Helsinki Metropolia University of Applied Sciences
Degree Programme in Information Technology

Chao Wei

Design and Implementation of a Tank War Game Application with the Active Floor

Bachelor's Thesis. 15 March 2010
Instructor: Kari Salo, Principal Lecturer
Supervisor: Olli Hämäläinen, Senior Lecturer
Language Advisor: Taru Sotavalta, Senior Lecturer

Helsinki Metropolia University of Applied Sciences **Abstract**

Author	Chao Wei
Title	Design and Implementation of a Tank War Game Application with the Active Floor
Number of Pages Date	62 15 March 2010
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Instructor Supervisor	Kari Salo, Principal Lecturer Olli Hämäläinen, Senior Lecturer

The purpose of the project was to study the basic principles of the active floor by ELSI technology, and to develop a computer game to prove that the active floor is endowed with more expansibility for other commercial fields. The goal of this project was to develop an interactive computer game for entertainment based on the existing active floor. In addition to this, the project aimed at implementing such a game control panel on the mobile phone.

The project was carried out by implementing combination programming technologies for three different sides (game server, active floor and game client) of the game including Java technology and Python technology. The development process was following the V model.

The results showed the game was playable with the mobile phone, extendible and it is deployed easily. The results also proved that the active floor used in the game is possible to extend to commercial fields in the entertainment business. However, the results demonstrated the game was not a real game without implemented artificial intelligence. It is recommended to further develop the current implementation for improved user interface and to fix potential bugs.

Keywords	computer game, active floor, mobile phone, V model

Contents

ABBREVIATIONS

3D	Three-dimensional
API	Application Programming Interface
CLDC	Connected Limited Device Configuration
CPU	Central Processor Unit
CRT	Cathode Ray Tube
GPU	Graphics Processor Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
Java SE	Java Platform, Standard Edition
Java EE	Java Platform, Enterprise Edition
Java ME	Java Platform, Micro Edition
JSP	JavaServer Pages
JVM	Java Virtual Machine
MIDP	Mobile Information Device Profile
PDA	Personal Digital Assistant
TankWar	Tank War Game Application
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 INTRODUCTION

The goal of the project is to study the basic principles of the active floor by the ELSI technology, and to develop a computer game to prove that the active floor is endowed with more expansibility for other commercial fields, for instance entertainment. Moreover, the game should be playable with the mobile phone, extendible and be easily deployed.

To achieve this goal, an interactive computer game for entertainment based on the existing active floor will be developed. In addition to this, the project aims at implementing the control panel of a game on the mobile phone. In that way, the game would allow people to use their own phone to play the game.

The scope of this project is limited to a basic and a simple implementation based on the target technologies, thus narrowing the range of the study to the properties of interest. First, the game does not contain any artificial intelligence support, and second, there is an unknown factor whether the mobile phone control could be implemented in this project.

2 ACTIVE FLOOR

The product of the active floor used into this project is named the Elsi floor and it is a part of the Elsi™ underfloor monitoring system developed by MariMils® Oy. The system is a new generation monitoring and security system for elderly care and other monitoring applications [1]. The main functionality of the system is tracking people's movements, which alerts the care personnel when something out of the ordinary happens, and the system provides real-time information of the resident's movements in the nursing home area. The system is already deployed in a nursing home in Kustaankartano, Helsinki, which is the second largest nursing home in Finland [2].

Figure 1. A piece of Elsi floor, photo by Wei Chao

The Elsi floor is sensitive and can detect a minute change when an object touches the floor for an unusually long time. In Figure 1 one piece of the real Elsi floor is shown. The Elsi floor is based on a sensor foil that is installed under the flooring. All common floor materials can be used on top of it. The ends of the foils are equipped with electricity units which are concealed inside the baseboards. The electronics are further connected via standard cabling to the Elsi server. The server contains intelligence of the system and user interface for the nurses. [3]

3 COMPUTER GAME HISTORY AND DEVELOPMENT REVIEW

People always need games, because games are a fundamental part of human existence [4]. The first electronics game is acknowledged to be born in 1947, and the game was designed for playing on a Cathode Ray Tube (CRT). This very simple game was designed by Thomas T. Goldsmith Jr. and Estle Ray Mann in United States of America. [5]

As electronic technology was growing fast, in the early 1960s the first computer game was born and the game was called “Spacewar” and developed by Martin Graetz, Alan Kotok and Steve Russell at the Massachusetts Institute of Technology, the United States of America [6]. The game was run on a PDP-1 (Programmed Data Processor-1) computer. PDP-1 had an 18-bit word and had 9 kilobytes as standard main memory. The magnetic core memory's cycle time was 5 microseconds, and the clock speed was approximately 200 kHz. [7]

Due to limited hardware resources, in the early 1980s, a single developer with strong programming skills could handle almost all the tasks of developing a game. Nowadays, because of hardware revolution, people can obtain much more powerful computers, and the common clock speed of Central Processor Unit (CPU) is running at least over 1 GHz. Moreover, the three dimensional (3D) technology and Graphics Processor Unit (GPU) has been introduced into the computer, which allows people to develop rich, colorful, and vivid games, but it will consume much more human resources and a larger budget.

In fact, a computer game is just a computer program and the difference between a computer game and a normal computer program is only that a computer game can entertain people. Thus the development process of a computer game is exactly the same as that of a common computer program.

One typical software development process is called the V model, shown in Figure 2. By quickly reviewing this concept, the developer needs to realize that before coding, there is much work to be done. The requirement analysis is always uppermost in the development process, and it is the cornerstone of a successful project.

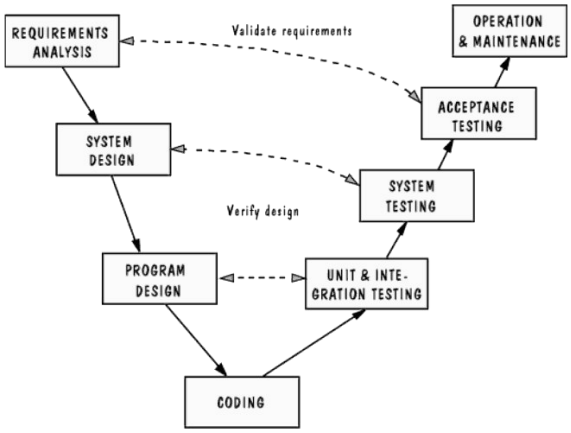

Figure 2. The V model concept. Reprinted from Pfleeger, Software engineering: theory and practice [8, 53]

The role of the requirement analysis is defined in the basic development rules that the developer should follow, and the analysis pointes out the key elements of the whole development work. The second important issue is testing, which means that the developer should prove the code is strong enough in this development scope. As Figure 2 illustrates, after coding, the verification design must be done at each phase including unit testing, system testing and acceptance testing.

