Design and Implementation
# of a Java Game Applet

Ye Zhu

A Major Report

in

The Department

of

Computer Science

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

July 2002

0-612-72953-2

ABSTRACT

This report is an exploration of basic principles of game programming with Java as well as some Java technologies. In this report, a basic framework for game programming is briefly discussed and provided for programming Java games. On the base of the basic principles for Java game programming, an online game called *CrazyRoad1.0* is designed and implemented. As this is a project jointly with **Ying Dong**, in this report the principles related to game programming are only briefly described.[1] Instead, the detail of the design and implementation of the game *CrazyRoad1.0* will be mainly covered in this report. Some common suggestions on writing Java programs are also provided for making fast and efficient Java games.

[1] For the detail of design and evaluation of Java game programming environment, please refer to the collaborated work Ying Dong's report [15].

Table of Contents

List of Figures

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]. It 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, there is still plenty of room for people to make their marks on the gaming industry. According to [1], Java games often appeal to the 85% of computer users who are considered the real gamers, 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 principle to use Java language to create robust, flexible gaming applications. It covers the fundamental ideas of creating games using Java with a detailed description of a case study (an online game implementation – CrazyRoad1.0). This report also describes the work carried out in collaboration with Ying Dong [15].

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

2.1 A Brief History of Java

At first glance, it may appear that Java was developed specifically for the Web. However, interestingly enough, Java was developed independently of the Web, and went through several stages of metamorphosis before reaching its current status for the Web. Below is a brief history of Java from its infancy to its current state.

Oak

According the Java FAQ [12], Bill Joy, currently a vice president at Sun Microsystems, is widely believed to have conceived the idea of a programming language that later became Java. In late 1970's, Joy wanted to design a language that combined the best features of MESA and C. In an attempt to rewrite the UNIX operating system in 1980's, Joy decided that C++ was inadequate for the job. A better tool was needed to write short and effective programs. It was this desire to invent a better programming tool that swayed Joy, in 1991, in the direction of Sun's "Stealth Project" as named by Scott McNealy, currently Sun's CEO [12].

In January of 1991, Bill Joy, James Gosling, Mike Sheradin, Patrick Naughton (formerly the project leader of Sun's OpenWindows user environment), and several other individuals met in Aspen, Colorado, for the first time to discuss the ideas for the Stealth Project. The goal of the Stealth Project was to do research in the area of application of computers in the consumer electronics market. The vision of the project was to develop "smart" consumer electronic devices that could all be centrally controlled and programmed from a handheld-remote-control-like device. According to Gosling, "the goal was ... to build a system that would let us do a large, distributed, heterogeneous network of consumer electronic devices all talking to each other." With this goal in mind, the Stealth group began work [13].

Members of the Stealth Project, which later became known as the Green Project, divided the tasks among themselves. Mike Sheradin focused on business development, Patrick Naughton began work on the graphics system, and James Gosling identified the proper programming language for the project. Gosling, who had joined Sun in 1984, had previously developed the commercially unsuccessful NeWS windowing system as well as GOSMACS - a C language implementation of GNU EMACS. He began with C++, but soon afterward was convinced that C++ was inadequate for this particular project. His extensions and modifications to C++ (also known as C++ ++ --) were the first steps toward the development of an independent language that would fit the project objectives. He named the language "Oak" while staring at an oak tree outside his office window. The name "Oak" was later dismissed when to a patent search determined that the name was copyrighted and used for another programming language. According to Gosling, "the Java development team discovered that Oak was the name of a programming language that predated Sun's language, so another name had to be chosen" [13, 12].

"It's surprisingly difficult to find a good name for a programming language, as the team discovered after many hours of brainstorming. Finally inspiration struck one day during a trip to the local coffee shop," Gosling recalls. Others have speculated that the name Java came from several individuals involved in the project: James gosling, Arthur Van hoff, Andy bechtolsheim [14].

There were several criteria that Oak had to meet in order to satisfy the project objective. Given the wide array of manufacturers in the consumers electronics target market, Oak would have to be completely platform independent and function seamlessly regardless of the type of CPU in the device. For this reason, Oak was designed to be an interpreted language, since it would be practically impossible for a complied version to run on all available platforms. To facilitate the job of the interpreter, Oak was to be compiled to an intermediate "byte-code" format that then would be passed around across the network and executed/interpreted dynamically [13, 12, 14].

Additionally, reliability was of great concern. A consumer electronics device that would have to be "rebooted" periodically was not acceptable. Another important design objective for Oak would then have to be achieved by minimizing programmer-introduced errors. This was the motivation for several important modifications to C++. The concepts of multiple-inheritance and operator overloading were identified as sources of potential errors and were eliminated in Oak. Furthermore, in contrast to C++, Oak included implicit garbage collection, thereby providing efficient memory utilization and higher reliability. Finally, Oak attempted to eliminate all unsafe constructs used in C and C++ by only providing data structures within objects [13, 14].

Another essential design criterion was security. By design, Oak-based devices were to function in a network and often exchange code and information. Inherently, security is of great concern in a networked environment, especially in an environment as network-dependent as the conceived Oak-based systems. For this reason, pointers were excluded from the design of Oak. This would theoretically eliminate the possibility of malicious programs accessing arbitrary addresses in memory [13, 14].

If Oak was going to be widely accepted and used within the consumer electronics industry, it would have to be simple and compact, so that the language could be mastered relatively easily and development would not be excessively complex. Some would argue that Oak/Java is C++ done right, but the jury is still out on that.

In April of 1991, Ed Frank, a SPARCstation 10 architect, joined the Green Project. He led the project's hardware-development effort. In two months they developed the first hardware prototype. known as star-seven (*7). The name *7 was somewhat demonstrative of the project's objective. *7 was the key combination to press on any telephone to answer any other ringing telephone on their network. In the meantime, Gosling was beginning work on the Oak interpreter. By August of 1991, the team had a working prototype of the user interface and graphical system, which was demonstrated to Sun's co-founders Scott McNealy and Bill Joy [13].

Development of Oak, the Green OS, the user interface, and the hardware continued through the summer of 1992. In September of that year, the *7 prototype was complete and demonstrated to McNealy and Joy. The prototype was a PDA-like (personal digital assistant) device that Gosling described as a "handheld remote control." Patrick Naughton proclaimed that "in 18 months, we did the equivalent of what 75-people organizations at Sun took three years to do an operating system, a language, a toolkit, an interface, a new hardware platform" [13].

While impressive, this type of technology lacked market appeal, as later demonstrated by Apple's Newton PDA. The Green Project's business planner, Mike Sheradin, and hardware designer, Ed Frank, had envisioned a technology similar to that of Dolby Labs, which would become the standard for consumer electronics products [13].

FirstPerson

In November of 1992, the Green Project was incorporated under the name FirstPerson. Given Java's lack of success in the consumer electronics industry, the company's direction was somewhat uncertain. Under Sun's influence, the company began reevaluating its mission.

In early 1993, Time-Warner issued an RFP (request for proposal) for a set-top box operating system and interactive, video-on-demand technology. FirstPerson identified this area as a new target market and began working in that direction. However, despite FirstPerson's great efforts, SGI was granted the contract by Time-Warner. By mid 1993 Sun began negotiating with 3DO to provide a Java-based OS for their set-top box. The negotiations were, however, unsuccessful and a deal was never made. FirstPerson was left on its own without any viable business prospects. Another attempt by the company to market its interactive TV technology failed when in February of 1994 a public launching of their products was canceled [13].

A higher-level review of FirstPerson determined the interactive TV market to be immature in 1994. FirstPerson then shifted its focus yet again. Business plans were submitted to Sun's executives for developing Oak-based on-line and CD-ROM applications. Sun's response was not favorable, and FirstPerson was dissolved. Most of FirstPerson's employees moved to Sun Interactive to work on digital video data servers. However, a few individuals from FirstPerson still pursued the objective of finding a home for Java in a networked desktop market [13].

Java and the Web

In June of 1994, Bill Joy started the "Liveoak" project with the stated objective of building a "big small operating" system. In July of 1994, the project "clicked" into place. Naughton got the idea of putting "Liveoak" to work on the Internet while he was playing with writing a Web browser over a long weekend. Just the kind of thing we'd want to do with our weekend! This was the turning point for Java [13].

The Web, by nature, had requirements such as reliability, security, and platform independence which were fully compatible with Java's design parameters. A perfect match had been found. By September of 1994, Naughton and Jonathan Payne (a Sun engineer) start writing "WebRunner," a Java-based Web browser which was later renamed "HotJava." By October 1994, HotJava was stable and was demonstrated to Sun executives. This time, Java's potential, in the context of the Web, was recognized and the project was supported. Although designed with a different objective in mind, Java found a perfect match in the Web. Introduction of Java marked a new era in the history of the Web. Information providers were now given the capability to deliver not only raw data, but also the applications that would operate on the data.

Sun formally announced Java and HotJava at SunWorld '95. Soon afterward, Netscape Inc. announced that it would incorporate Java support in their browser. This was a great triumph for Java since it is now supported by the most popular browsers in the world. Later, Microsoft also announced that they would support Java in their Internet Explorer Web browser, further solidifying Java's role in the Web.

2.2 Java language

According to the white paper of Java from Sun, Java language has the following properties:

Simple, Object Oriented, and Familiar

The Java programming language is designed to be *object oriented* from the ground up. Object technology has finally found its way into the programming mainstream after a gestation period of thirty years. The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. Java technology provides an object-based development platform.

