color. Make a note of the *RGB* value of the color we use to fill in the transparent places. Now we can use a program to convert our *GIF* file into a transparent *GIF* file.

3.2 Construction of a Graphics Engine

When we have the movable objects that know where they are supposed to be and do not eat up the background as they go there, we need to design something which keeps track of the movable objects and draw them in the proper places when necessary. This is the

job of *GraphicsEngine* class. Listing 2 shows the bare bones of a graphics engine.

Listing 2. A bare-bones of graphics engine that tracks movable objects.

```
import java.awt.*;
import java.awt.image.*;
public class GraphicsEngine
{
        Chain mobs = null;
        public GraphicsEngine()
        { }

        public void AddMOB (MOB new_mob)
        {
           mobs = new Chain (new_mob, mobs);
        }

        public void paint (Graphics g, ImageObserver imob)
        {
            Chain temp_mobs = mobs;
            MOB mob;
            while (temp_mobs != null)
            {
               mob = temp_mobs.mob;
               g.drawImage(mob.picture, mob.x, mob.y, imob);
               temp_mobs = temp_mobs.rest;
            }
        }
      }

class Chain
{
        public MOB mob;
        public Chain rest;
        public Chain (MOB mob, Chain rest)
        {
           this.mob = mob;
           this.rest = rest;
        }
}
```

This is the minimum we need to handle multiple movable objects.

Before we detail how the *GraphicsEngine* class works, we first briefly describe the *Chain* class. The *Chain* class is simply a data structure that holds two objects, which are called *item* and *rest* in the above code. The power of the *Chain* structure – and those structures like it -- is that it can be used as a building block to create a multitude of more complicated structures. These structures include circular buffers, binary trees, weighted digraphs and linked list etc. Here we actually use *Chain* class to create a linked list.

Our goal here is to keep a list of moveable objects that are to be drawn. A linked list suits our purpose well because a linked list is a structure that is used to store a list of objects. Each point in the list contains an object and a link to a list with the remaining objects. Since linked list is the basic knowledge in computer science, I will not talk about it more here.

Look at the method *AddMOB()* in **Listing 2**. Whenever we want to add another movable object to the list of movable objects we are controlling, we simply make a new list of movable objects that has the new movable object as the first item and the old *Chain* as the rest of the list. Here the reason why we make a method called *AddMOB()*, which is only one line long, is that if *GraphicsEngine* were subclassed in the future, it would be a lot easier to add functionality if all we have to do is override one method as opposed to changing every line of the code that calls *AddMOB()*. For example, if *GraphicsEngine* is subclassed by some other classes, and if we want to sort all the movable objects by size, with having the method *AddMOB()* we could just override *AddMOB()* so that it sorts all the objects in a sorted order to begin with. Without *AddMOB()*, we usually have to change every line of the code that possesses the similar functionalities to *AddMOB()* in the subclasses.

3.3 Painting Images from a List

Since we already have a method for keeping a list of all the objects that have to be painted, to use the list, the first thing we should be concerned about is how to add new objects to the list of objects that have to be painted. We add a new object to the list by using the method *AddMOB()* shown in **Listing 2**. As we can see that all the *AddMOB( )* method does is to replace the old list of objects in *mobs* with a new list that contains the new object and a link to the old list of objects.

When *AddMOB( )* has been called for all the removable objects we want to handle, from the method *paint( )* we can see that the first thing to do is to copy the pointer (though Java does not have explicit pointers, we can think it in this way) to *mobs* into a temporary *Chain* called *temp_mobs*. Now *temp_mobs* contains a pointer to the list of all the removable objects to be drawn. Then we should go through the list and draw each movable object. The variable *mob* is used to keep track of each movable object as we get to it. The variable *temp_mobs* represents the list of movable objects we have left to draw (that's why we started it from pointing to the whole list). When *temp_mobs* is *null* we can know that all movable objects have been drawn because that will be just like saying that the list of movable objects left to draw is empty. That's why the main part of the code is encapsulated in a *while* loop that terminates when *temp_mobs* is *null.* Simply speaking, we draw the movable objects by traversing the whole list.

We can also notice that in the *while* loop of the *paint( )* method, the first thing that is done is to assign *mob* to the movable object at the beginning of the *temp_mobs Chain* so that there is an actual movable object to deal with. Now it is the time to draw the movable object. The *g.drawImage( )* command draws the movable object in the proper place. The

variable *mob.picture* is the picture stored earlier when the movable object was created. The variable *mob.x* and *mob.y* are the screen coordinates at which the movable object should be drawn. We should notice that *paint( )* looks at these two variables every time the movable object is drawn. So changing one of these coordinates while the program is running has the same effect as moving it on the screen. The final argument passed to *g.drawImage( ) – imob*, is an *ImageObserver* that is responsible for containing those images that are to be drawn. The way we get an *ImageObserver* is that we will use the *GraphicsEngine* class to draw inside a *Component* (or a subclass of *Component* such as *Applet*), and in Java a *Component* implements the *ImageObserver* interface so that we can just pass the *Component* to *GraphicsEngine* whenever we want to repaint. For example, we can call *paint()* method of class *GraphicsEngine* in the way like *paint(g, myApplet)*, where *g* is an instance of *Graphics* and *myApplet* is an instance of *Applet*.

The final line inside the *while* loop shortens the list of movable objects that have to be drawn. It points *temp_mobs* away from the *Chain* that it just drew a movable object off the top of and points it to the *Chain* that contains the remainder of the **MOB**s. As we continue to shorten the list of **MOB**s by pointing to the remainder, *temp_mobs* eventually winds up as *null*, which ends the *while* loop with all the movable objects drawn.