The goal of unit testing is to take a small piece of the code that is responsible for enabling some very specific functionality within the software being developed, and to test it to ensure that it behaves exactly the same as the definition of the unit under various conditions [8, 67-69]. This approach allows the developer to test internal parts of the software that are not typically exposed directly to the end user. Integration testing is a logical extension of the unit test. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested [9, 53].

The objective of system testing is to establish confidence that the software will be accepted by its users, i.e., that it will pass its acceptance tests. During system testing, the functional and structural stability of the system will be demonstrated, as well as nonfunctional requirements such as performance and reliability. The objective of acceptance testing is to confirm that the software meets its business requirements and to provide confidence that the system works correctly and is usable before it is formally "delivered" to the end user. [9, 59-73]

4 GAME DESIGN

4.1 Game concept

The game is named TankWar, a short name from the title of the project: Tank War Game Application with the Active Floor. TankWar is an Elsi-floor-based shot computer game, which allows two users to play online during one game period. The role of the active floor is a platform which can detect the user's movements in the game period. The goal is simply to prove that the active floor can be used in the entertainment field. The mobile phone will be implemented as a control panel for the user, and the user can setup the game profile before the play.

The game content of TankWar is simple, and by default, the user has zombie tanks. The zombie tank will attack the enemy's wall automatically, and to prevent being hurt the wall, the zombie tank attacks the wall, and the user will control a cannon to shoot the enemy's zombie tanks. The movement of the cannon is corresponding to the movement of the user on the active floor. The fire instruction of the cannon will be given via the mobile phone, and the user can switch to other weapons on the mobile phone in the game, such as super missile, and call for more zombie tanks.

The super missile can impact a larger hurt on the player wall, and the user can obtain a super missile by hitting a quantity of zombie tanks. If the user loses some zombie tanks, the game will allow the user to call for more tanks into the game. The two users actually compete with time, and if one can destroy the other player's wall first, the one will win the game.

4.2 Game logic

First, the user will use the mobile phone to setup the profile, and then send a registration request to the game server. The registration for the user can be either successful or failed, and the result will be delivered back to user displaying on the phone screen. If registration is successful, the user needs to stand on the active floor for a while, in order to synchronize with the game server. The reason to do that is that the game server must realize the user is ready for playing. If the two users both are ready, the game is started immediately. After the game period, the two users quit from the game server. The way to play the game is demonstrated in Figure 3.

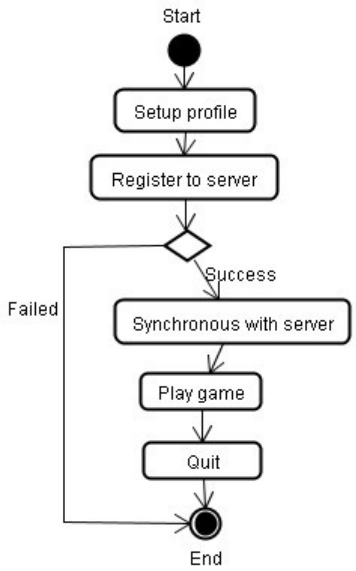

Figure 3. Flow chart, TankWar logic from the user point of view

The game is running on a normal personal computer as the game server, and there is a big screen or projector for displaying the game output. Because the mobile phones will carry out the control panel functionality, they need the two-way communication with the game server. The active floor only sends floor data via the floor server to the game server.

The application consists of the active floor, the mobile phone and the game server itself, and the architecture is described in Figure 4. To develop this simple game application TankWar, the Java programming language was used for all the necessary parts of the game, because it is a convenient, mature and powerful programming language. The Java programming language is an object-oriented high-level programming language designed and developed by Sun Microsystems originally in 1995 [10, 3].

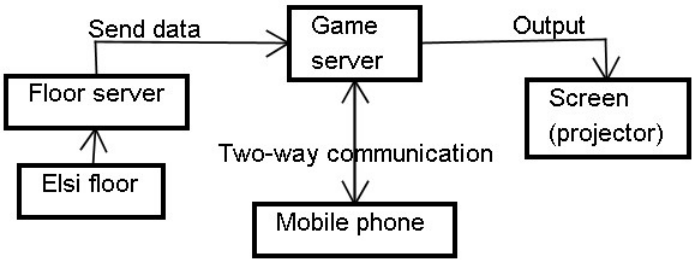

Figure 4. Technical architecture of TankWar

Under Java Platform there are four important members involving the Java Virtual Machine (JVM), Java Platform Standard Edition (Java SE), Java Platform Micro Edition (Java ME) and Java Platform Enterprise Edition (Java EE). The orientation of each member is totally different. JVM provides the Java runtime environment and allows the Java executable code to run under any operating system. Java ME focuses on mobile device development and Java EE is mainly used to web server level programming. Java SE is the main member of the Java platform, and it provides a variety of ready-to-use functionalities including GUI and networking. [10, 5]

5 IMPLEMENTATION OF GAME SERVER

The game server side is the core part of the game, and it contains the main game logic. As TankWar required a big screen for displaying the output, obviously, the game should have the Graphical User Interface (GUI) support. Therefore, Java SE will be a suitable option to achieve this.

5.1 Graphical frame and double-buffering

In the Java SE libraries, there are several packages to support the GUI, and normally package *java.awt* will be the first choice. This package contains all of the classes for creating user interfaces and for painting graphics and images [11]. While the program is executed, the GUI will be built up from the beginning. Hence, the main entry point is the graphical frame. The layout of the GUI for TankWar is described in Figure 5. The left and right side of the graphical frame are war zones for player 1 and player 2. The middle of the frame contains two blocks, and both of them are used for displaying players' information and game statistics.

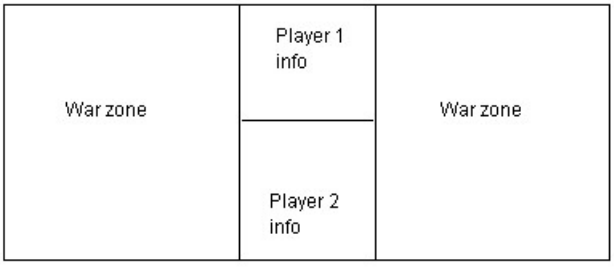

Figure 5. Basic layout of the GUI

As TankWar is a shot game, there will be animation of objects such as fire, movement and explosion. The basic idea of animation is to draw every object several times per second, and if an object has a little change, the player would observe that. To make the frames be refreshed frequently enough, multi-threading techniques will be introduced.

Generally, a computer program only has one thread running at a time, and as the operations are executed one by one, the program will exit if all the operations are completed. The Java Platform multi-threading concept allows the program to have several threads running concurrently, so that the program could distribute the tasks to different threads separately. In Java, there are two ways to implement multi-threading. One is to implement the interface *Runnable* and the other one is to declare a class to be a subclass of class *Thread*. Both two ways require overriding the *run()* method of the class [11]. In case of TankWar. This multi-threading concept is frequently used in the GUI and networking.