Programmers using the Java programming language can access existing libraries of tested objects that provide functionality ranging from basic data types through I/O and network interfaces to graphical user interface toolkits. These libraries can be extended to provide new behavior.

Keeping the Java programming language looking like C++ results in it being a *familiar* language, while removing some unnecessary complexities of C++. Having the Java programming language retain many of the object-oriented features and the "look and feel" of C++ means that programmers can migrate easily to the Java platform.

Robust and Secure

Java provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmers towards reliable programming habits.

The memory management model works in this way: objects are created with a new operator. There are no explicit programmer-defined pointer data types, no pointer arithmetic, and automatic garbage collection. This simple memory management model eliminates entire classes of programming errors.

Java technology is designed to operate in distributed environments, which means that *security* is of paramount importance. With security features designed into the language and run-time system, Java technology lets user construct applications that can't be invaded from outside. In the network environment, applications written in the Java programming language are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

Architecture Neutral and Portable

Java technology is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces. To accommodate the diversity of operating environments, the Java Compiler^{TM} product generates *bytecodes*--an *architecture neutral* intermediate format designed to transport code to multiple hardware and software platforms. The interpreted nature of Java technology makes the same Java programming language byte codes possibly run on any platform.

The architecture-neutral and portable language platform of Java technology is known as the *Java virtual machine*. It's the specification of an abstract machine for which Java programming language compilers can generate code. Specific implementations of the Java virtual machine for specific hardware and software platforms then provide the concrete realization of the virtual machine. The Java virtual machine is based primarily on the POSIX interface specification--an industry-standard definition of a portable system interface.

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.

Interpreted, Threaded, and Dynamic

The *Java interpreter* can execute Java bytecodes directly on any machine to which the interpreter and run-time system have been ported.

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, 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 in its compile-time static checking, the language and run-time system are *dynamic* in their linking stages. Classes are linked as needed. New code modules can be linked in on demand from a variety of sources. 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.

2.3 Java platform

A *platform* is the hardware and software environment in which a program runs. There are many popular platforms in the world 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's 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)

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Pascal, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called compiler. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, we can use an interpreter, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: they can let us use a machine-language program meant for one type of computer on a completely different type of computer. For example, there is a program called "Virtual PC" that runs on Macintosh computers. Virtual PC is an interpreter that executes machine-language programs written for IBM-PC-clone computers. If we run Virtual PC on our Macintosh, we can run any PC program, including programs written for Windows. (Unfortunately, a PC program will run much more slowly than it would on an actual IBM clone. The problem is that Virtual PC executes several Macintosh machine-language instructions for each PC machine-language instruction in the program it is interpreting. Compiled programs are inherently faster than interpreted programs.)

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java virtual machine. The machine language for the Java virtual machine is called Java bytecode. There is no reason why Java bytecode could not be used as the machine language of a real computer, rather than a virtual computer. In fact, Sun Microsystems -- the originators of Java -- have developed CPU's that run Java bytecode as their machine language.

However, one of the main selling points of Java is that it can be used on **any** computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the Java virtual machine in the same way that Virtual PC simulates a PC computer.

Of course, a different Jave bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program. And the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.

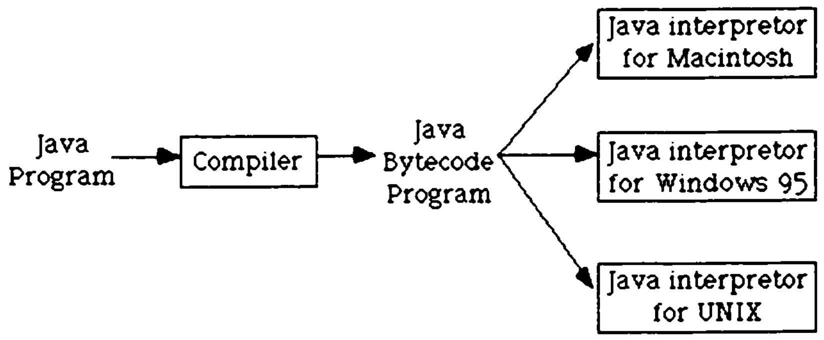
Java Program
Compiler
Java Bytecode Program
Java interpretor for Macintosh
Java interpretor for Windows 95
Java interpretor for UNIX

Figure 1. Java virtual machine

Why use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer he/she wants to run it on? There are many reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a fairly small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It vould be much harder to write a Java compiler for the same computer.

Furthermore, many Java programs can be downloaded over a network. This leads to obvious security concerns: we don't want to download and run a program that will damage our computer or our files. The bytecode interpreter acts as a buffer between us and the program we download. We are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect us from potentially dangerous actions on the part of that program.

There is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages could be compiled into Java bytecode. However, it is the combination of Java and Java bytecode that is platform-independent, secure, and network-compatible while allowing us to program in a modern high-level object-oriented language.

2.4 Can Java be used to write Games?

Game programming usually 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. So if a language has many built-in constructs (these constructs can deal with the calculations, image processing or other computing which are some of the critical required functionalities for game programming), then such kind of language can allow newcomers to the field of game programming to feel comfortable more quickly, allowing them to break down projects into small, manageable steps. Some languages that are popular for games like C/C++ all have rich feature set and powerful libraries to deal with those required tasks for game programming. Java, like C/C++, also has a large set of packages (libraries) that can deal with various kinds of calculation, computing and image processing. Its 2D and 3D libraries improves its ability to deal with graphics. And also Java is completely object-oriented. So we can say that Java has the necessary features for game programming.

2.5 Why choose this topic as a graduate level work?

Java first caught my eyes due to its simple and easy to use API. Because of my C/C++ background and the similarity of look and feel of Java with C++, it took me little time to learn to use Java in the beginning. Later I also found that Java is easy to use for building GUI for applications and writing small applications (applets) for the web. This began to catch my attention more and made me change my interest from C/C++ to Java. These years Java has improved and spread very fast that nowadays more and more software developer 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 concepts, new thinking and new technologies. Some Java technology that are presently popular like J2EE, JMS and CORBA with Java even make me feel quite necessary to do some research on Java. And, 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 a java game applet is designed and implemented using the Java game programming environment designed by Ying Dong [15] in a collaborated work.

Chapter 3: Java Game Programming

This chapter present a summary of the Java game programming environment designed by Ying Dong [15]. The design and implementation of the game applet using this environment, which is the main focus of this report, will be presented in the next chapter.

Creating games with Java is a lot like creating games with other languages. There are usually some basic components that consist of 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. 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, we need to deal with the design of a graphics engine to keep track of movable objects, and we also need to deal with double buffering to make the movement look smooth.

Java has the ability to take 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. And, the easy implementation of multiple threads makes Java a better way for game application.

From the above brief description we can know that for a game programming using object-oriented language such as Java, we need a class to represent the movable objects, since almost all games consist of a lot of movable objects; and we also need a class to represent the graphics engine since those movable objects need to be controlled by some mechanisms. We should not put these two classes into a single class since they play different roles in a game. Additionally we also need a place (or component) to install the graphics engine, just like installing an engine into a car. This place (or component) is usually where those movable objects will be drawn on. In general, at least three classes are required to program a game in Java. One is used to represent a movable object; one is used as graphics engine to control all the movable objects; and the last one is used to install the graphics engine and usually is the entry point of the program to make the game run. These three classes form the minimum framework for Java game programming, or we can even say for the game programming with any other language. There might be some alternatives, but eventually we will find that their basic ideas are quite similar.

The details referring to the principles of Java game programming are covered in the collaborated work of Ying Dong [15]. From [15] we know that the framework required for building a Java game consists of the following classes:

(1) MOB

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;
   }
}
```

The instance of this class represents a movable object. 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.

(2) GraphicsEngine

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 class keeps track of the movable objects and draws them in the proper places when necessary. Class Chain is a linked list used to keep a list of moveable objects that are to be drawn. We draw the movable objects by traversing the whole list.

(3) Game

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;
		}

}
```

This class is the place to install the graphics engine. When drawing movable objects, the drawing task is transferred from here to its installed graphics engine and it is finally the graphics engine that draws these objects.

In the remaining part of this report, we will describe the detailed design and implementation of an on-line game called CrazyRoad1.0, which is implemented on the basis of the above mentioned framework.