3.4 Installing the Graphics Engine

The graphics engine in **Listing 2** certainly had some important things left out, but it does work. Having the *GraphicsEngine* class, we have to install it inside a *Component* to use it. **Listing 3** shows an example of how to install the *GraphicsEngine* in side a Component:

Listing 3. A sample applet installing *GraphicsEngine*

```
import java.awt.*;
import java.applet.Applet;
import java.net.URL;
public class Game extends Applet
{
        GraphicsEngine engine;
        MOB picture1;
        public void init()
        {
            try
            {
                engine = new GraphicsEngine();
                Image image1 = getImage( new URL(getDocumentBase(),
                                        "one.gif"));
                picture1 = new MOB(image1);
                engine.AddMOB(picture1);
            }
          catch (java.net.MalformedURLException e)
          {
             System.out.println("Error while loading pictures...");
             e.printStackTrace();
          }
        }

        public void update(Graphics g)
        {
           paint(g);
        }

        public void paint (Graphics g)
        {
          engine.paint(g, this);
        }
        public boolean mouseMove (Event evt, int mx, int my)
        {
           picture1.x = mx;
           picture1.y = my;
           repaint();
           return true;
        }

}
```

The *Component* we are installing the *GraphicsEngine* in is an *Applet*. So we can view the results with a web browser or with JDK's *appletviewer*. In the above code, the instance of *GraphicsEngine* is the variable *engine*. It controls all the movable objects we can deliver, and *picture1*, a movable object that draws the chosen image. In the *init( )* method, we initialize *engine* by setting it equal to a new *GraphicsEngine*. Next, the image we chose is loaded with a call to *getImage( )*. This line creates the *try* and *catch* statements that surround the rest of the code to catch any invalid *URLs*. After the image is loaded, it is used to create a new *MOB*. And *picture1* is initialized to this new *MOB*. The work is completed by adding the movable object to *engine* so that *engine* will draw it in the future. The remaining lines are there to provide information about any errors that occur.

In the *paint( )* method we can see that it actually calls the *paint( )* method in the graphics engine.

The update( ) method is used to avoid flickering. By default, applets use the *update( )* method to clear the window they live in before they repaint themselves with a call to their *paint( )* method. This can be a useful feature if we are changing the display only once in a while, but with graphics intensive programs, this can create a lot of flicker because the screen refreshes itself frequently. Because the screen refreshes itself so frequently, once in a while, it catches the applet at a point at which it has just cleared its window and has not yet had a chance to redraw itself. This is what causes flicker. So we have to rewrite the *update()* method to solve this problem.

The flicker was eliminated here by leaving out the code that clears the window and going straight to the *paint( )* method. But here another problem arises. It is that the movable object is leaving streaks. This is because this method only uses a single buffer to do the

redrawing. As now we don't clear the window, the same object will be redrawn within the same buffer but at different places with the moving of this object, so the streaks happen here. Double buffering that will be talked about later can eliminate this.

As we can see, the *Game.paint( )* method consists of one line: a call to the *paint()* method in *engine*. It might seem like a waste of time going from *update( )* to *paint( )* to *engine.paint( )* just to draw one image. But if there are a dozen or more movable objects on the screen at once, we can immediately find the simplicity of being able to add the object in the *init( )* method and then forget about it the rest of the time, letting the *engine.paint( )* method take care of everything.

3.5 Improvements

We can say that all the codes we have mentioned before provide the basic framework for the game programming. On the basis of that, we can make some improvements. Lets start with movable objects. When we want to write a game with a lot of movable objects, what should be considered? It would be much easier to come up with some useful properties that we want all our movable objects to have now so that we do not have to deal with each movable object individually later.

One area that merits improvement is the order in which movable objects are painted. Consider if we have a ball (represented by a movable object) that was bouncing along the screen, and we wanted it to travel in front of a person (also represented by a movable object), how could we make sure that the ball was drawn after the person every time, to make it look like the ball was in front of the person? We could make sure that the ball is the first movable object added to the engine, ensuring that it is always the last movable

object painted. However the approach can get hard if we have 10 or 20 movable objects that all have to be in a specific order. Also, what if we wanted the same ball to bounce back across the screen later on, but this time behind the person? The method of adding movable objects in the order we want them drawn obviously wouldn't work, because we would be switching the drawing order in the middle of the program.

What we need is some sort of prioritization scheme to decide which object should be drawn before other objects. So we can improve the graphics engine in the way that let it implement a scheme in which each movable object has an integer that represents its priority. The movable objects with the highest priority number are drawn last and thus appear in front. **Listing 4** shows the changes that have to be made to the *MOB* class to implement prioritization and to *GraphicsEngine* class.

Listing 4. Improved code for *MOB* class and *GraphicsEngine* class

```
import java.awt.*;
public class MOB
{
        public int x=0;
        public int y=0;
        public Image picture;
        public int priority = 0;
        public boolean visible = true;
        public MOB (Image pic)
        {
           picture = pic;
        }
}

import java.awt.*;
import java.awt.image.*;
public class GraphicsEngine
{
  Chain mobs = null;
  public Image background;
  public Image buffer;
```

```
Graphics pad;
public GraphicsEngine (Component c)
{
        buffer = c.createImage (c.size().width, c.size().height);
        pad = buffer.getGraphics ();
}
public void AddMOB (MOB new_mob)
{
  mobs = new Chain (new_mob, mobs);
}
public void paint (Graphics g, ImageObserver imob)
{
   if (background != null)
   {
      pad.drawImage(background, 0, 0, ct.getSize().width, ct.getSize().height,
                    imob);
   }
  Chain temp_mobs = new Chain(mobs.mob, null);
  Chain ordered = temp_mobs;
  Chain unordered = mobs.rest;
  MOB mob;
   while (unordered != null)
   {
     mob = unordered.mob;
     unordered = unordered.rest;
     ordered = temp_mobs;
     while (ordered != null)
      {
       if (mob.priority < ordered.mob.priority)
       {
              ordered.rest = new Chain(ordered.mob, ordered.rest);
              ordered.mob = mob;
              ordered = null;
       }
      else if (ordered.rest == null)
       {
              ordered.rest = new Chain(mob, null);
              ordered = null;
       }
       else ordered = ordered.rest;
      }
   }
    while (temp_mobs != null)
    {
      mob = temp_mobs.mob;
```

```
          if (mob.visible)
          {
              pad.drawImage(mob.picture, mob.x, mob.y, imob);
          }
        temp_mobs = temp_mobs.rest;
      }
    g.drawImage(buffer, 0, 0, imob);

  }
}

class Chain
{
  public MOB mob;
  public  Chain rest;

  public Chain(MOB mob, Chain rest)
   {
     this.mob = mob;
     this.rest = rest;
   }
}
```

The prioritization scheme does not impose any restrictions on the priority of each object. There is no need to give objects sequential priorities. We can also assign the same priority to more than one object if we do not care which object is drawn on top.