Because TankWar requires drawing objects frequently, there is a blink problem. The reason for that is that the draw instruction will draw an entire picture or object directly to the screen, pixel by pixel or line by line. By default, the draw instruction will read an object and draw it immediately. The time required for reading an object depends on the size of the object, and in the case of meeting large enough object, the reading time will be longer. Hence, it can easily happen that the screen is blinking.

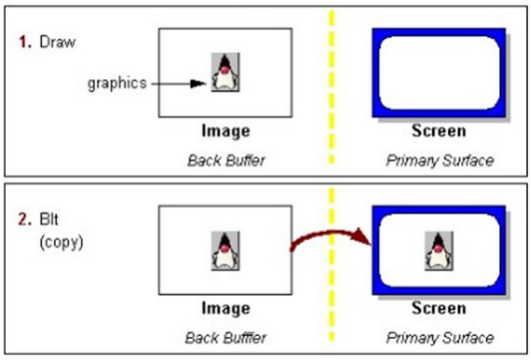

Figure 6. Double-buffering technique. Reprint from Java official Website, The Java™ Tutorials[12]

To avoid the blink problem, the double-buffering technique was implemented. The concept of double-buffering is shown in Figure 6. An object is first drawing on a back buffer image, and then the back buffer image is drawing on the screen.

After that there is no blink problem any more on the screen. The screen surface is commonly referred to as the primary surface, and the offscreen image used for double-buffering is commonly referred to as the back buffer. [12]

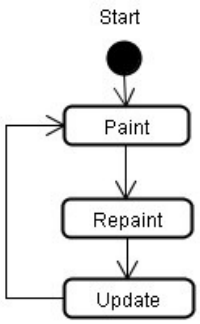

Figure 7. The implementation of double-buffering technique in an infinity loop

To build up a graphical frame, the Java SE package *java.awt* provides three methods for drawing, and they are *paint()* method, *update()* method and *repaint()* method. By default, after calling the *paint()* method, if the *repaint()* method is invoked, JVM will call the *update()* method first to clean the screen, and then JVM will call the *paint()* method again [13]. In this case, the *update()* method is re-implement (override) to achieve the double-buffering technique and the *repaint()* method is located in an endless loop. The implementation of this technique in an infinite loop is described in Figure 7. The sample code of this section is shown in Appendix 1.

5.2 Game components and Java reflection

There are nine main component objects to be created for TankWar, and they are *cannon*, *cannon missile*, *cannon missile explosion*, *tank*, *tank missile*, *tank missile explosion*, *super missile*, *super missile explosion* and *block wall*(player wall). By analyzing the objects' behavior in an object-oriented way, the objects can be defined to have similar method features in TankWar, such as 'draws itself', 'move', 'fire', 'collides with tank' and 'collides with block wall'.

Table 1 represents the behavior of each object. Based on Table 1, it is easy to define methods in each class, and the most important is that the table will enable to manage code for convenience in later implementation. Almost every object has a collision situation with another object, and in Java SE libraries, there is a mechanism to determine the graphical object's collision. First, the shape of every graphical object can be covered in an exactly same rectangle. There is a method called the *intersects()* method (in package *java.awt.Rectangle*) which allows to compare two rectangles. Hence, the implementation of those objects may contain this feature.

Table 1. The behavior of each object in TankWar

Object\ Behavior	**Draw**	**Move**	**Fire**	**Collides with tank**	**Collides with block wall**
Cannon	*	*	*		
Cannon missile	*	*		*	*
Cannon missile explosion	*				
Tank	*	*	*	*	*
Tank missile	*	*			*
Tank missile explosion	*				
Super missile	*				*
Super missile explosion	*				
Block wall	*				

Except the object *block wall*, every object will have one or several images to describe itself, so that, while the repaint thread is running, the animation is created automatically and can be seen on the screen. As already mentioned in chapter 5.1, when the *repaint()* method be called, every time the image will be read first, and if the animation is required to play smoothly, there is a way to load the image into the computer memory before the object is created from the hard disk, that is, the Java reflection technique.

The Java reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in JVM [14]. The definition of the class *Class* is that the instances of the class *Class* represent classes and interfaces in a running Java application [11]. An instance of the class *Class* contains the information of a compiled Java file (with extension .class). Each class in a Java application is loaded by a class loader and an object that constructs a *Class* object from Java bytecode[15, 121]. JVM will take care of this operation, and after that JVM will search main entry point among class objects and will start to execute. Those operations will be executed before any object is created, which is an important feature of the Java programming language.

For TankWar, an instance of the class *Class* will invoke one method from the class loader *getResource()* method and that will enable JVM to know the place of the images. To be able to load the images in advance, the help tool Toolkit class in the package *java.awt* needs to be used. It allows the instruction to create the image based on the data source. That will enable TankWar to implement colorful images for each object. The sample code of this section is given in Appendix 1.

5.3 Behavior of zombie tank

As the game concept defined, the zombie tank can not be controlled by the user. It will move and fire automatically by program operations. Hence the behavior of a zombie tank needs to be implemented. Without considering implementing artificial intelligence for the zombie tank, there is a way to achieve the performance of the zombie tank containing a little intelligence, and that is using a random method. TankWar is a two dimensional game, and a zombie tank can have nine directions for movement, such as left, upper left, up, upper right, right, lower right, lower left, down and stop. Hence, an enumeration was defined for the directions.

In implementation, by using the class *Random* from the Java SE libraries, a random number was obtained, and then the conditions for movement and fire were implemented. The direction of a gun on zombie tank also had to be considered, and the simple way was to set the direction same as the direction of movement. Finally, a zombie tank can move and fire itself. Moreover, while the zombie tank collides with another zombie tank or block wall, the behavior of that probably is to stay at the previous point. Every time the previous movement of the zombie tank is recorded, it will be compared with the new movement. The sample code of this section is shown in Appendix 1.

5.4 Defining message structure

Before implementing networking of TankWar, the message structure of communication should be defined. In TankWar, in addition to the registration service, there are two other network communications including floor information receiving and control panel information receiving. The messages of these two parts are simple and easy to decode. TankWar is a simple game, so that it is not necessary to design a complex message structure. The common way would be to define a number of digits (1010101…) as a string message for communication. The advantage of this solution is that it is easy to debug, maintain and extend.

Because two users will play with active floor at the same time, there may be confusion if only one type message is used, and thus the base of the message identity should be different indicating different players. The base of player 1 will be 100, and the base of player 2 is 200. As the real message needs to be defined from the floor side, at this moment, the message can be expressed as 1XX and 2XX. Additionally, TankWar requires the user to synchronize with the active floor before the play. Hence, it is defined that if the active floor sends only a base message, it means the message contains nothing but only sync information. If the active floor sends 1XX or 2XX, that means a message for the cannon movement.