Chapter 4: CrazyRoad 1.0 --- Online Game

4.1 Brief Description

CrazyRoad1.0 (hereafter called *CrazyRoad*) is an online game which is designed and implemented in Java as a Java applet, and which users may download from the web site and run it on their local computer using a web browser. It is designed and implemented using all the technologies mentioned above and described in the collaborated work [15] by making some changes and adding some new features to the framework. When the game runs, it is shown in **Figure 2**:

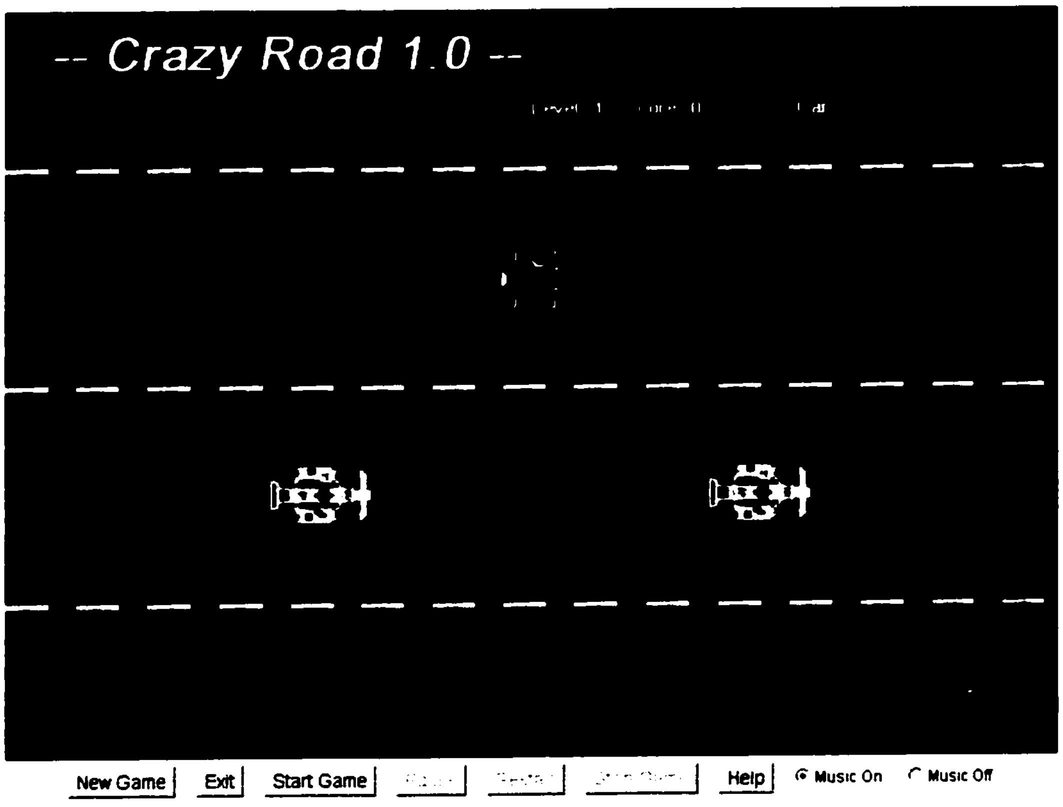

Figure 2. The interface of the implemented game – CrazyRoad1.0

The game is quite simple. From the above figure we can see that there are three lines on the screen. These are the driving lanes on the road. The road is separated into two lanes 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. The user 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 opposite to that of the playing car. In this game, we only allow two possible positions for the playing car to go, which are up per lane or lower lane since this game just has two lanes. At the beginning of the game, the 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 the playing car has collided with an obstacle 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 stops running. On the top of the screen, the user can check the message related to the level of the game the user has reached and the score the user has gained. At the bottom of the screen, there are some buttons from which the user can control the game or do some selection or get help.

4.2 Design

This game uses all the basic Java game programming technologies (or framework) mentioned in section 6. Here we need a *MOB* (movable object) class to represent movable objects, a graphics engine to control all movable objects and a place to install the graphics engine. We also need to make some changes to the framework since now the game we are dealing with is comparatively graphics intensive and with a graphical user interface. There will be four classes in this game instead of three classes as in the basic

framework mentioned in section 6. These classes are ***GameEngine*** class, ***MOB*** (movable object) class, ***CrazyRoad*** class and ***Game*** class. ***MOB*** class represents a movable object. It uses the basic *MOB* class in the **framework** with adding priority and visible features. ***GameEngine*** class controls all movable objects. It still uses the basic structure of a graphics engine in the **framework** but has some changes in it. ***CrazyRoad*** class is a subclass of *Panel. Panel* is a component of AWT and a container that can contain other components. In this program we make ***CrazyRoad*** as a subclass of *Panel* in order to draw those movable objects on it. ***CrazyRoad*** also implements *Runnable. Runnable* is an interface in *java.lang* package. It should be implemented by any classes whose instances are intended to be executed by a thread. As each instance of ***CrazyRoad*** will be executed as a thread, we make ***CrazyRoad*** implement *Runnable* here. The main functionality of class ***CrazyRoad*** is to install the graphics engine and initialize the movable objects and actually control the view of all the movable objects in the movable objects list. ***Game*** class extends *Applet* class. It is the entry point of the game. It loads all the required images, initializes a ***CrazyRoad*** class instance and builds up the graphic user interface. The relationship between the classes can be represented as in **Figure 3**:

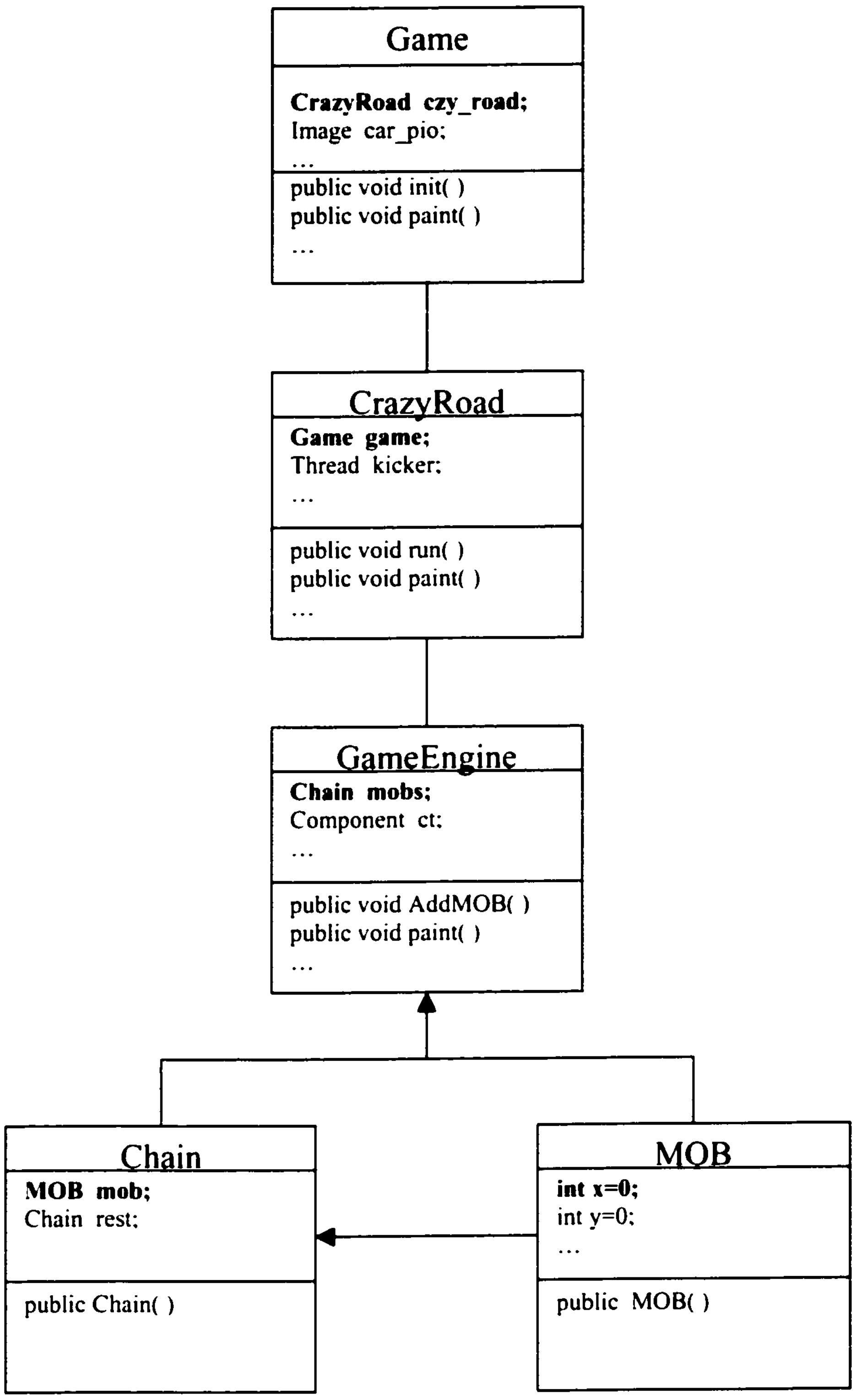

Figure 3. Class diagram for the implemented game – CrazyRoad1.0

The reason why we add a new class ***CrazyRoad*** here instead of using the ***Game*** class to install the graphics engine is that this program has a graphical user interface, if we use applet ***Game*** to control both the GUI for the game and the drawing of movable objects, to draw these movable objects, we have to re-write ***Game***'s paint( ) method and we need to give the position for the movable objects in this method to indicate where to draw them; in this way we also have to draw those GUI components for the game in paint( ) instead of adding these components to some containers and letting the layout manager of the applet to control them, this makes it very hard to give the exact position for these components and also, if during the game we change the size of the window in which the game is running, it will be very difficult to control the extending of these components so that they can fit the change of the window's size well. So instead, we use *applet* ***Game*** to control the user interface and ***CrazyRoad*** to control the drawing of the movable objects.

4.3 Implementation

All the source code can be found in Appendix C.

4.3.1 MOB class

MOB class is the same as the extended *MOB* class in [15]. Its instance represents a movable object. It incorporates the priority scheme to make every movable object have a priority number. The priority of every movable object is initialized with the value of 0 for ease of use.