The heart of the prioritization scheme lies in the new version of *GraphicsEngine.paint( )* method. The basic idea is that before any movable objects are drawn, the complete list of movable objects has already been sorted by priority. The highest priority objects are put at the end of the list so that they are drawn last and appear in the front; the lowest priority objects are put at the beginning of the list so that they are drawn first and appear in back. From the above code we can see that a Bubble sort algorithm is used to sort the objects. Bubble sort algorithms are usually slower than other algorithms, but they tend to be easier to implement. In the case of having small number of movable objects, the extra

time taken by the bubble sort algorithm is relatively negligible because the majority of the time within the graphics engine is spent on displaying the images.

3.6 Double Buffering

When we have images loaded and doing something the next topic on the agenda is flicker. Although simplistic animations or slideshows run fairly well, as more drawing is done to the viewing window in advanced applets, one can begin to see the screen refresh causing a flicker.

We would say this is the bane of the animator or game developer because many solutions exist on a variety of platforms to speed up rendering. So far the most common solution for this problem is double buffering.

Double buffering is the process of storing a copy of the screen in a section of memory and doing all drawing to this canvas as if it were the screen. Since all drawing is being done off the screen, the only drawing onscreen is the actual copying of the buffer to the screen that can be timed to avoid the refresh. For example, in the *paint()* method of class *GraphicsEngine* in **Listing 4**, we use the variable *buffer*, which is an instance of *image* to represent the off-screen. Then before updating the screen, we use the *Graphics* context *pad*, which is gotten from *buffer* to draw all the images that consist of the next screen that will be shown immediately after. When *pad* has finished its job, we pass *buffer*, which is actually the updated screen to *Graphics* context *g*, which is gotten from the present screen to do the update. So now the updated screen is shown to users without any flicker or streaks.

Some systems or environments have automatic buffering that means little or no

programming. In Java it is really only a matter of redirection from what we've done so far.

We already know that a good thing about *GIF* files is that we can make transparent *GIFs* quite easily using many paint programs and image utilities. This fact allows gif images that are not rectangular to keep their fine shape and blend in with whatever background we like.

From the improved code in **Listing 4**, we can find two big features that are double buffering and addition of a background respectively. Notice the changes of *GraphicsEngine*. The graphics engine now creates an image so that it can do off-screen processing before it is ready to display the final image. The off-screen image is named *buffer*, and the *Graphics* context that draws into that image is named *pad*. In the *paint( )* method in *GraphicsEngine*, we can see that until the end, all the drawing is done into the *Graphics* context *pad* instead of the *Graphics* context *g*. The background is drawn into *pad* at the beginning of the *paint( )* method and then the movable objects are drawn into *pad* after they have been sorted. Once everything is drawn into *pad*, the image buffer contains exactly what we want the screen to look like, so we draw *buffer* to *g*, which causes it to be displayed on the screen.

3.7 Invisibility and Other Possible Extensions

Another feature that was added to the extended version of the movable objects was the capability to make the movable objects disappear when they are not wanted. This was accomplished by giving *MOB* a flag called *visible*. In the *GraphicsEngine.paint( )* method in **Listing 4** we can see how it works. This feature would come in handy if we have an object that we want to show only part of the time. For example, we could make a

bullet as a movable object. Before the bullet is fired, it is in a gun and should not be visible, so we set *visible* to *false* and the bullet is not shown. Once the gun is fired, the bullet can be seen, so we set *visible* to *true* and the bullet is shown.

Apart from those new features we mentioned above, we can also add some other new features like a centering feature for movable objects so that they are placed on the screen based on their center rather than their edge – this can be done by calculating the size of the movable objects to get the distance from their edges to their centers, or the addition of velocity and acceleration parameters, or even a collision-detection method that would allow us to tell when two movable objects have hit each other etc. Anyway, we can extend the codes as needed to accommodate our needs.

3.8 Sound Effect

To make our games more appealing to users, we can add sounds to the games. The basic way to make games have sound effect in Java is to get into *java.applet.AudioClip* interface. There are only three methods: *loop( )*, *play( )* and *stop( )* in this simple interface. We can use *Applet.getAudioClip( )* to load an *AudioClip* in the *AU* format and then we have two choices: Use the *play( )* method to play it at specific times or use the *loop( )* method to play it continuously. The applications for each are obvious. Use the *play( )* method for something that is going to happen once in a while, such as the firing of a gun; use the *loop( )* method for something that should be heard all the time, such as background music or the hum of a car engine.

3.9 Java-Specific Game Design Issues

There are some specific design issues we have to consider when we design our games in Java. One of Java's most appealing characteristics is that Java programs can be downloaded through the web and run inside a browser. This networking aspect brings several new considerations into play. Java is also meant to be a cross-platform language, which has important ramifications in the design of the user interface and for games that rely heavily on timing. More will be discussed in the following sections.

3.9.1 User Interface

When picking a user interface, there are several things we should keep in mind. Above all, remember that our applet should be able to work on all platforms because Java is a cross-platform language. This applet must be paid much more attention when we choose to use mouse or keyboard as our input device. Alternatively, we can choose to use either *AWT* components or *Swing* components to design and implement user interface. The AWT components are those provided by the JDK 1.0 and 1.1 platforms. Java 2 Platform still supports the AWT components. We can identify Swing components because their names start with J. The AWT button class, for example, is named Button, while the Swing button class is named JButton. Additionally, the AWT components are in the java.awt package, while the Swing components are in the javax.swing package.

The biggest difference between the AWT components and Swing components is that the Swing components are implemented with absolutely no native code. Since Swing components are not restricted to the least common denominator -- the features that are present on every platform -- they can have more functionality than AWT components. Because the Swing components have no native code, they can be shipped as an add-on to

JDK 1.1, in addition to being part of the Java 2 Platform.

Even the simplest Swing components have capabilities far beyond what the AWT components offer:

- Swing buttons and labels can display images instead of, or in addition to, text.
- We can easily add or change the borders drawn around most Swing components. For example, it's easy to put a box around the outside of a container or label.
- We can easily change the behavior or appearance of a Swing component by either invoking methods on it or creating a subclass of it.
- Swing components do not have to be rectangular. Buttons, for example, can be round.
- Assisted technologies such as screen readers can easily get information from Swing components. For example, a tool can easily get the text that is displayed on a button or label.