Therefore, the message structure has been defined as shown in Figure 8. For the control panel message, it will be implemented in the same way, but only be using a base 1000. During the game period, the message of call three more tanks is 1001, and the message of launch super missile is 1010. The cannon fire message is 1100, and the last one 1111 is nothing but end message. After having defined the message structure, the method of the decode message is convenient using basic mathematical calculation. The example code of this section is shown in Appendix 1.

Message from the Elsi floor

Message	**100**	**200**	**1XX**	**2XX**
Meaning	Synchronize Player 1 base	Synchronize Player 2 base	Player 1 cannon movement	Player 2 cannon movement

Message form the control panel (mobile phone)

Message	**1000**	**1001**	**1010**	**1100**	**1111**
Meaning	Base	Call three more zombie tanks	Launch super missile	Cannon fire	End message

Figure 8. TankWar message structure

5.5 Game registration and networking

In the TankWar game server, there are three different networking classes. The first one is called the *RegistrationServer* class, playing the role of managing game registration. The second one, the *FloorInfoReceiver* class, is in charge of receiving data from the active floor. The last one is *PlayerListener* class that takes care of data receiving from the mobile phone during the game period.

TankWar only allows two users online playing the game in one game period, and the way to guarantee that only two users are involved is to require registration. The principle of *RegistrationServer* is to establish a listener on a specific Transmission Control Protocol (TCP) port. If there is a registration request coming, and it is checked whether the player listener User Datagram Protocol (UDP) port is available.

The principle of *RegistrationServer* is shown in Figure 9. There are two player listener ports in total by default. If available, distribute the port to the client, and otherwise, reject the registration request. After the playing game, the game server will recycle and reuse the player listener port. The reason for the implementation of the TCP listener for *RegistrationServer* is that at this phase, the user will send not only the registration request, but also the profile and game setup via networking.

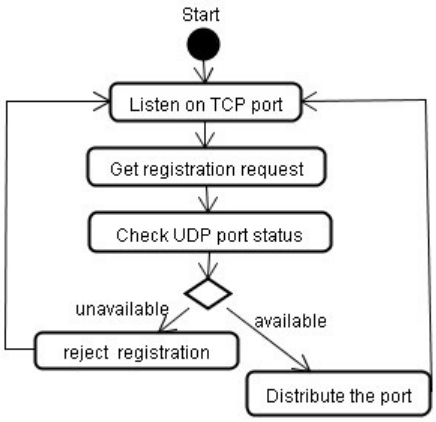

Figure 9. Principle of game registration

The communication needs to be reliable and interruptions need to be avoided. *FloorInfoReceiver* and *PlayerListener* will use the UDP for networking communication, because during the game period, there is no need to guarantee that every data package is received. Moreover, if the network is broken down, it is easy to reconnect each other via UDP, but not for TCP. All those three networking classes need to enable the multi-threading in an infinite event loop, and the common way is implemented by the *Runnable* interface. The Java SE has strong libraries supporting network communication, and the example code (only show the overridden *run()* method) of this section is given in Appendix 1.

5.6 Sound and singleton pattern

Typically, a game should have sound effect support in addition to the game concept and animation, and a suitable sound will be able to affect players. TankWar is a shot game, and considering the background sound, a fast-paced sound would be a good choice. With limited resources and budget, it was impossible to create or purchase a sample sound for TankWar in the development period, and also the copyright issues did not allow the casual use of any sound sample. On the Internet there are abundant sound resources without copyright, and for study purposes, it is possible to download some of them and embed them into TankWar. The quality of sound samples was not satisfactory, and so the samples were only used for testing.

Changing a design pattern for sound implementation is a moderate challenge. In Java, when one needs to create an object instance from class, normally, a *new* operator will be used, and a string example is shown as below.

String something = new String();

This way is suitable for the most common cases. If there is only one set resource to handle the operation, the program needs to reuse that for all the time, but the program can not hold a reference of the resource from beginning to end. In the case of such a situation, one solution would be to implement the singleton pattern for the program. The singleton pattern is a pattern that ensures there is one and only one instance of an object, and that it is possible to obtain global access to that one instance [16, 38].

In TankWar, the sound will be reused through out the game before play, during the game period and after the play. It is possible that in the program, only one instance will deal with all the related work. For singleton pattern, it can be difficult to inherit a Singleton, since this can only work if the base Singleton class has not yet been instantiated [16, 42]. The example code of this section is shown in Appendix 1.

6 IMPLEMENTATION OF ACTIVE FLOOR

As mentioned in chapter 2, the Elsi floor is used as the active floor. The size of the Elsi floor used is approximately 1.8 m * 3.6 m, and all the necessary hardware and software are already installed and configured completely. While the game is required to be synchronized with the floor, the zone division is shown in Figure 10, and by default, the first registered user will be required to synchronize in zone 1 and play in zone 2. The second user will need zone 3 for synchronizing and zone 4 for playing. The size of the zones is defined to be almost equal.

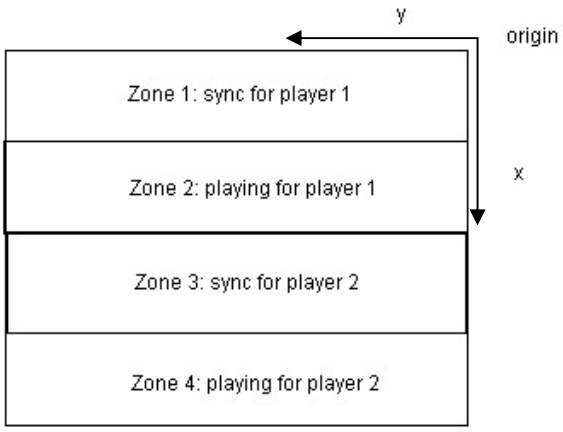

Figure 10. Zone division of active floor for game

For the Elsi floor, on the Elsi server, it is needed to define and implement the sent floor message program for TankWar. By default, the programming language used in the Elsi floor server is Python. Hence the implementation for TankWar game module will also be carried out with Python, because that makes all systems more compatible. Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with a very clear syntax [17].

The techniques used in the ELSI technology are extremely complicated. Simply, if floor is active, it means someone is standing or walking on the floor, the Elsi floor will know that, and based on the defined physical zones, the floor system will detect whether people are on the zone. With limited development time, the easiest way to implement a module for TankWar is to modify the sample code. The hints in the sample code help to achieve that goal.