4.3.2 GameEngine class

Listing 4. The code for class **GameEngine**

```
import java.awt.*;
import java.awt.image.*;
```

```
public class GameEngine
{
        Chain mobs = null;
        public Image background;
        public Image buffer;

        Graphics pad;
        Component ct;
        boolean first_time = true;

        String title = "-- Crazy Road 1.0 --";
        String level = "Level: ";
        String score  = "Score: ";
        String car = "Car: ";
        int level_num = 0;
        int score_num = 0;

        public GameEngine(Component c)
         {
              ct = c;

         }

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

        public void paint(Graphics g, ImageObserver imob)
         {
           if (first_time)
            {
                   buffer = ct.createImage(ct.getSize().width, ct.getSize().height);
               pad = buffer.getGraphics();
               first_time = false;
            }
           if (background != null)
            {
               pad.drawImage(background, 0, 0, ct.getSize().width, ct.getSize().height,
                             imob);
               pad.setFont(new Font("TimesRoman", Font.BOLD+Font.ITALIC, 36));
               pad.drawString(title, 35, 45);
               pad.setFont(new Font("dialog", 1, 14));
              pad.drawString(level + level_num, ct.getSize().width/2, 80);
              pad.drawString(score + score_num, ct.getSize().width/2 + 70, 80);
```

```
                pad.drawString(car, ct.getSize().width/2 + 200, 80);

                    }

            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;
   }
}
```

GameEngine class uses the basic structure of class *GraphicsEngine* in the **framework** with 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 shown by this class. We can see that now the constructor has become:

```
public GameEngine (Component c)
 {
    ct = c;
 }
```

It takes a parameter of the type of *Component* and passes the parameter to its variable *ct*. Here we add a new flag *first_time*. The variables *buffer* and *pad*, which are used for double buffering and are initialized in the *paint( )* method instead of in the constructor. This is because in this program, we add a new class *CrazyRoad*, which installs the *GameEngine* inside it and is the actual controller of all the movable objects. It is the subclass of *Panel*, which is also a subclass of *Component*. *GameEngine* takes a *CrazyRoad* instance as its parameter in this program. When the *applet* is loaded, the actual size of the component *CrazyRoad* is still unknown. So if *buffer* and *pad* are initialized in the constructor of *GameEngine*, when the *applet* is loaded, *buffer* and *pad* will have undefined values. Thus we can see nothing on the screen. So we add a flag called *first_time* here to check if it is the first time the *apple*t is being loaded. If the *applet* is just loaded for the first time, then *buffer* and *pad* will be initialized.

Within the *paint( )* method we can see that we first check if it is the first time the applet being loaded. If it is, then *buffer* and *pad* are initialized and the value of the flag *first_time* is set to false. The variable *ct* here will be an instance of the class *CrazyRoad*, which is a subclass of the class *Component*.

In class *GameEngine* we can see that there are no changes in the *Chain* class. We should notice that before we draw the movable objects in the movable objects list, we first sort the objects in the list according to their priorities. This makes sense because that as those objects are movable objects, when they are moving, their priorities will change that makes the order of the priorities of the movable objects in the list change as well. Since the order of the drawing for these movable objects is dependent on the order of priorities, so every time before they are drawn, we need to sort their priorities in the movable objects list.

4.3.3 CrazyRoad class

CrazyRoad is the class where the graphics engine (here is *GameEngine*) is installed. The reason why we add a new class *CrazyRoad* here has been mentioned in section 7.2. The constructor of this class is shown in **Listing 5**:

Listing 5. The constructor of class *CrazyRoad*

```
public CrazyRoad ( Game gm, Image bk, Image car_pic, Image car1_pic,
                   Image car2_pic, Image sidebar_pic, Image bck_car )
 {
       ...
       game = gm;
      try {
            sound = game.getAudioClip (new URL(game.getDocumentBase(),
                                         "spacemusic.au"));
             hurt    = game.getAudioClip(new URL(game.getDocumentBase(),
                                        "hurts.au"));
        }
       catch ( Exception e )
        { }

      //install engine and register MOBs
       engine = new GameEngine(this);
      engine.background = bk;
       engine.level_num = 1;
       engine.score_num = 0;
      side_bar = sidebar_pic;

      car = new MOB(car_pic);
      ...
      //get the size of all objects
      car_height = car_pic.getHeight(this);
      car_width  = car_pic.getWidth(this);
      ...
      engine.AddMOB(car);
      ...
}
```

In the constructor, we pass the applet *Game* as a parameter, and all the images loaded in the applet as other parameters to the *CrazyRoad* constructor. Also here the *GameEngine* is initialized and all the movable objects are inserted into the movable objects list of the *GameEngine*. In the constructor, at the beginning, two *AudioClip* instances that will be used to generate sound effects are initialized. The reason why we initialize them here instead of giving all these jobs to the applet *Game* is because since these instances are only used in a *CrazyRoad* object, we want to access them directly instead of having to referencing them through a *Game* object.

Having the *AudioClip* instances, we begin to install the *GameEngine*. We can see that we pass a *CrazyRoad* object (keyword *this*) to the *GameEngine* constructor to get a *GameEngine*. Since the variable *engine* is an attribute of the *CrazyRoad* class, so here we actually make the *GameEngine* installed because from the *GameEngine's* constructor we can see that *engine* takes the component *CrazyRoad* as the container for the movable objects. The next step is to register the movable objects (*MOBs*). We firstly initialize the level and the score for the game, then use those loaded images to generate movable objects. After that, we get the actual sizes of all the movable objects. Finally, we call the *AddMOB()* method of the class *GameEngine* to add all the generated movable objects into the movable objects list of *engine*.

Here we use the *x* coordinates as the priorities for the playing cars and obstacle cars, *y* coordinates as the priorities of the driving lines (*side_bar*). Every time before the screen is updated, we use a sort algorithm (usually any kind of sort algorithm can be used) to sort the priorities of the movable objects in the movable objects list of *engine*. In the game, only the obstacle cars can move (change their *x* coordinates). The playing car can only change its *y* coordinate. We make it remain its *x* coordinate (at the center of the screen) all the time during the running of the game. To make the users feel that the playing car is moving during the game, we make the sidebars move to the opposite direction of the playing car continually such that it looks like the playing car is moving as well.

Images

In this program, we load the pictures that will be used to make up all the images (playing cars, obstacle cars, backup cars, driving lines, background,) in the game from some image resources. Then we use Photoshop to edit these pictures (for example, enlarge the sizes, reduce the sizes, rotate, change color, make them transparent etc.) to make them meet our requirements for the game. At last we make them all as transparent *GIF* files. One important thing we should notice here is that we had better make the sizes of these files as small as possible so that the download time of these files can be reduced more and the performance can be much more satisfactory. To load these files, we use *MediaTracker* [1], which is an easy and convenient way to load images with tracking the loading process in Java.

Priorities of the movable objects

For the playing car and the obstacle cars, we make their *x* coordinates as the values of their priorities in this game. Because the playing car will never change its *x* coordinate, its priority is fixed. As the obstacle cars move during the game, their priorities will change. For the *side_bars* that consist of the driving lines, their priorities are fixed as well. They use their *y* coordinates as the values of their priorities. The values of the priorities for the *side_bars* are all the same. It has no problem to make them in this way since there is no such a condition we make in this program that one sidebar shown above another thus which one to be drawn first really doesn't matter. The priorities of the movable objects do not need to be different. The priority of the background is initialized to have the value of 0 and it remains this value through the running of the game. The priorities of other movable objects (other than background) are all initialized above 0. Because of this, all other movable objects are drawn before the background.

Playing car

As there are two lanes in the game, the *y* coordinate of the playing car can be either at the center of the up lane or at the center of the lower lane. Its *x* coordinate always remains the same - at the center of the screen. The changing of its position is controlled by the space bar.

Obstacle cars

The *y* coordinates of the obstacle cars can also be at either the center of the up per lane or the center of the lower lane. But their *x* coordinates will change during the game as they move from one place to another. There are in fact only three obstacle cars in the game. When they move, because they have moving speed and also when they reach the end of the lane on the screen, they will reenter the lanes randomly (which means they might not appear in the same lane as the previous one), in this way they make the user feel that there are more than just three obstacle cars driving on the two lanes.

The driving speed depends on two facts. The first one is the value we provide for each obstacle car about where it should be drawn the next time. This is done using the following method:

```
public void increment (MOB m)
{
      m.x += step_amount;
      if (m.x >= panel_width)
      {
           m.x = 0 - bkcar_width;
           m.y = randomY();
       }
      m.priority = m.x;
 }
```

At the beginning of the game, we initialize the variable *step_amount* with a value. At the end of every running cycle, we add the value of the *x* coordinate of each obstacle car with the value of *step_amount*. Thus the *x* coordinate of each obstacle car will be changed and the next time it will be drawn at a new place. This makes the user feel that the obstacle cars are moving along the road. We should notice that in the *if* block, when the *x* coordinate of an obstacle car is out of the range of the screen, we force it to be the value of zero minus the width of the obstacle car. What does this mean? This means the next time this obstacle car will be drawn from the left side of the screen so that to prevent it from being lost during the game. Another fact to affect the driving speed of the obstacle car is the running time interval of a thread. This can be seen in the *run( )* method.

```
public void run( )
{
   ...
   try
    {
          Thread.sleep(100);
          ...
       }
       catch (InterruptedException e)
       {
          System.out.println(e);
       }
   ...
}
```

By changing the sleep time of the thread, we can decide when to draw the next picture of the game. If the step amount for the obstacle cars is fixed, then reducing the sleep time will increase the moving speed of the obstacle cars. But changing sleep time is not used for changing the moving speed of the obstacle cars in this game. It is used to control the thread that is used to draw the picture of the game to make sure the updating of the picture for the game look smoothly. So we would better fix the sleep time and by changing the step amount we change the moving speed of the obstacle cars. Usually we have to do some tests to find an appropriate value for the sleep time in order to get the best effect.