Swing lets us specify which look and feel our program's GUI uses. By contrast, AWT components always have the look and feel of the native platform.

Another interesting feature is that Swing components with state use models to keep the state. A JSlider, for instance, uses a BoundedRangeModel object to hold its current value and range of legal values. Models are set up automatically, so we do not have to deal with them unless we want to take advantage of the power they can give us.

If we are used to using AWT components, we need to be aware of a few things when using Swing components:

- Programs should not, as a rule, use "heavyweight" components alongside Swing components. Heavyweight components include all the ready-to-use AWT

components (such as Menu and ScrollPane) and all components that inherit from the AWT Canvas and Panel classes. This restriction exists because when Swing components (and all other "lightweight" components) overlap with heavyweight components, the heavyweight component is always painted on top.

- Swing components are not thread safe. If we modify a visible Swing component -- invoking its setText method, for example -- from anywhere but an event handler, then we need to take special steps to make the modification execute on the event-dispatching thread. This is not an issue for many Swing programs, since component-modifying code is typically in event handlers.
- The containment hierarchy for any window or applet that contains Swing components must have a Swing top-level container at the root of the hierarchy. For example, a main window should be implemented as a JFrame instance rather than as a Frame instance.
- We do not add components directly to a top-level container such as a JFrame. Instead, We add components to a container (called the *content pane*) that is itself contained by the JFrame.

The implemented game for this project has both an *AWT* version and a *Swing* version so that we can make some comparison on them and meet different needs for the users using different platforms.

If we use the keyboard as the input device when writing a game program, it is even more critical to remember that although the underlying platforms might be vastly different, Java is platform independent. This becomes a problem because the different machines that Java can run on may interpret keystrokes differently when more than one key is held

down at once. For example, we may think that it is worthwhile to throw a *supermove()* method in our game that knocks an opponent off the screen, activated by holding down four secret keys at the same time. However, doing this might destroy the platform independence of our program because some platforms may not be able to handle four keystrokes at once. The same thing is true for the mouse. For example, we might want to use two buttons in our game for the mouse but some platforms like Macintosh only has one button. The best approach is to design a user interface that does not call into question whether it is truly cross-platform. We just try to get by with only one key or one button at a time, and stay away from control and function keys in general because they can be interpreted as browser commands by different browsers in which the game applet runs.

3.9.2 Limiting Factors

As with any other programming language, Java has its advantages and its disadvantages. It is good to know both so that we can exploit the advantages and clear the disadvantages. Several issues arise when we are dealing with game design in Java – some of which are the product of the inherent design of Java and some of which are the product of the environment in which Java programs normally run.

3.9.2.1 Downloading

One of the main features of Java is that it can be downloaded and run across the net. Because automatically downloading the Java program we want to run is so central to the Java software model, the limitations imposed by using the net to get our Java bear some investigation. First we have to keep in mind that most people with a network connection

are not on the fastest lines in the world. Although we may be ready to develop the coolest animation ever for a Java game, remember that nobody will want to see it if it takes forever to download. It is a good idea to avoid extra frills when they are going to be costly in terms of downloading time.

One trick we can use to get around a lengthy download time is to download everything we can in the background. For example, we can send level one of our game for downloading, start the game, and while the user plays level one, level two and up are sent for downloading in a background thread.

Suppose we want to download a series of images and then perform a number of image enhancement operations on them. How can we be guaranteed that the image has finished loading before applying the image operations? Implementing a delay mechanism might work, but the value we use for our delay will most likely be longer or shorter than the actual time needed to completely load in the image data. We need a way to tell exactly when our images have completely loaded. Luckily this task is simplified considerably with the *java.awt.MediaTracker*.

A common concern with applets is that images are sometimes referred to before they are fully loaded, which has unpredictable results. [1] The *MediaTracker* class monitors the status of media types, such as images, sounds, etc. *MediaTracker* can load data asynchronously (in the background) or synchronously (waits for data to load first). Image loading is sped up by threads that dedicate themselves to a group of images identified by a selected ID. With the *MediaTracker*, we can track the process of the images we want to load. To create a *MediaTracker* object, simply pass a *Component* object to it in its constructor method (this will usually be a reference to our applet or frame window). The

passed *Component* represents the object onto which the images will eventually be drawn. We can attach as many images as we like to a *MediaTracker* object and then retrieve the loading status of a particular image or the status of all attached images collectively. We can also attach ID numbers to distinguish image groups from one another. Lower numbered IDs have a greater priority than higher numbered IDs. So if we wish to load a group of images that we wish to enhance, we might assign a zero value to those images. Immediately after images with priority zero finish loading, we can perform our image enhancements while images with a lower priority continue to load.

An alternative way is store all our images, classes and other files in a JAR (JAR stands for Java Archive. It's a file format based on the popular ZIP file format and is used for aggregating many files into one.) file and load them at one time to reduce the number of connections of network.

Opening a network connection can take a significant amount of time. If we have 30 or 40 pictures to send for downloading, the time this takes can quickly add up. One trick that can help decrease the number of network connections we have to open is to combine several small pictures into one big picture. We can use a paint program or an image-editing program to create a large image that is made up of our small images placed side by side. We then send for downloading only the large image. This approach decreases the number of network connections we need to open and can also decrease the total number of bytes contained in the image data. Depending on the type of compression used, if the smaller images that make up our large image are similar, we will probably achieve better compression by combing them into one picture. Once the large picture has been loaded from across the network, the smaller pictures can be extracted using the

java.awt.image.CropImageFilter class to crop the image for each of the original smaller images.

3.9.2.2 Execution Speed

Timing with applets is another important thing we have to keep in mind. Java is remarkably fast for an interpreted language, but graphics handling usually leaves something to be desired when it comes to rendering speed. Our applet probably will be rendered inside a browser, which slows it down even more since users have to load some other things other than the images. We may be developing our applets on a state-of-the-art workstation, but a large number of people will be running them inside a web browser on much slower PCs. When our applets are graphics intensive, it is always a good idea to test them on slower machines to make sure that the performance is acceptable. If we find that an unacceptable drop in performance occurs when we switch to a slower platform, we can try shrinking the *Component* that our graphics engine draws into. We may also want to try shrinking the images used inside our movable objects because the difference in rendering time is most likely the cause of the drop in performance.