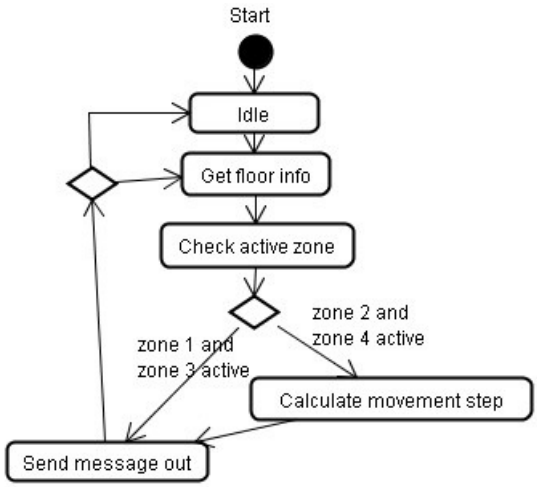

Figure 11. Principle of game module on active floor

The principle of the game module is figured out and demonstrated in Figure 11. Because the active floor obtains floor information passively, the floor will enter idle status if the floor is deactivated. The floor will gain information immediately if the floor becomes active. If either zone 1 or zone 3 is active, the program will send the synchronizing message to the TankWar game server, but if either zone 2 or zone 4 is active, the program will calculate the movement step first and then send the movement message to the game server.

Physically, the maximum width of each zone is approximately 0.45 m, and that is close to the width of a piece of the active floor. The maximum length of each zone is 3.6 m, but to avoid negative impact of electronics on the floor, for movement calculation, the maximum effect length is set to 2.9 m. The minimum effect length is 0.2 m.

If the minimum width between the two feet of a player is around 0.22 m, then by using mathematical calculation, the maximum step of the movements would be 12. According to the message definition, the movement message was set up. For instance, the movement message of player 1 is from 101 to 112. The sample code of this section is shown in Appendix 1.

7 IMPLEMENTATION OF GAME CLIENT

On the game client side, the target is to implement a mobile phone as the TankWar game control panel. The basic functionality includes a game setup, game fire action detection, and communication with the TankWar game server. In this case, networking support is also needed. The programming technology choice for developing The TankWar client would be Java ME.

Java ME focuses on developing an application on the limited resources of mobile devices such as Personal Digital Assistants (PDA) and mobile phones [18]. Java ME has the Mobile Information Device Profile (MIDP) and is aimed at providing a solid Java platform for developing applications to run on devices with limited memory, processing power, and graphical capabilities [18]. MIDP is commonly supported by modern mobile phones.

7.1 MIDlet

A MIDlet is an application that uses the MIDP of the Connected Limited Device Configuration (CLDC) for the Java ME environment [19]. Briefly, a MIDlet is the application entry point. All applications for the MID Profile must be derived from a special class, MIDlet. The MIDlet class manages the life cycle of the application. It is located in the package *javax.microedition.midlet* [19].

Interactive applications can get access to the display by obtaining an instance of the *Display* class. A MIDlet can get the class *Display* instance by calling the *Display.getDisplay()* method. The *Display* class provides a *setCurrent()* method that sets the current display content of the MIDlet. The actual device screen is not required to reflect the MIDlet display immediately—the *setCurrent()* method just influences the internal state of the MIDlet display and notifies the application manager that the MIDlet would like to have the given Displayable object displayed [20].

A MIDlet can exist in four different states: loaded, active, paused, and destroyed, which are shown in Figure 12. When a MIDlet is loaded into the device and the constructor is called, it is in the loaded state. This can happen at any time before the program manager starts the application by calling the *startApp()* method. After *startApp()* method is called, the MIDlet is in the active state until the program manager calls *pauseApp()* method or *destroyApp()* method; *pauseApp()* method pauses the MIDlet, and *desroyApp()* method terminates the MIDlet. All state change callback methods should terminate quickly, because the state is not changed completely before accomplishing the executed method [20].

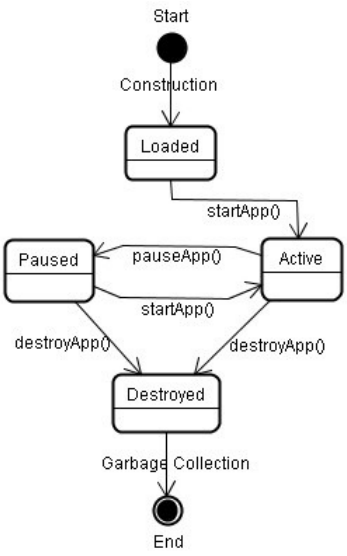

Figure 12. Life cycle of a MIDlet. Reprint form Kroll M. and Haustein S. J2ME Application Development [20]

The application programming interface (API) of the MIDP user interface is pure GUI API, and it is divided into a high- and low-level API. The high-level API provides input elements such as text fields, choices, and gauges, and all elements are ready-to-use. The low-level API contains several members including graphics, font, and image. All the members should be programmed by the developer. As TankWar client requires text input and graphics display, and both high- and low-level user interface are needed. The *setCurrent()* method will be used for switching the actual content display on the phone screen between high- and low-level user interface.

7.2 High-level user interface

For high-level user interface, Screen class is the common superclass of all high-level user interface classes. Because Screen class is an abstract class and extended from Displayable class, an object of Screen class can be displayed on the phone screen by invoking *setCurrent()* method of Display object. On the Screen object, text field, image and item choices group can be placed by using *append()* method. The welcome prompt of the client program will show only the game name and an image of the tank, and then the registration command at the bottom of the phone screen. By pressing the registration command, the screen will turn to registration.

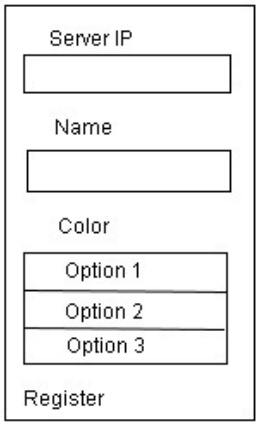

Figure 13. High-level user interface for input

The high-level user interface for TankWar client is simple, and it only needs several text fields for the input game server IP and the user's name, as well as one choice group for choosing the color in the game. The order of those components is demonstrated in Figure 13. The text field for input IP may break into another four text fields, because that would be much easier to verify the IP. A 'register' command is needed to be added at the bottom, so that while the player presses this command, the program will handle and forward all the input to the networking model.

After a successful registration, the screen will display a built-in message that indicates user needs to synchronize with the active floor as shown in Figure 14. At bottom the ‘play’ command is placed. After the ‘play’ command has been pressed the screen will turn to low-level user interface for playing. If the registration failes, the phone screen will show an alert message and then go back to the program prompt.

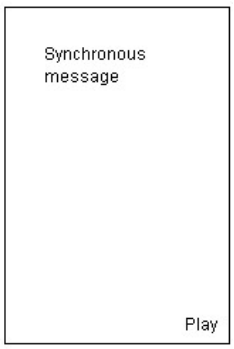

Figure 14. High-level user interface after successful registered

7.3 Sound and low-level user interface

The game sound affects two places: one is the game prompt and the other is the fire key event occurred. Because the memory of the mobile phone is limited, every time a sound event occurs, the program will load from the disk to reduce the occupied memory. Additionally, playing the sound will need multi-threading for support, as otherwise the operation of the program would probably be stuck.