Here we meet another problem. How do we control that in which lane the obstacle cars will be shown? Or how to control the number of the obstacle cars appearing on one lane at a time? In this program we use a random number base algorithm to solve this problem. It is shown as follows:

```
public int randomY()
 {
      int n = (int)(Math.random()*100);
      if ( n >50 )
        return down_y;
      else return up_y;
  }
```

We can see the method is quite simple. The variables *up_y* and *down_y* represent the center *y* coordinates of the up lane and lower lane respectively. We use the *random( )* method in ***Math*** class to first generate a random number and then to use this to decide where to put the obstacle cars. This makes sense since the obstacle cars should come out in an unpredictable way.

side_bars (driving lines)

The driving lines separate the lanes. When the driving lines are moving, the user will feel that the playing car is moving. But to achieve this is a little tricky. Driving lines are composed of the *side_bars*. The *side_bars* should have some distance intervals and can move from one place to another to make the user feel that the playing car is moving. In this program we use three vectors to store these *side_bars* for the three driving lines respectively. The three vectors are initialized in this way:

```
Vector topside_bars  = new Vector();

Vector midside_bars  = new Vector();

Vector botmside_bars = new Vector();
```

How many *side_bars* we should put into these vectors? Because users may use different kinds of monitors with different screen sizes, here we first calculate the length of each *side_bar* and then according to the length of the screen (or the length of the playing area of the game) of the computer used by the user, we can further calculate the number of the *side_bars* we need to make up of the driving lines. This is done in the *paint( )* method as follows:

```
public void paint( )
{
      ...
      sidebar_num = panel_width/sidebar_width;
       for (int i=0; i<sidebar_num; ++i)
       {
          topside_bars.addElement(new MOB(side_bar));
          ((MOB)topside_bars.elementAt(i)).x = i*(sidebar_width + 20);
          ((MOB)topside_bars.elementAt(i)).y = topline_y;
          ((MOB)topside_bars.elementAt(i)).priority = topline_y;

          midside_bars.addElement(new MOB(side_bar));
          ...

          botmside_bars.addElement(new MOB(side_bar));
          ...

          engine.AddMOB((MOB)topside_bars.elementAt(i));
          ...
       }

   ...
}
```

From the above code we can see that firstly we divide the panel width (the width of the playing area) by the width of a *side_bar* to get the number of *side_bars* we need for the driving lines and then within the *for* loop we put those *side_bars* into the three vectors with making each *side_bar* as a movable object. Then we assign values to x, y coordinates and the priority for each *side_bar* and finally we add all the *side_bars* to the movable objects list in the graphics engine. Now we can control the moving and drawing of the driving lines. The way to control the moving speed of the driving lines (or we can say the speed of the playing car) is similar to the way to control the speed of the obstacle cars. It uses another method:

```
public void incr_step (MOB m)
 {
     m.x += car_speed;
      if  (m.x > panel_width)
      {
          if  (panel_width >= SCREEN_WIDTH)
               m.x = m.x - panel_width + sidebar_width ;

          else  m.x -= panel_width;
      }
 }
```

It is a little different from the way to control the obstacle cars in that usually we found that on different screens with different sizes, the moving *side_bars* will generate different effect if we control them without considering the size of the play area. So in the nested *if* block we can see that we use a constant SCREEN_WIDTH (we set its value as 850 in this game) as the key value (because we make the applet size as 800×600) so that we can maintain the best effect on screens with different sizes. Another difference is that the value for the variable *car_speed* is fixed. So in this game we will feel that the speed of the playing car remains unchanged. In order to make the game more fun or more difficult to play, we are free to choose to change the *car_speed* to make the playing car have different moving speed. For example, we can change the value of *car_speed* in different level so that the *side_bars* can move with different speeds. As we have mentioned before, if we regard *side_bars* as objects standing there without moving, this actually means we are changing the speed of the playing car.

Conflict of the playing car and the obstacle cars

The goal of this game is to avoid the playing car from being hit by the obstacle cars moving towards it so that to play the game as long as possible and to increase the score and level as much as possible. The conflict (hit) of the playing car and the obstacle cars is defined using the following way:

```
public boolean hit (MOB m)
{
    return ( (m.x < car.priority+bkcar_width/2) &&
        (m.x > car.priority - bkcar_width) && (m.y == car.y) );

}
```

This hit() method takes a movable object (in the game, an obstacle car) as the parameter passed to it. It checks the x and y coordinates of the movable object to see if their values have some conflicts with the values of the playing car.

To better understand the code, consider **Figure 4**:

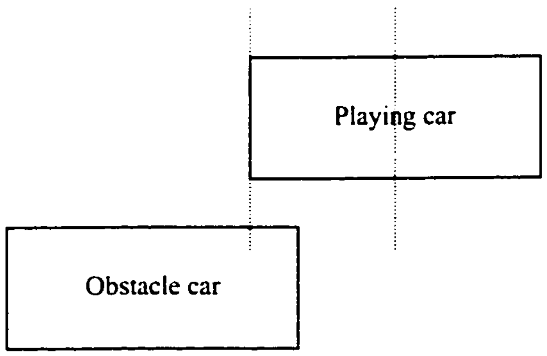

Figure 4. Conflict of the playing car with an obstacle car

From **Figure 4** we can see that if the right side of the obstacle car is within the area between the two dashed lines, it means that the playing car is hit by the obstacle car. Of course the first condition that should be met is that they must be on the same lane.

Every time when the playing car is hit by an obstacle car the number of the backup cars will be reduced by one and the obstacle car that just hit the playing car will be drawn on the other lane to make the user be able to continue the game. One important trick here is that though the hitting obstacle car will be drawn on the other lane, those obstacle cars following it before the collision will remain on their previous way and move towards the playing car at a speed. If they are moving at a high speed, when the hitting obstacle car hits the playing car, they may be at the positions that are very close to the playing car and ready to hit the playing car soon. So after the playing car is hit by the hitting car and the hitting car changes to the other driving way, the user may have no time to change the position of the playing car while the playing car is hit again by another obstacle car which is moving towards it at a high speed. Under such a condition, the game will be very hard to play and the user will find that when he is just ready to have the fun of the game, the game is already over. To solve this problem, every time the playing car is hit by an obstacle car, we not only change the lane of the hitting car, but also reduce the moving speed of all the obstacle cars moving on the road so that to make the user have enough time to get through the surprise caused by what just happened. Let's look at the following method:

```
public void run( )
{
    ...
    if ( hit(car1) || hit(car2) || hit(car3) )
    {
        ...
        if (car_amount > 1)
        {
            --car_amount;
            switch (car_amount)
            {
             ...
```

```
                }
                step_amount = -10;
                 ...
            }
          ...
        }
}
```

From the above code we can notice that every time the playing car collides with an obstacle car, the moving speed of all obstacle cars is reduced by10. But here we meet another problem, which is that because the algorithm used to calculate the game level affects the value of the moving speed of the obstacle cars, if the hitting occurs at the time not long after the game just changing the level, then the level will remain to be the new one but the speed is lower than the requirement for this level. This does not make sense because now the speed is actually below the requirement of this level. This produces some unfairness for the users. To solve this problem, we first make the level unchanged when the hitting occurs, and at the same time after the obstacle cars' speed has already been reduced by 10, each time we add the speed with a small value to make it come to the desired speed for this level gradually. In this way the speed will reach the required value for this level soon. (The time the speed returning to reach the level is controlled by the small value mentioned above. Making the value bigger can short the time but may be still not enough for the user to overcome the problem. This value is usually chosen by doing some tests.)

Thread

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method *main* of some designated class. The daemon thread needs some other threads that are not daemon threads to be running. It doesn't need any child threads of its own. And if we have a program running we always have at least one thread running. So when we create a daemon thread and start it, it will start running right away and keeps doing it until we stop it or all non-daemon threads have exited.). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The *exit* method of class *Runtime* has been called and the security manager has permitted the exit operation to take place.
- All threads that are not *daemon* threads have died, either by returning from the call to the *run* method or by throwing an exception that propagate beyond the *run* method.

Programs run from start to finish, executing a single thread of control. Tasks are performed in linear sessions, where one task must wait for another task to be completed before it can have its turn. This is the schematic of a single-threaded program.

Multithreading is the execution of several threads at the same time where each task is performed concurrently with other tasks. Each thread receives a priority value and is allocated a certain amount of system time or attention based on this.

Threads apply well to Java because each applet can run in its own thread without interfering with system resources. On web pages, this enables users to download files in the background while listening to sounds and viewing animations. More than one applet can run on the same page and each will receive an amount of time to advance in its current task.

There are two ways to create a new thread of execution. One is to declare a class to be subclass of *Thread*. This subclass should override the *run* method of class *Thread*. An instance of the subclass can then be allocated and started. The other way to create a thread is to declare a class that implements the *Runnable* interface. That class then implements the *run* method. An instance of the class can then be allocated, passed as an argument when creating *Thread*, and started. In this game, we make the game run as a thread by making the class whose instance will be executed as a thread implement *Runnable* interface. As we already mentioned, we make the *CrazyRoad* class implement *Runnable* interface. In the *start( )* method of *CrazyRoad* class, we pass an instance of *CrazyRoad* to *Thread( )* to generate a new thread as follows:

```
public void start()
 {
      ...
      kicker = new Thread(this);
      kicker.start();
   }
```

Here we can see that a *CrazyRoad* instance (key word "*this*") passes itself as the argument to the *Thread* constructor to generate a new thread. When we call the *start( )* method of a thread, it will invoke the *run( )* method of the thread object. In our implemented game the *run( )* method in the *CrazyRoad* class is called. Here we should pay some attention on those methods related to a thread such as *run( )*, *stop( )* etc. since some of them are dead lock prone and have been deprecated in the latest Java version [4].