Another thing to watch out for is poor thread handling. A top-of-the-line workstation may allow us to push our threads to the limit, but on a slow PC, computation time is often far too precious. Improperly handled threading can lead to some bewildering results. For example, we often make some threads sleep for a certain time to yield the processor to other threads, if these threads don't co-operate well, then one threads may use all the processor's time to lock up the running of other threads.

Admittedly, compared to natively compiled programs written in languages like C/C++ or

Pascal, Java programs are slow in some fields. However, this complaint has to be weighed heavily against the inherently cross-platform nature of Java, which simply is not possible with native programs such as those generated by C/C++ and Pascal compilers. In an attempt to alleviate the inherent performance problems associated with processor-independent Java executables, various companies are offering just-in-time (JIT) Java compilers, which compile Java bytecode executables into native programs just before execution. As we know that, when a Java program is compiled, it is compiled to be executed under the Java Virtual Machine (VM). But the fact is that VM itself is highly platform dependent. In other words, each different hardware platform or operating system must have a unique VM implementation that routes the generic VM calls to appropriate underlying native services. JIT compilers alter the role of the VM a little by directly compiling Java bytecode into native platform code, thereby relieving the VM of its need to manually call underlying native system services. A basic JVM executes a small subroutine for each Java bytecode. A JIT compiler takes the bytecode and translates them into native machine code with respecting system calls so as not to violate security. JIT compiles only bytecodes that are actually executed, whereas conventional compilers must compile all of the source code. By compiling bytecode into native code, execution speed can be greatly improved because the native code can be executed directly on the underlying platform. This stands in sharp construct to the VM's approach of interpreting bytecodes and manually making calls to the underlying platform. In this way, the Java programs can be run almost as fast as compiled languages such as C and C++.

Chapter 4: Case study -- *CrazyRoad1.0* (Online Game)

This chapter presents a summary of the Java game applet design and implementation by Ye Zhu [13] to evaluate the game environment presented in the previous chapter.

4.1 Brief Description

CrazyRoad1.0 (hereafter called *CrazyRoad*) is a game, which is designed and implemented with Java as a Java applet. Users can download it from web site and run it on their local computer using a web browser. It is designed and implemented using all the technologies we have mentioned before in this report with making some changes and adding some new features. When we get into the game and ready to run it, it looks like this:

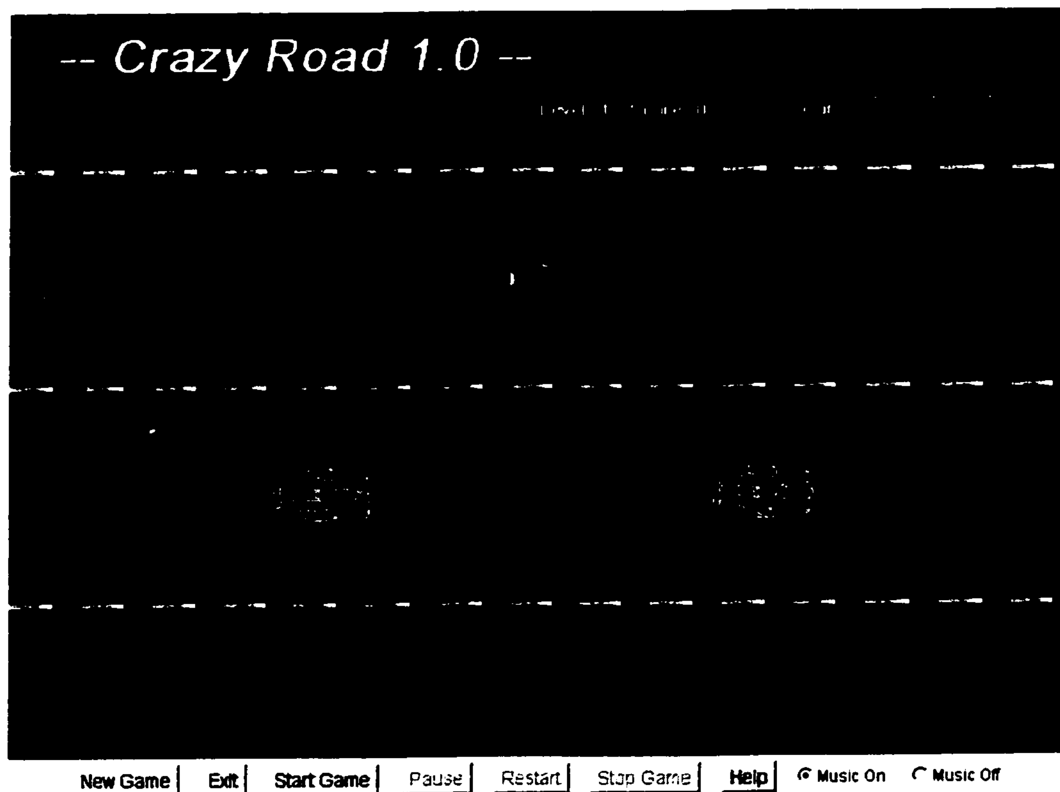

Figure 5. The interface of the implemented game – CrazyRoad1.0

The game is quite simple. From the above figure we can see that there are three lines on the screen. These are the driving lines on the road. The road is separated into two ways by the middle driving line. A car with red color and its head facing left is the playing car (driven by the user) whose moving direction is to the left side in this game. The cars with yellow color and their heads facing right are the obstacle cars whose moving directions are to the right side. User just uses the space bar on the keyboard to control the position of the playing car to avoid hitting by the obstacle cars whose moving directions are conflict with that of the playing car. This program is designed in the way so that there are only two possible positions for the playing car, which are up driving way or the down driving way. At the beginning of the game, user is provided one playing car on the road plus three backup playing cars in the garage shown on the right top of the screen. Once an obstacle car hits the playing car, the number of the backup playing cars will be reduced by one until no backup cars are stored in the garage. If the playing car on the road collides with an obstacle car and at this time there are no more backup playing cars stored in the garage, the game is over and the applet stops running. On the top of the screen, user can check the message related to the level of the game user is reaching and the score user has gained. At the bottom of the screen, there are some buttons from which user can control the game or do some selections or get help. There are also some short-cut keys that make user easy to control the game.