In Java ME, the low-level user interface is essentially composed of class *Canvas*. The supporting classes are used within an instance of class *Canvas* to create different visual effects. The class *Canvas* is a base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display.

The class *Canvas* provides the developer with methods to handle game actions and key events. The methods are also provided to identify mapping of keys to game actions [21]. The key events are reported with respect to key codes, which are directly bound to concrete keys on the device, and this may hinder portability.

The class *Canvas* requires applications to inherit it in order to use it. The *paint()* method is declared abstract, and so the application must provide an implementation in the subclass. At this phase, the items will be drawn up including the name of the game, the name of the player, the color of the player choice, and the play zone number. Those items are static and drawn only once without update, as shown in Figure 15. In order to avoid many phone keys being affected during the game period, the smart way is to only enable the fire key as the game fire action is triggered. By default, TankWar has three different weapons for players, hence, besides the fire key, at least another key needs to be used for switching the weapon.

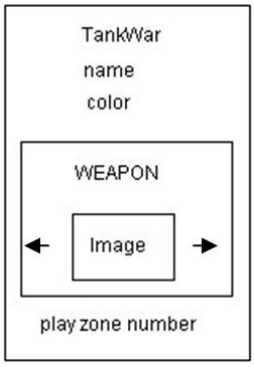

Figure 15. Low-level user interface for play game

Keeping in mind the normal behavior of people, using thumb presses with the phone is frequent, so that it is possible to enable the key which is close to the fire key. It is better to enable both left and right key which are close to the fire key. The low-level user interface is shown in Figure 15, and the two arrows indicate the switching weapon. From the Internet, it is easy to obtain small size pictures, and those pictures will be drawn up on the canvas as the weapon icon.

When every time the weapon is switched, the screen shall be refreshed, and like in the previous game server case, multi-threading is needed to support repainting the canvas in an endless loop. It is possible to press either the left key or the right key for switching the weapon in a loop, and that will enable the player to press only two keys for the game. In class *Canvas*, there are several key event monitor methods and *keyReleased()* method will be the first choice for TankWar.

Because there are only three weapons, an array is defined containing numbers 1, 2 and 3 corresponding to each weapon. The logic of switching is to check the current weapon every time first, and if the left key is pressed, the current weapon will be shifted left by one and the corresponding array value will also be shifted left by one. If the left shift weapon reached the array value is the first element, and then the array value will point to the last element. For the right shift case, the logic is similar. The switching weapon will perform as scrolling on the screen. The sample code of this section is given in Appendix 1.

7.4 Networking

Networking on the TankWar client is in charge of two types of communications including registration and game action data transmissions. In the TankWar game server, two type communication modes were implemented, and they are the TCP and the UDP socket connections. In a similar way, on the client side, a corresponding socket connection is needed. In Java ME libraries, different format connections are supported. However according to the rule of the Java ME developer Sun Microsystems, to be enable communication with any socket connection for the Java ME MIDlet application, a commercial digital signing is required [21].

With the limited budget of the project, it was impossible to obtain a signing for TankWar. Fortunately, the Java ME MIDlet application supports the HTTP connection by default. There is an exclusive way to implement network communication via the HTTP connection. The logic of game registration communication is, at first, to establish the HTTP connection, and then send the player setup to the game server.

After gaining a response from the game server, the program decides on the response whether the registration is successful or not. If the registration is successful, the program will prepare to enter the game period, or otherwise will return to the game promote. During the game period, when it is needed to transmit data, the program will establish HTTP connection, and then send the data out immediately. Both the two HTTP connections need multi-threading support, and an infinite loop is required for the game action sent, but not for game registration. After implementing the HTTP networking, there is the question that something should be done, so that the game server could receive information from the client. This will be covered in section 7.5.

7.5 Servlet as a middleman

As mentioned in chapter 7.4, the way that allows the game server to receive client data is to build up a so called middleman, that is, Servlet. A Servlet is a Java programming language class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model [22]. The concept of Servlet belongs to Java Web Technology, and Servlet is a component of Java EE.

The middleman architecture is demonstrated in Figure 16, and Servlet is in charge of parsing the client's request, processing it, and returning the results back to the client. In TankWar, between the game server and the game client there are two different communications, and to simplify the Servlet functionality, the communications are combined into one Servlet.

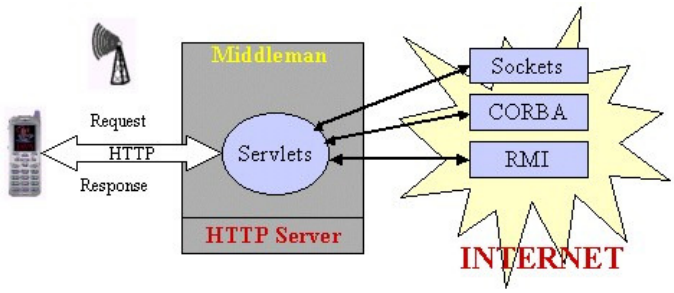

Figure 16. Middleman architecture. Reprint from Mahmound Q. Advanced MIDP networking, Accessing Using Sockets and RMI [23]

The easy way to do this is to define a request code on the client side and Servlet. Before sending real data from the client, the program binds a request code and sends all data out. While Servlet receives the data, first, the request code is checked, and if a match one is found, the related operations will be executed. The implementation of the Servlet is simple, and it only conveys information from one side to another side as a bridge. The sample code of this section is shown in Appendix 1.

7.6 Application server

After implementing the Servlet, the next step to be carried out was to establish an application server which would allow Servlet to start the service for the game server and the game client. Hence, a professional application server had to be introduced, that is Apache Tomcat or Tomcat. Tomcat is open source software developed by Apache Software Foundation. It is a feature-complete Servlet container, and also it is Sun Microsystems reference implementation of a Servlet container, which means that Tomcat's first goal is to be 100% compliant with the versions of the Servlet and the JavaServer Pages (JSP) API specification that it supports [24, 3].

The Tomcat web server is similar to the Apache web server, but Tomcat is a stand alone web server. Hence, there is no need to build up a separate Apache web server. To deploy the Servlet under Tomcat, there are several things that need to be done as follows. Compile the Servlet into class file, because Tomcat can only load the class format Servlet. Create a directory named *tankwar* and place under *webapps* directory of the tomcat. Create a directory with the name *WEB-INF* under *tankwar* directory. Create a directory called *classes* and an empty file with the name *web.xml* file under *WEB-INF* directory. The sample code of this section is given in Appendix 1.