Scoring

The scoring algorithm used in this program is quite simple. We can see it in the following code:

```
public void run()
 {
    Thread me = Thread.currentThread ();
    while (kicker == me)
    {
          ...
          ++engine.score_num;
          ...
    }
    ...
}
```

We can see that as long as the current thread is alive and running, the score is simply increased by one during every cycle of the *while* loop. Actually there are a lot of alternative algorithms to selcct for scoring; for example, we can calculate the time the game has been played and use the accumulated value as the score the user get, but here we choose to use a very simple way.

Level of the game

There are a lot of alternatives we can choose to use as the algorithms to decide the level of the game. In this program, we use the score the user gets when playing the game to decide the level of the game. This is achieved in the *run( )* method as follows:

```
public void run()
 {
    Thread me = Thread.currentThread();
    while (kicker == me)
    {
       ...
       ++engine.score_num;

        if ( (engine.score_num >=400) && (engine.score_num < 800) &&
           (engine.level_num != 2) )
         {
           ...
         }
        ...
    }
}
```

This game is designed to have five levels. Level one for scores under 400; level two for scores from 400 to 800; level three for scores from 800 to 1800; level four for scores from 1800 to 2000; and level five for scores above 2000. And also we should notice that the moving speed of the obstacle cars is different at different game level. Higher levels have higher speed so that higher levels will be harder to play with than the lower levels.

Sound effects

In this game we use the methods of *java.applet.AudioClip* interface to generate the sound effects. There are two places where we use sound effects here. One is the background music when the playing car is moving without being hit by any obstacle cars. Another place is when the playing car collides with an obstacle car. Obviously we should use the *loop( )* method in the *AudioClip* interface to play the background music and use the *play( )* method to give the hitting sound effects. To get the *AudioClip* instances, we can use the following:

```
...
try {
        sound = game.getAudioClip ( new URL(game.getDocumentBase(),
                                            "spacemusic.au"));
        ...
    }
catch ( Exception e )
  { ... }
...
```

The above code loads the sound file from the specified places and initializes the *AudioClip* instances. There is some trick in managing these sounds such as when they should be played, when they should be stopped or when they should be restarted etc. Here we will introduce our approach briefly.

In this game, we have two kinds of sound, one is the background sound and the other one is the sound when a collision occurs between the playing car and an obstacle car. Here we use two flags to control these sounds. The flag *has_music* controls the background sound and the flag *activate_music* is used by the GUI (radio button and shortcut key) to control if we need sound effects. When the game starts, *has_music* is initialized with the value of *true* and *activate_music* is initialized with the value of *false*. At this time we call *start()* method of *CrazyRoad* once. Thus only the background sound plays and the hitting sound keeps quiet. And now though the value of *activate_music* is false, as the background sound is playing and is not stopped by any command, so *activate_music* at this time does not affect the playing of sounds. Suppose we click the radio button on the screen to turn off the sound, the sound will be stopped immediately. When we click the radio button to turn the sound on, as now the thread that is running is *kicker*, and the value of *activate_music* turned to be *true*, now *activate_music* works and the background sound can be played again.

When the playing car collides with an obstacle car, the background sound will stop and the hitting sound will play. After this, the hitting sound should stop and the background sound should replay to continue the game. As we know that the playing of the hitting sound needs some time, so if we do not control these two sounds properly, they might be overlapped each other. So here we add another variable called *music_trigger*. When a hitting happens, we give *music_trigger* value of 1 and then continue the game with adding 1 to *music_trigger* during each loop until its value reaches a number we predefined (usually this number should be gotten from test according to different sound file). Then we check the value of *music_trigger* and the value of *has_music* to decide if the sound effects should start.

The other features

There is one feature that we want to mention here. To make the game a little more interesting for users, we make the game change its background when the user reaches a certain level. This can bring some changes to the game so that the users feel more fun. To achieve this is quite simple. We just load two or more images for the background and tell the engine which one should be drawn when the need of changing background occurs. As for the backup cars, we also add them to the movable objects list though they cannot move. Every time the playing car collides with an obstacle car, we make one of the backup cars that are stored in the garage invisible. So when the screen is refreshed, the user will see that the number of the backup cars in the garage is reduced by one.

4.3.4 Game class

As we have mentioned, the *Game* class is the entry point of this program. It extends the *Applet* class and implements the *KeyListener* interface in order to add some key controlling function for this program. It loads all the images used in the program and builds up the graphic user interface for the game. It uses *cardLayout* (A CardLayout object is a layout manager for a container. It treats each component in the container as a card. Only one card is visible at a time, and the container acts as a stack of cards.) as its layout because we want to switch to the help page within the applet instead of linking to another page to avoid another connection to the server. It uses a *CrazyRoad* instance as a component to be added into it. It also defines all the events and behaviors related to the components making up of the user interface. Since the entry point of the game is an *Applet*, after compiling (the *Game.class* is generated), we should integrate the class file into a HTML file so that user can load the game referred in the HTML file which is provided to the user by a web server. One thing we like to discuss here is the loading of the images. In this program we use *java.awt.MediaTracker* class to do the task of loading images [1].

The *MediaTracker* class monitors the status of media types, such as images, sounds, etc. A common concern with applets is that images are sometimes referred to before they are fully loaded, which has unpredictable results. *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.

When using *MediaTracker*, there are three steps to achieve the goal:

- Create an instance of *MediaTracker.*
- Call *MediaTracker.addImage( )* to indicate the image we want to track.
- Generate the try/catch blocks. In the *try* block, the instance of *MediaTracker* waits for the loading of the image that has a specific ID to be complete.

If we want the image loaded completely before being shown, we can use *MediaTracker*. We can see the code as follows:

```
...

try
   {
     //get the pictures of the objects
    MediaTracker tracker = new MediaTracker(this);

    bk = getImage(new URL(getDocumentBase(), "background.gif"));
```

```
        ...
        tracker.addImage(bk, 0);
        ...
        tracker.waitForID(0);
        ...
    }
catch ( Exception e )
    {
        ...
    }
...
```

The three steps in the above code are very clear. We can either put all the three steps into the *try* block or just the last step. Here we put everything into the *try* block.

We can also notice the building of the user interface in this class. One thing we should pay more attention is the implementation for the function of the short-cut keys. The trick things here are the changing of focus and the different facts on different platform.

4.4 Running the game

To run the game, we should put all the compiled files into the database or some directory residing on the server side and integrate them into the HTML file which the server provides to the user. We should tell where to find the *Applet* (the path to the *Applet*). The usual way to do this is:

```
<HTML>
...
<APPLET code="Game.class" width = 800 height = 600>
</APPLET>
...
</HTML>
```

We can also pack all the required files into 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 the JAR file into the local computer with which the user is using. So we can rewrite the HTML file as:

```
<HTML>
...
<APPLET code="Game.class" archive="awtGame.jar" width = 800 height = 600>
</APPLET>
...
</HTML>
```

If we want to run the Swing version of the game, we have to make sure that our browser supports Java Swing. As on result of our research, we found that Internet Explore 4.0 above and Netscape4.0 above all support Java AWT. But only the latest Netscape6.0 above support Java Swing. So for the browsers that do not support Java Swing, we have some alternatives to make the Swing version run on these browsers. One is to pack the Swing package into a jar file and load it into the local computer of the user. This can be done as follows:

```
<HTML>
...
<APPLET code="Game.class" archive="swingGame.jar, swingall.jar" width = 800
            height = 600>
</APPLET>
...
</HTML>
```

But this way is tedious because this jar file is about 1 MByte. It will cost users (particularly those users who access the internet through common telephone lines) quite a lot of time to download this file.

Another way we can use is to download Java *plug-in* patch program from *java.sun.com* and load it into the user's browser. The Java *plug-in* patch program also has a large size but user only needs to load it once and thereafter user does not need to download it any more to make his browser support Java Swing. But using the first way, the user has to load the jar file every time when he wants to run this game. The *plug-in* way can be shown as follow:

```
<HTML>
...
<EMBED type="application/x-java-applet;version=1.1"
    pluginspage="http://java.sun.com/products/plugin/1.1/plugin-install.html"
    code="Game.class"
    archive="swingGame.jar"
    width = 800
    height = 600>
</EMBED>...
</HTML>
```

4.5 Improvements and Extensions

There are a lot of ways to improve this game. That's why we call it *CrazyRoad1.0.* We leave it for the future extending.

Some suggestions we can put forward for the extension as:

- Increase number of the levels involved in the game.
- Change the algorithm for the scoring.

For example, we can make the scoring depend on the time used by the user who is playing the game, or depend on the number of obstacle cars the playing car has avoided. We even can give some bonus to the user who has reached higher level. The bonus can be either adding some score to the user or increase the number of backup cars in the garage.

- Change background between every different level.

In this program, we only change the background once. We can also think about changing the background every time the user reaches a different level.

- Add more visual effects.

We can add some gas trail on the back of the playing car and the obstacle cars to make the moving of these cars looks more real. When the playing car collides with an obstacle car, at the hitting part of the cars we can add some broken pieces to stress the hitting effects.

- Change the story.