4.2 Design

The small example we mentioned before only has three classes, which are *MOB*, *GraphicsEngine* and *Game*. While in this project, from the above figure we can see that

this game is made up of two areas: one is the playing area, and the other one is the control area, which is the graphic user interface.

Firstly we should make sure that three classes must be involved in this program. *MOB* class is required since every movable object has to be represented by it. *GraphicsEngine* is required since we have to have such an engine to control the drawing of all the movable objects. *Game* class is also required since it is the entry point of program and also it is the place we have to find to install the graphics engine.

But one thing, which is different from the above small example we should notice here, is that this game has a graphic user interface. As the entry point of this game we will design is an Applet, if we make the applet handle both the image drawing and the user interface, it will be hard to implement and easy to cause problems. So we add a new class here to be in charge of taking care of the drawing of the playing area, and it is also responsible for the installation of the graphics engine. To reach this goal, the new added class must have the feature of being drawn images on it. So making this new added class as a subclass of *Panel* (A component of AWT: It is a container that can contain other components.) is an obvious way. Now what we just need to do is to pack the playing area and user interface area into the entry point class. Thus the image processing and the user interface are being taken care of separately. This makes the program easy to be implemented and extended.

Finally, there are totally four classes making up of this program. Their names and main functionalities are as follows:

MOB class:

MOB class is just the same as the extended *MOB* class in **Listing 4**. Its instance represents a movable object. It incorporates the priority scheme to ensure that every

movable object has a priority number. The priority of every movable object is initialized with the value of 0.

GameEngine class:

Instead of being called *GraphicsEngine*, We give the graphics engine a new name – *GameEngine* for easy understanding. This engine is used to generate the movable objects list and to take care of the drawing of all the movable objects. This class uses the basic framework of class *GraphicsEngine* as in Listing 4 some changes. Some information about the game such as the name of the game, the level of the game and the scoring of the game etc. are controlled and displayed by this class. From the above figure, we can see that there are several objects on the screen. These are the potential candidates that consist of the movable objects list. Here we take the background, the playing car, the obstacle cars, the backup playing car and the side bar, which consists of the driving ways, all as the movable objects.

CrazyRoad class:

This is the new added class. It extends *Panel* class and implements *Runnable* interface. The reason why we make it implement *Runnable* class is that any instances of this class will be run as a *thread.*

This class is responsible for the installation of the *GameEngine*, and, actually the control of the drawing of all the movable objects is achieved here. The main functionalities of this game such as scoring, hitting, sound effect etc. are all implemented in this class.

Game class:

This is the entry point of this game. It is designed as an applet, which means this game will be played using a web browser.

This class is also responsible for the packing of all the parts consisting of the game, like the playing area and the user interface. Actually the user interface is controlled here. After the compilation, the compiled file should be integrated into a HTML file in order to make the user download it from some server and run it on the local computer.

The class diagram can be shown as follows:

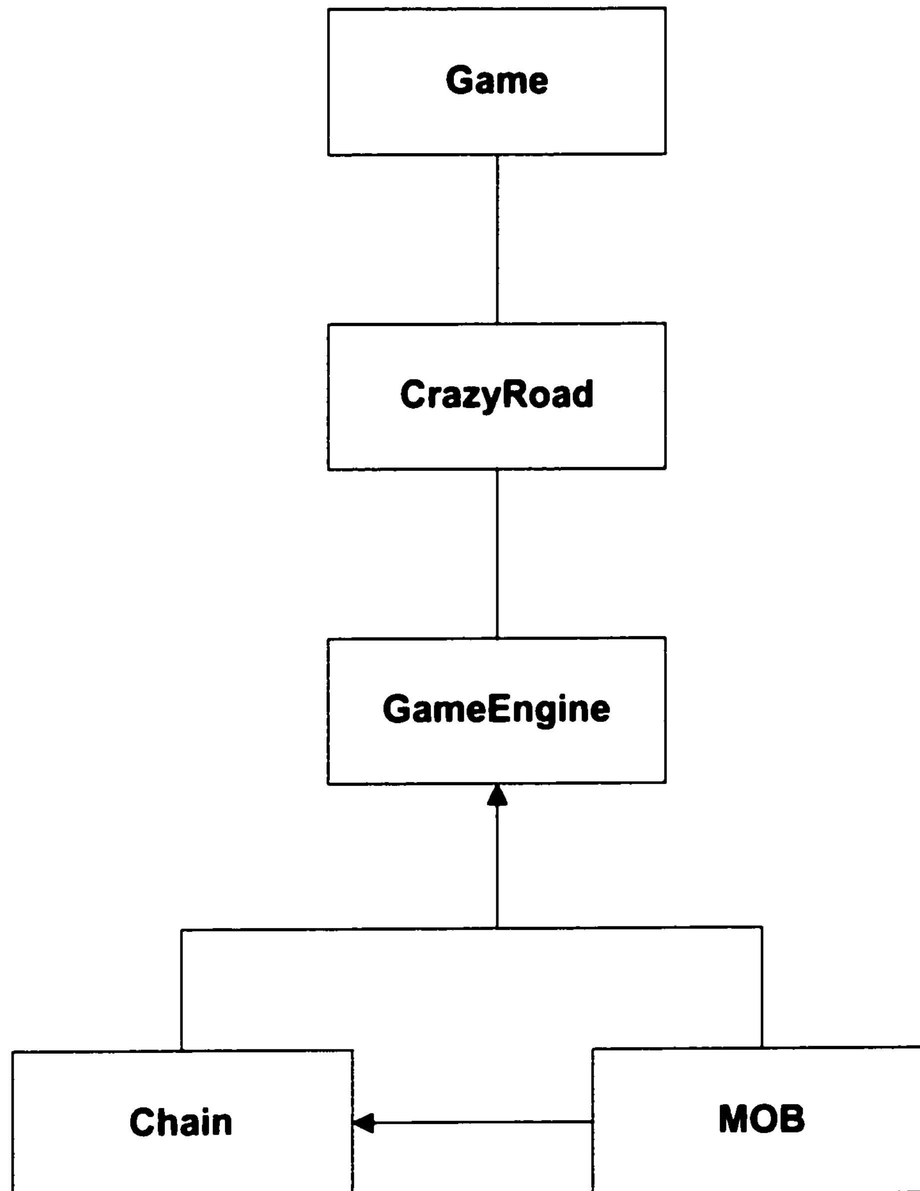

Figure 6. Class diagram for the implemented game – CrazyRoad1.0

To see more details of design and the detail of the implementation, please refer to the collaborated work [13].