8 TESTING

As TankWar is a software application, after implementation, the testing tasks should be done. According to the concept of the classic V model, the step of testing would be unit testing, integration testing, system testing and acceptance testing. With limited time and resources, it was impossible to apply a professional testing approach for the testing task, and thus the traditional way was accepted.

8.1 Unit testing

The target of unit testing is to verify that every class or function works correctly. On the game server side, apply a *main()* method for every class, and in the *main()* method invoke all defined methods with reasonable parameters. Observe every program's output both on the console and the GUI. Sometimes, in the program, two or three classes have a combination relationship, and to avoid such interference, disable that effect code. For the active floor side, write test program with a number of predefined data, and then by executing the test program, observer every program's output on the console.

For the game client, using an emulator separately test, the performance of high- and low-level user interface. The networking model was tested with the applied extra Servlet, and Tomcat server was started to observe whether the program throws exceptions or not. After three parts testing, the result of this step was positive with no errors detected. Before moving to the next phase, all the test methods were disabled and the system code status was restored back to the one before testing.

8.2 Integration testing

The goal of integration testing is to make sure that every part is functioning smoothly. For the game server, the networking model is isolated and all its effects are disabled, and then executed by the main program. The output both on the console and the GUI are checked. The outcome of the testing will be able to verify without network communication, whether all the GUI components are working properly or not.

On the active floor side, import the game model into the package of the Elsi server, and by applying the sample data resources and starting the server, observe the output. For the game client, the networking model is isolated. Pack all the necessary classes into a jar file, and then install it into Nokia N82 mobile phone. After installation, run the application, and operate the phone, observing the phone screen.

After testing, on the game server side and the active floor side no errors were detected, but the game client side had a display problem. The problem was the screen switched to the low-level user interface, and the location of every object was incorrect. The reason for that was the original location of the objects was not fixed for the N82 phone screen. The objects' location was redefined, and then the problem was resolved. Before moving to the next testing phase, the system code status was restored back to the one before the testing.

8.3 System testing

The objective of system testing is to guarantee, before acceptance testing, that the whole system is functioning correctly and is stable. To do that, the active floor side was applied by the sample data resources and the game client was deployed into two N82 mobile phones for testing. All three sides allowed the networking model to work. Because the mobile phone requires a wireless network connection to communicate with the game server, a LINKSYS® wireless router with default wireless environment setup was applied.

The order of the execution was the game server, the active floor and the game clients. At the beginning, the game was playable and the performance was smooth. After the game period, both two game clients quitted the application, and then if two clients restarted to register into the game, it failed. After reviewing the program code, the problem was found. The problem was that the network transmitting was passive for the game server, and the game server could not know when the data was coming, so that it would block the operation until the data was received.

The problem could be solved by implementing another communication threading. The purpose of the threading is to enable the game server to post a message to inform all networking components that the game is over. After this problem was resolved, the system code status was restored back to the one before the testing.

8.4 Acceptance testing

The goal of acceptance testing was to examine it TankWar was ready to be deployed and could start to provide its service to the public. In acceptance testing, the real active floor was used for testing and this was the only difference compared to system testing. The game period was operated in a total of four times, and each period lasted approximately three to five minutes. No error impact on TankWar was noticed during the testing period.

The final result is that TankWar is a successful project, and the game is ready to be deployed. At last, for maintenance purposes, all necessary design and implementation of TankWar was documented with detailed comments, and the user guide of TankWar can be found in Appendix 2.

9 RESULTS AND DISCUSSION

The results of every testing present some interesting arguments for discussion. As a recall for the purpose of this project, the aim was based on the existing active floor, to develop a computer game for entertainment interactive people. The purpose of all the tests was to prove that a mobile phone could be a game control panel.

The development of TankWar followed the classical software engineering processing V model, and after implementation, several tests were carried out. This approach always reminds the developer to improve the performance of software during the development period, and also proved that the total testing time is much longer than implementation. This approach can be used for any software developer training.

By developing TankWar, it was proved the Esli floor can be applied to other fields, besides a nursing home discussed in chapter 1, such as entertainment. As a result, a sale manager from MariMils® Oy is interested in TankWar, and intends to expand their business along the idea of TankWar. However, the sampling rate of the Elsi floor has a physical limit, and that may discourage many applications.

This development work was an opportunity to study new concepts including design patterns, game and animation design. The most important point was to obtain experience on software development. After developing TankWar, the developer realized that the Java programming language is easy to study and achieve something except game applications, because the performance of the graphical objects was slow and not vivid. The multi-threading concept is useful for any network communication.

10 CONCLUSIONS

The goal of the project was to study the basic principles of the active floor by the ELSI technology, and to develop a computer game. The game created in this project is playable with the mobile phone, extendible and easy to deploy.

The outcome of the project was that the performance of the active floor TankWar game is matching the original design. The outcome proved that it is possible to extend the usage of the Elsi floor to the entertainment field. However, the results demonstrated that TankWar is not a real game without implementing artificial intelligence, which should be the core part of games nowadays. The results also proved that the mobile phone is able to play a role in the game as the control panel.

With limited time and knowledge, the project concentrated on a simple GUI application development containing game elements, such as basic animation design and game logic. The artificial intelligence probably will be implemented into TankWar in the future version. Finally, it is recommended to further develop the current implementation for improved user interface and to fix potential bugs. It would be a concrete point to reengineer and implement the game server of TankWar by using any other suitable game programming language.

REFERENCES

1 ELSI™ System [online]. MariMils Oy.
URL: http://www.marimils.fi/index.php?k=12150&x=1267884937.
Accessed 05 March 2010.

2 Elsi ElderlyCare [online]. ELSI TECHNOLOGIES.
URL: http://www.elsitechnologies.com/en.php?k=16419.
Accessed 05 March 2010.

3 Elsi ElderlyCare references [online]. ELSI TECHNOLOGIES.
URL: http://www.elsitechnologies.com/en.php?k=16387.
Accessed 05 March 2010.

4 The Art of Computer Game Design [online]. Washington State University; July 1996.
URL: http://www.vancouver.wsu.edu/fac/peabody/game-book/Chapter1.html.
Accessed 05 March 2010.

5 Winter D. Welcome to Pong-Story [online]. pong-story.
URL: http://www.pong-story.com/intro.htm.
Accessed 05 March 2010.

6 Russell S. History of Spacewar [online]. Maury Markowitz; 13 December 2001.
URL: http://www3.sympatico.ca/maury/games/space/spacewar.html.
Accessed 05 March 2010.

7 PDP 1 Handbook [online]. Digital Equipment Corporation; 1963.
URL: http://www.dbit.com/~greeng3/pdp1/pdp1.html.
05 March 2010.