The most interesting thing we can think of is to change the story of the game. For example, in a certain level, the playing car can be equipped with guns to shoot the obstacle cars moving towards it. Or there are not only the obstacle cars which can hit the playing car, but also we can add some other obstacle objects like bombs, unmovable stones etc. to collide with the playing car. We can even add some planes on the air to throw some bombs. In this way, the playing car will have to be more active which means it can change both its coordinates. All in all, there are too many things that can be improved, can be done, can be expected, can be imagined on this aspect.

- Add more sound effects.

Sound effect usually can make the game more interesting and catch more of the user's attention. With the sound effects, users can be more excited and stimulated.

- Set up top score player file.

We can set up a top score player file, for example top 100 players file, on the server side such that every time the user finishes one game, if his score is within the range of the scores in the top player file, then he is allowed to add his name into this file to make himself as one of the top score players. This makes the game to have more competition features and so makes the game more interesting. One important feature that can make the game much more fun is to make it playable by two or more users on the web to compete with each other. This means we can play the game with real people on line.

- Make it as a Java Application

Making this game as a Java application instead of an *Applet* can make the user play the game off line. On the other hand, without the security constraint of Java *Applet* [4], the application can be extended easily. And the most important thing is that it can be saved in the local computer or other storage media instead of downloading from the slow web, thus it can be run much faster than running on the web.

4.6 Problems when running this game

After we finished the implementation of this game, we did some tests and found some problems. One major problem is related to the browser that the game was running in. This is the biggest issue.

As we have mentioned before, we made two versions for this game: AWT version and Swing version. The reason why we made two versions for this game is mainly because the user interface for the game can either be done by using Java AWT or by using Java Swing. 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 aren't restricted to the least common denominator -- the features that are present on every platform -- they can have more functionality than AWT components. Swing lets us specify the look and feel our program's GUI uses. By contrast, AWT components always have the look and feel of the native platform. So when designing some Java application programs that have GUI, we are recommended to convert AWT to

Swing. The strongest reason to convert to Swing is because it offers many benefits to programmers and end users. Among them:

The rich set of ready-made components means that we can easily add some snazzy features to our programs -- image buttons, tool bars, tabbed panes, HTML display, images in menu items, color choosers etc.

We might be able to replace or re-implement some custom components with more reliable, extensible Swing components.

Having separate data and state models makes the Swing components highly customizable, and enables sharing data between components.

Swing's Pluggable Look & Feel architecture gives us a wide choice of look-and-feel options. Besides the usual platform-specific look and feel, we can also use the Java Look & Feel, add an accessory look and feel (such as an audio "look and feel"), or use a third-party look and feel.

Swing components have built-in support for accessibility, which makes our programs automatically usable with assisted technologies.

But it is reasonable to put off converting if we don't think our users will be able to run Swing programs conveniently. For example, if our program is an applet and if we want anyone on the Internet to be able to use it, then we have to consider how many Web surfers have browsers that can run Swing programs. For browsers that don't have Swing support built in, the user must add it by downloading and installing Java *Plug-in.*

Before testing the game using any web browsers, we tested it in Java JDK's *appletviewer.* Both versions work correctly as we expect. Then we tried it on some major browsers. We first test this game using Internet Explore 5.0. Both the AWT version and the Swing version worked very well. Then we tested it using Netscape (we used 4.6 and 4.7), it does not work at all. Later we tried it on Netscape6.2, the AWT version works very well but the Swing version works with some problems (the function keys do not work properly). We found that this is because of the browsers. Netscape 4.x browsers do support Java but they have some problems to deal with images. Internet Explorer has very good ability to process images so both versions of the game work properly using IE. But the good news is that Netscape6.x browsers have been greatly improved so now both the AWT version and Swing version can run in them. While there are still some problems when running the Swing version in Netscape6.x browsers and these problems are all related to keyboard. Problems caused by using keyboard as the input device is a small issue since we know that the key functions really have different effects under different environment because of different interpretation of the keystrokes under different platforms. But we can think that this is a constraint for Java's platform independent conception.

One way to run this program without using any browsers is to run it using *appletviewer*, which is attached with JDK and any other Java development IDE. But one drawback to run the program in this way is that it does look not so smoothly as it does in the browsers because of the thread structure in its virtual machine. This problem also arises when we run it on our PC at home and run it on a computer connected to Windows NT.

During this project, I and Ying Dong [15] made a few observations on programming in general and Java game programming in particular. These observations are presented in this chapter.

Chapter 5: A further discussion on Java and Java game programming

5.1 Some common 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.

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

The principle of abstraction is one of the cornerstones of object-oriented programming. Since Java is object-oriented, knowing when to abstract becomes even more important because each object usually should have distinct functionality in a program. For example, in our game we separated the classes according to their different functionalities 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 themselves. 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. We avoid using these classes whenever possible, especially for game programming. Firstly, these classes can really affect the throughput of our applications since some of these classes such as *Method*, when programming using these classes, it is quite inconvenient, error prone and slow, and as we know speed is a critical component to any game. Reflection classes also make our code unclear with adding bunch of code that have no help for our code to implement its functionality.[1] So as a general rule, do not use reflection and use preferred Java techniqucs such as anonymous inner classes, subclasses, or, if we need to, a *switch* block, to discriminate among different routes of flow control.

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

This includes incorporating things such as image strips (for details please refer to [1]) into the projects, as well as eliminating the use of synchronization 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 crashing completely, by all means use it.

Excessive use of threading can bring some problem to the programs as well. Using a few threads in the games is usually necessary, but having 500 movable objects each using its own thread may cause the machine to spend all of its time switching between threads, this will slow things down too much.

Also, because of the improvement of machine memory in these years, we would generally take speed over size if it comes down to choosing between the two.

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

Though 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. Components such as *JApplet*, *JFrame*, *JButton*, etc., are based upon their lightweight AWT counterparts and contain the ability to take on a pluggable look and feel of any operating system. If we use Swing just for the GUI in common applications instead of game development, Swing is really the preferable choice than AWT. But for game programming, we feel that the benefits that Swing provides do not outweigh the speed 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. For instance, say we have used the *Image* class to load and draw images from file. This is an abstract class, and it is programmed abstractly. On Win32 platforms, the *Image* class is implemented as the *Wimage* class found under the *sun.awt.window* package. Obviously, this package is not part of the standard Java API. Furthermore, this class or its methods are not guaranteed to exist in further Java implementations. So even if we program our games exclusively for Windows, directly using the *Wimage* class, or any other sun.* package class, is risky business.

- **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, the fewer calculations the game code makes, the faster it will run.

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

Although we 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. But optimizing Java doesn't mean that we should not extend upon available Java classes. We have made hundreds of extensions to what is available in the standard Java library. What we mean is that if some people feel as though they have developed a better class, the fact usually is that they probably haven't.

5.2 A further discussion on Java

To be successful, a system must have a user base. The users must get enough of their tasks achieved within reasonable limits of performance, and given reasonable limits of training. From its first public release in 1994, Java has rapidly become a very popular programming language. Java is associated with the World Wide Web -- and the Web's global scale -- and it has applications near and far, from smart cards to the 2001 Mars Lander. There are hundreds of books on Java, specialist magazines, and web sites. Java is now taught in hundreds of universities. Java is clearly a mainstream phenomenon.[16]

In many ways, Java is a classic computer application, with its design and introduction requiring trade-offs. Its design had to balance being different and being "better." It had to successfully draw on enough users to make it a viable product. Java closely resembles C and C++, so existing programmers find it familiar. However these languages have many problems and ambiguities, so Java made changes in order to have advantages over them. Compared to C/C++, Java removes some features from them. These are:

Java has no more Typedefs, Defines, or Preprocessor

In Java there is no *preprocessor*, no #define and related capabilities, no typedef, and absent those features, no longer any need for *header files*. Instead of header files, Java language source files provide the declarations of other classes and their methods.

Java has no more Structures or Unions

Java has no structures or unions as complex data types. We can achieve the same effect by declaring a class with the appropriate instance variables.

Java has no Enums

Java has no *enum* types. We can obtain something similar to enum by declaring a class that holds its attributes as constants. We could use this feature something like this:

```
  class Direction extends Object
{
    public static final int North = 1;
    public static final int South = 2;
    public static final int East  = 3;
    public static final int West  = 4;
  }
```

Java has no Functions

Java has no *functions*. Anything we can do with a function we can do just as well by defining a class and creating methods for that class.

Java has no Multiple Inheritance

Multiple inheritance was discarded from Java. The desirable features of multiple inheritance are provided by *interfaces*.

An interface is not a definition of a class. Rather, it's a definition of a set of methods that one or more classes will implement. An important issue of interfaces is that they declare only methods and constants. Variables may not be defined in interfaces.

Java has no Goto Statements

Java has no goto statement. goto statement may mislead users, especially when it is used in some loops. Most of C/C++ developers and programmers were ever warned that they should pay attention when they use goto statement, or they were recommended not to use it at all. In Java, eliminating goto led to a simplification of the language.

Java has no Operator Overloading

There are no means provided by which programmers can overload the standard arithmetic operators. Once again, the effects of operator overloading can be achieved by declaring a class, appropriate instance variables, and appropriate methods to manipulate those variables. To some degree eliminating operator overloading leads to simplification of code.