Chapter 5: Improvement and fallacies on Java

During the course of this collaborated project, I and Ye Zhu[13] made a few abstractions on Java programming in general and Java game programming in particular. This chapter presents a summary of these abstractions.

5.1 Some suggestions for Java game programming

In our programming endeavors using Java, we will come across instances where the code we write can either optimize or hinder the output of our programs. The following are a few brief points we should consider when writing Java code. Some of them detail practices we should definitely incorporate into our code; others are things we should definitely avoid. More details related to this topic, please refer to the collaborated work of [13].

- **Do code abstractly, but not too abstractly**

The principle of abstraction is one of the cornerstones of object-oriented programming. Since Java is nearly 100 percent object-oriented, knowing when to abstract become even more important. For example, in our game we separate the classes according to different functionalities each of which will perform so that we can generate objects with different features freely and also can extend these objects freely. Thus we can notice that the structure of our game is very clear. But we should remember to use abstraction only when it will benefit or clarify our code.

- **Do not use Java reflection classes**

The *java.lang.reflect* package contains classes that allow Java classes to obtain information about them. Use of this package includes determining the run-time name of

an object, along with any methods it contains. These classes are commonly used in the Java Virtual Machine itself, as well as external debuggers and profilers. If we come from a C++ background, we have probably used function pointers in our code. Since Java does not contain support for function (or method) pointers, it might seem temping to send a *Method* object as a parameter to a function to simulate this functionality. Firstly, programming with these classes can really reduce the throughput of our applications since some of these classes such as *Method* are quite inconvenient, error prone and slow; and as we know, speed is a critical component to any game.

- **Do try to incorporate code that optimizes both speed and size of the programs**

This includes incorporating things such as image strips into the projects, as well as eliminating the use of synchronization (since without cautious design and programming, synchronization is error and deadlock prone) and exception handling when possible. Do not, however, blindly eliminate all of the exception-handling code; if catching an exception can save the program from completely crashing, by all means use it.

Excessive use of threading can trouble the programs as well. Using a few threads in the games is usually necessary, but don't make the program have hundreds of objects, and each of them use their own threads may cause the machine to spend all of its time switching between threads, which will obviously slow things down way too much.

- **Don't use Swing classes for games**

Thought as we discussed before, Swing has more advantages than AWT, and programmatically Java Swing is a very clean and robust way to create applications, during the practice we found that Swing is too bloated for game development. If we use Swing just for the GUI in common application instead of game development, Swing is

really the preferable choice than AWT. But for game programming, the benefits that Swing provides do not outweigh the speed hit that it makes on applications.

- **Do think about minor code optimizations, but not too much**

Techniques such as loop unrolling and using register variables (not in Java) used to be very common among game programmers. However, with advances in processor speed, such as seen with a lot of today's popular games, minor optimizations have become less important.

- **Don't use sun.* packages**

Java packages such as those contained under java.*, javax.* and org.* are standard and are supported across all platforms, but classes under the sun.* package are not. Classes under sun.* packages are generally platform-specific and are known to change from version to version.

- **Don't calculate values more than once if possible**

It goes without saying that all program calculation take a finite processor time to execute. Therefore, it is common sense for programmers that the fewer calculations the game code makes, the faster it will run.

- **Don't attempt to optimize standard Java**

Although one can sometimes find ways to rewrite standard C functions to perform faster or better, we would not try to improve upon any standard Java class or method. Even if it looks like making an optimization to native Java code would be appropriate, it might not be due to considerations such as Java Virtual Machine issues or platform compatibility.

5.2 Some fallacies on Java

With the occurrence of Java and its development, people usually like to compare it with some other popular programming languages like C, C++. We can find this kind of comparison in a lot of books related to Java technology. We found that there are some fallacies in these comparisons that may mislead readers. The purpose of this section is to explain some commonly held misbelieves on Java.

Fallacy 1: C is compiled; Java is interpreted.

This is wrong. Both C and (modern) Java VM's compile code. C compiles at build-time, while Java VM compiles at run-time.

C/C++ compile to native machine code that executes only on the target machine. Java compiles to bytecodes that run on any machine that has a JVM. Some advanced JVMs further compile bytecodes to native machine code for the platform on which they are running. As we have mentioned before, JIT is a technique that compiles during execution (at run time) and consequently first slows down and then speeds up execution.

Fallacy 2: Build-time compilers produce faster programs.

This is wrong. Because:

- Run-time compilers can optimize somewhat better than build time compilers.[3]
- Run-time compilers know how one is using the code. [3]
- Run-time compilers can perform "dangerous" optimizations because they can recover. [3]

Fallacy 3: C is faster than Java.

This is wrong. As we know, C is really faster at something, for example like array access.

But Java is also faster at some things such as:

- Java has much better memory allocation that can result in speeding up execution.
- Java has better recursion and in lining. For example, standard *fft* in Java is almost twice as fast as *MSVC* and *gcc*. [3]
- Over-all performance of C and Java is roughly equivalent today. [3]

Fallacy 4: Java programs port themselves.

This is wrong since:

- Pure Java programs will run on any Java VM, but will need to be qualified for system specific behaviors and performance characteristics.
- 85% - 90% of a game will likely port itself, the critical sections for multithreading will need to be tuned. [3]

5.3 Java's improvements over C and C++

C is a very popular systems programming language. It is concise, efficient and provides close control over hardware. The language has several idiomatic expressions (such as while(*p++ = *q++)) that at first appear very obscure; their effective use gives expert C programmers a deep sense of mastery. C programs, however, are notoriously unreliable. Indeed, the C idiom shown above will generally leave both pointers, p and q, pointing outside the original data structures, and any use of their values will then lead to run-time errors. In general C allows pointers to point to anywhere, and it provides type casts so that whatever a pointer points at can be converted into something that can be manipulated. In particular, C has an operator (&) that allows a program to obtain pointers

to *anything*. Such features are extremely useful for directly controlling hardware -- say, to send bytes to a peripheral -- but unfortunately the same features *ensure* a C compiler cannot provide any protection, as when a program makes accidental changes to itself in arbitrary places.

Java overcomes these problems by two restrictions. Java simply has no pointers, so it is not possible to access arbitrary parts of a program or to get 'inside' private data structures, whose actual structure may not quite be what the programmer planned. Secondly, Java has much stricter type checking. For example, it is not possible to change the exponent of a floating point number by treating it as an array of bytes -- whether deliberately or accidentally.

Both C and Java have arrays. In C, arrays are defined to be equivalent to pointer expressions, so they have exactly the same liabilities pointers do, perhaps with the added danger that they look innocuous. The legal C assignment a[5]=6 looks perfectly reasonable, but if a is a pointer to an array declared to have only four elements, then the result of the assignment would be undefined. In fact, a is a pointer and could be pointing *anywhere* (say, to a C function) when the attempted assignment occurs! Quite possibly the assignment would change the C program's return stack, and cause the current function to return to an arbitrary point in the program, and then anything could happen. In Java, however, the subscript range is part of the array's type, and attempting to access an element outside the appropriate range is an error that is always detected. In other words, the common traps of C have been converted to explicit barriers in Java.

Despite these improvements, Java nevertheless remains very close to C. A simple C program can be converted to Java more-or-less by just saying it is a class and fixing the

errors the Java compiler reports. Doing so will, in many cases, produce a much more reliable program.

C++ is an object-oriented extension to C, and while introducing object oriented concepts, it retains all the features of C that make it both efficient and unreliable, as described above. C++ is a complex language, and Java effectively uses the 'core' object oriented features without many of C++'s complications, such as multiple inheritances. C++'s design principles were not to create reliable or portable programs, instead it was designed to be useful and enjoyable for serious programmers. C++ certainly succeeded, and Java builds on that enthusiasm -- trading the fun of C++ with the fun of the Internet and the opportunity to run Java programs world widely.

But at the same time we should also notice that Java still has some unfortunate and avoidable weaknesses, these reside in its notation, its design issues, some of its features like Strings and Arrays and some of its unnecessary confusion etc. For details of these issues please see [14]. While generally we have to say that lots of Java features are still under testing and discussing, it changes too fast so that those new versions or features need time to be tested. But those features we have mentioned so far in this report are really catching more and more developers' attention. Here are some comments from some software developers at what they think of Java:

Java programs are faster to write and less buggy.

Most of C/C++ engineers who have moved to Java have reported a 2 to 10 times productivity increase. [3]

The design of Java prevents whole classes of "late detection" bugs:

- No un-initialized variables
- No wild pointers
- No array over-runs or under-runs

Chapter 6: Summary

In this report, we introduced the basic idea of the game programming with Java as well as briefly mentioned the history and technology of Java. We developed a basic graphics engine that can be used for game creation. This graphics engine incorporated movable objects with prioritization and visibility setting, double buffering and a background. This is a good framework for Java game programming, even with other programming languages since the principles for the game programming with different languages are quite similar. This report also covers some of the design of a case study for the implemented game whose details can be found in the collaborated work of **Ye Zhu**[13]. Here we can see that the design is on basis of the basic framework with some improvement and extension. However the focus was on the construction of the tools rather than the construction of the implemented game because the tools can be expanded to produce a multitude of games.

This report also touched on issues we should keep in mind when developing games with Java. It is important to remember that Java is a cross-platform language and therefore runs on different platforms. When we develop our games, we should be aware that people will want to run them on machines that may not be the same as our machines.

Some common suggestions on Java programming are also raised at the end of this report. These are a few points we got during our practices. Actually there are hundreds of more common suggestions that are applicable to Java programming. However for Java game programming, those mentioned here are some key points to improve upon the code so that the games will perform as quickly and efficiently as possible. At the end of this report, we discussed some fallacies on Java in order to make the facts clear.

In general, as a new object-oriented programming language after C++, Java has the necessary features for game programming. Being simple to use, fast improvement, becoming more powerful than before and being used in more and more fields, are the main reasons that made me to do some research on it. While we are happy to see those powerful features of Java and using it more and more in so many fields, we should also notice those avoidable weaknesses, confusing features and unnecessary inconvenience etc. People are waiting for some revolution on modern computer science. When Java came to the world, some people said the revolution is Java. But these days, few people say so. Java did bring lot of new concepts and surprises to developers and programmers, but it did not bring modern computer science with the kind of expected influence that a revolution will do. Perhaps Harold Thimbleby's comments on Java will make us look at Java from a different point of view "It *looks* simple yet is complicated enough to conceal *obvious* deficiencies".[14]

References:

[1]. Premier Press, Thomas Petchel, Java 2 game Programming

[2]. Java Unleashed, Second Edition http://www.informit.com

[3]. Java 2001, The year of games. Jeff kesselman, Staff Engineer, Sun Microsystems, VGEE specialist, co-Author Java platform performance

[4]. Java 1.2 Unleashed, http://www.Sams.net

[5]. SAMS, Steve Potts, The Waite Group's Java 1.2 How – To

[6]. SUN Microsystems Press, Peter van der Linden, Just Java 2 Fourth Edition

[7]. SUN Microsystems Press, Gay S. Horstmann, Gary Cornell, Core Java 2

[8]. SUN Microsystems Press, Gay S. Horstmann, Gary Cornell, Core Java 2 Volume II

[9]. SUN Microsystems Press, David M. Geary, Graphic Java 1.2 Mastering the JFC Volume I: AWT (3rd edition)

[10]. SUN Microsystems Press, David M. Geary, Graphic Java 2 Mastering the JFC Volume II: Swing (3rd edition)

[11]. Paul Hudson, "Technology Overview" http://www.herts.ac.uk/ltdu/technology/

[12]. Sun Microsystems, Inc., "The Java Technology Phenomenon" http://java.sun.com/docs/books/tutorial/getStarted/intro/

[13]. Game programming with Java : a case study , major report, Concordia University 2002, Ye Zhu

[14]. Harold Thimbleby, "A critique of Java" http://www.cs.mdx.ac.uk/harold/srf/javaspae.html