Java has no Automatic Coercions

Java prohibits C and C++ style *automatic coercions.* If we wish to coerce a data element of one type to a data type that would result in loss of precision, we must do so explicitly by using a cast. Consider this code fragment:

```
int myInt;
double myFloat = 3.14159;
myInt = myFloat;
```

The assignment of myFloat to myInt would result in a compiler error indicating a possible loss of precision and that we must use an explicit cast. Thus, we should re-write the code fragments as:

```
int myInt;

double myFloat = 3.14159;

myInt = (int)myFloat;
```

Java has no Pointers

Most studies agree that *pointers* are one of the primary features that enable programmers to inject bugs into their code. Given that structures are gone, and arrays and strings are objects, the need for pointers to these constructs goes away. Thus, Java has no pointer data types. Any task that would require arrays, structures, and pointers in C can be performed by declaring objects and arrays of objects. Instead of complex pointer manipulation on array pointers, we access arrays by their arithmetic indices. The Java run-time system checks all array indexing to ensure indices are within the bounds of the array.

We no longer have dangling pointers and trashing of memory because of incorrect pointers since there are no pointers in Java.

Most of these above changes to some degree make Java simple, easy to use and prevents it from error prone.

But on the other hand, Java has unfortunate and avoidable weaknesses. For example, here are some illustrative notational issues:

1. In Java the 32 bit numeral 71 can be made into a 64 bit numeral by appending a letter l, as in 71l. Arguably, the notation should make this sort of difference clearer. Or it would have been easier if Java had been designed so numerals took just as many bits (char, int, long) as they required, and the compiler complained when there was numeric overflow.

2. Java uses Unicode, so a Java letter (for use in an identifier name) can be from almost any language, so A (Latin alphabet) and A (Greek alphabet) are different.

3. Casts in Java are prefix (as in C). If they were postfix, fewer brackets would be needed and the code would be more readable. For example, Java's ((AClass) a.elementAt(n)).action() could be more conveniently written, without having to balance nested brackets across arbitrarily long pieces of code, as a.elementAt(n) (AClass).action().

4. Java's statement syntax "taken over" from C allows compound statements wherever simple statements are permitted. Thus, the conditional statement in an *if* can either be a simple statement, or a compound statement grouped by curly brackets. Java introduced a new control structure (throw, catch, finally) but the statements guarded by these structures *must* be "compound." For example, it is not permitted to write try a = b/c; instead one has to write try { a = b/c; }

Besides the notational issues, the programmer faces two quite different sorts of more serious problem: *barriers*, which are explicit limitations to desired expressiveness, and *traps*, which are unknown and unexpected problems. Typically, a barrier reveals itself as a compile time error, or in the programmer being unable to find any way to conveniently express themselves. A trap, however, is much more dangerous: typically, a program fails for an unknown reason, and the reason is not visible in the program itself.

Java has become very popular largely because of its improved type checking, its run time array bound checks, and its removal of explicit pointers: all these improvements can be understood as converting traps in C and C++ into barriers in Java: thus helping programmers write more robust programs.

In contrast, Java's rules for variable initialization form a trap. There are several different sorts of initialization (that apply differently to local variables, class variables, parameters, *etc.*, and to different types -- primitive, class, and array), and Java provides protection against only one sort of initialization error. But with the apparent guarantee, a programmer might accidentally rely on a class variable or an array element being correctly initialized. Unfortunately, they are not. And since Java has no 'undefined' value any such incorrect assumption is unlikely to be easily discovered.[16]

There are also some other issues, anomalous features and unnecessary confusion etc. in Java. Like importing packages design issue, some of the obfuscation of Java's inheritance and polymorphism mechanisms, anomalous strings and arrays (Java's parameterized classes), Java being strongly typed and being unnecessarily confusing etc. For the detail of these issues please refer to [16].

Java is successful, and improving it in an "objective" sense would be to forget the vast investment programmers have had in learning Java as it is. The conclusion, then, is not that Java should be changed, but that when designing a system, certainly one intended for a world wide market, one should take -- should have taken -- great care to explore the design issues.

Although Java is now more robust than when it was first released, why wasn't it released with a test suite? After its early version (1.0), Java was quickly replaced by Java 1.1 -- even though the designers *had* said they believed Java to be a "mature language, ready for widespread use." All the revisions 1.1 (and 1.2, and so on) represent lost time to a huge number of programmers who must now learn and re-learn the extensions and variations, as well as the time they will waste recoding existing applications so that they still compile.

Chapter 6: Conclusions

As we have seen, game programming in Java is quite a similar task as in any other languages like C/C++. They have the same basic ideas. They usually should have the same problems during the design and the implementation. But we should notice that Java itself has some better features over other programming language and it is mostly these features that make Java to be successful and wildly used. Some tasks referring to the game programming could be easier in Java, such as graphics processing, double buffering, some data structure, integration with multimedia and thread programming etc. Some of this kind of simple and easy to use aspects can often be seen in *GUI* construction for applications programs. One very important feature of Java is that it is a cross-platform language. The programs implemented with it can run on any platforms without making any changing.

In [15], a basic graphics engine is developed with Java that can be used for game creation. This graphics engine incorporated movable objects with prioritization and visibility settings, double buffering and a background image.

Game programming is different from other application programming. Once we get the tools, the remaining jobs are more focused on composing stories, art work imagination, art work editing, dealing with multimedia etc.

Some issues in developing games with Java are touched in [15]. It is important for us to keep these issues in mind in order to write reliable, robust game applications for the users. When we develop our games, we should also be aware that people would want to run them on machines that may not have the same capabilities as our machines.

We got some very useful experiences during Java programming. With the fast improving and extending features of Java, we should not forget to put ourselves at a middle ground so that to have a fair point of view on it. Good programming requires using a good language. The way to understand a language is a good indication of how well it is designed; ideally, one should be able to learn incrementally, building constructively on past learning. Simple things should be easy, complex things should not conflict with simple things. But with Java, one always has to revise one's "knowledge" of it as more is learnt.

The problem with a complex language like Java is that so much is unsaid. Sometimes this results in clearer and more compact programs. They don't need to mention garbage collection, and they can't get it wrong. But sometimes it leads to incredible but hidden complexity -- such as the obscure rules for inheritance.

If we regret some aspects of Java's design, then we must ensure that future designers take better account of good design practice. Many Java programmers (including myself) believe Java is a great success. Yet their programs are usually written in a Java-like subset of C. They surely gain by not having the risks of pointers and unchecked array subscripting. Thus, we cannot conclude it is just the design of Java to blame: much of the poor quality of programming (including the rough-and-ready implementations of Java and its packages) is due to lack of programming skill. "Proper" computing science, including human computer interaction and software engineering, has been taught long enough to be well known; it is now time well-trained designers and programmers start to raise standards. They need to be taught not just what is, but what could be.

But finally we should say that Java is widely used, and this in itself is sufficient reason to teach and learn it. It also has useful features that in themselves are useful to learn, such as concurrency and object orientation; the convenience of these being available in a single language also makes it a good choice as a teaching language. Developers and programmers (including me) have lot of expectation on Java. They hope Java becomes more powerful, robust and easy to learn and use. They hope Java can bring revolution to modern computer science. Can Java really do that?

While we are waiting to see what Java can bring further to the world, I think we also should think about what kind of roles we – ourselves will play in this revolution.

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 Volume I

[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]. Elliotte Harold, "Café au Lait Java FAQs, News, and Resources" http://www.ibiblio.org/javafaq/javafaqhtml

[12]. Harold, Elliotte R. "comp.lang.java FAQ." http://sunsite.unc.edu/javafaq/javafaq.html

[13]. O'Connell, Michael. "Java: The inside story." http://www.sun.com:80/sunworldonline/swol-07-1995/swol-07-java.html

[14]. McCarthy, Vance. "Gosling On Java." http://www.datamation.com/PlugIn/java/03ajava2.html

[15] Java Game Programming – with a case study, major report, Concordia University 2002, Ying Dong

[16] Harold Thimbleby, "A critique of Java"

http://www.cs.mdx.ac.uk/harold/srf/javaspae.html

[17] David J.Eck, "The Java Virtual Machine"

http://math.hws.edu/javanotes/c1/s3.html

Appendix

A. How to load this game

We can put this game in the database at the server side and provide link to the game in the web page that is shown to the user. We can either require the user render a user name and a password to access the game page or do not need any authentication to play the game at all. After the user load the game, the game will be run on the local computer of the user.

B. How to play this game

From the **Figure 2** we can see what the game looks like when it is loaded.

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. The user 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 opposite to that of the playing car. At the beginning of the game, the 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 the playing car has collided with an obstacle car, the number of the backup playing cars will be reduced by one. If the playing car on the road collides with an obstacle car and at this time there are no more backup playing cars in the garage, the game is over and stops running. On the top of the screen, the user can check the message related to the level of the game the user has reached and the score the user has gained. At the bottom of the screen, there are some buttons from which the user can control the game or do some selection or get help. The name of the buttons and their functionalities are described as follows:

Buttons:

[New Game] - pushing this button, the user can clear the screen and begin a new game

[Exit] - to exit the game and change to another page

[Start Game] - Starts the game

[Pause] - pushing this button, the user can pause the game while the game being executed

[Restart] - restarts the game which was just being paused

[Stop Game] - stops the game

[Help] - shows the help page

Music On - turns on the music

Music Off - turns off the music

To make it easy to play and control the game for the user, this game also provides some short-cut keys. These short-cut keys are:

N ------- New Game

Q ------- Exits

A ------- Start Game, Restart Game

S ------- Pauses the Game

C ------- Stops Game

O ------- Turns Music On

F ------- Turns Music Off

F1 ------- Shows the Help Page