8 Pfleeger L, Atlee. M. Software engineering: theory and practice. 3rd edition.
New Jersey, USA: Prentice Hall; 2006

9 Watkins, J. Testing IT: An Off-the-Shelf Software Testing Handbook.
Port Chester, NY, USA: Cambridge University Press, 2001

10 Eubanks, B. Wicked Cool Java. San Francisco, CA, USA: No Starch Press; 2005.

11 Java™ Platform, Standard Edition 6 API Specification [online].
Sun Microsystems.
URL: http://java.sun.com/javase/6/docs/api/.
Accessed 05 March 2010.

12 Double Buffering and Page Flipping [online]. Oracle Corporation.
URL: http://java.sun.com/docs/books/tutorial/extra/fullscreen/doublebuf.html.
Accessed 05 March 2010.

13 Painting in AWT and Swing [online]. Oracle Corporation.
URL: http://java.sun.com/products/jfc/tsc/articles/painting/index.htm.
Accessed 05 March 2010.

14 Java reflection [online]. Oracle Corporation
URL: http://java.sun.com/docs/books/tutorial/reflect/.
Accessed 06 March 2010.

15 Forman R, Forman N. Java Reflection in Action.
Greenwich, CT, USA: Manning Publication Co.; 2005

16 Cooper W. The design patterns Java companion [online].
IBM Thomas J. Watson Research Center; 1998
URL: http://www.patterndepot.com/put/8/JavaPatterns.htm.
Accessed 06 March 2010.

17 General Python FAQ [online]. Python Software Foundation; 26 October 2009
URL: http://www.python.org/doc/faq/general/.
Accessed 06 March 2010.

18 Mobile Information Device Profile (MIDP); JSR 37, JSR 118 Overview [online].
Oracle Corporation
URL: http://java.sun.com/products/midp/overview.html.
Accessed 06 March 2010.

19 MID Profile [online]. Sun Microsystems
URL: http://java.sun.com/javame/reference/apis/jsr118/.
Accessed 06 March 2010.

20 Kroll M, Haustein S. J2ME Application Development [online]. developer.com;
Sams Publishing; December 26, 2002
URL: http://www.developer.com/java/j2me/article.php/1561591.
Accessed 06 March 2010.

21 JavaOne™, Signing Java™ ME Applications and Signing Them in Java Verified [online]. Sun Microsystems. 4 June 2008
URL: http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-5682.pdf.
Accessed 06 March 2010

22 What Is a Servlet? [online]. Sun Microsystems. October 2008
URL: http://java.sun.com/javaee/5/docs/tutorial/doc/bnafe.html.
Accessed 06 March 2010

23 Mahmoud Q. Advanced MIDP Networking, Accessing Using Sockets and RMI from MIDP-enabled Devices [online]. Sun Microsystems. January 2002
http://developers.sun.com/mobility/midp/articles/socketRMI/.
Accessed 06 March 2010

24 Chopra V, Bakore A. Professional Apache Tomcat 5. Indianapolis, IN USA:2004

APPENDIX 1: SAMPLE CODE

Graphical frame and double-buffering section

```
// import all necessary classes form the package
import java.awt.*;

public class TankApp extends Frame {

       private static TankApp ta = null;
       private boolean terminatePaintTheard = false;
       private Image offScreenImage = null;

      public void paint(Graphics g) {
            // draw objects on screen
      }

       // override update method for double-buffering technique
      public void update(Graphics g){
            // if no such offScreenImage object, create one.
            if (offScreenImage == null){
            offScreenImage=this.createImage(FRAME_WIDTH, FRAME_HEIGHT);
            }
           // draw other objects on Image object offScreenImage

         // draw object offScreenImage on screen
          g.drawImage(offScreenImage, 0, 0, null);
       }

       public void launchFrame(){
            // create other graphical components
            // create the repaint thread and execute it
            new Thread(new PaintThread()).start();
       }

     // program main entry (start) point
     public static void main (String[] args){
            ta = new TankApp();
            ta.launchFrame();
       }
```

```
    // a thread for repaint frame
     private class PaintThread implements Runnable{
          // override run method
          public void run() {
               while(!terminatePaintTheard){
                    repaint();
               }
          }
     }
}
```

Game components and Java reflection section

```
// code is wrote at field place before class constructor
private static Toolkit tk = Toolkit.getDefaultToolkit();
private static Image img = null;

// static code block
static{
        img = tk.getImage(OBJECT.getClassLoader().getResource(URL));
}


// similarity attribute interfaces
void draw();
void move();
Rectangle getRect();
void fire();
boolean collidesWithBlockWall();
boolean collidesWithTank();
```

Behavior of zombie tank section

```
public enum Direction {
  LEFT, LEFT_UP, UP, RIGHT_UP, RIGHT, RIGHT_DOWN, DOWN, LEFT_DOWN,
  STOP
}

import java.util.Random;

public class Tank {

      public static final int XSPEED = 5;
      public static final int YSPEED = 5;
      private Direction dir= Direction.STOP;
      private Direction gun_dir= Direction.DOWN;
      private int x, y;
      private int oldX, oldY;
      private static Random random = new Random();
      private int step = random.nextInt(12) + 3;

      private void move(){
            this.oldX = x;
            this.oldY = y;

            // The calculation for movement is NOT accurate
            switch(dir){
                  case LEFT:
                        x -= XSPEED;
                        break;
                  case LEFT_UP:
                        x -= XSPEED;
                        y -= YSPEED;
                        break;
                  case UP:
                        y -= YSPEED;
                        break;
                  case RIGHT_UP:
                        x += XSPEED;
                        y -= YSPEED;
                        break;
                  case RIGHT:
```

```
                    x += XSPEED;
                    break;
                case RIGHT_DOWN:
                    x += XSPEED;
                    y += YSPEED;
                    break;
                case DOWN:
                    y += YSPEED;
                    break;
                case LEFT_DOWN:
                    x -= XSPEED;
                    y += YSPEED;
                    break;
                case STOP:
                    break;
            }

            // judge gun's direction
            if(dir != Direction.STOP){
                gun_dir = dir;
            }
            Direction[] dirs = Direction.values();
            if(step == 0){
                 // condition for movement
                step = random.nextInt(12) + 3;
                int rn = random.nextInt(dirs.length);
                dir = dirs[rn];
            }
            step--;

            // condition for fire
            if(random.nextInt(24) > 18){
                fire();
            }
        }

         // while collides with other zombie tank or block wall
        public void stay(){
            x = oldX;
            y = oldY;
        }
}
```

Defining message structure section

```
public void decodeFloorMessage(String msg){
    int temp = Integer.valueOf(msg);
    int playerMsg = temp / 100;
    int playerCondition = temp - (playerMsg * 100);
    if(playerMsg == this.p_1){
        if (playerCondition == 0){
            // player 1 synchronize
        } else {
            // player 1 movement
        }
    } else
    if(playerMsg == this.p_2){
        if (playerCondition == 0){
            // player 2 synchronize
            }
        } else {
            // player 2 movement
        }
    }
}

public void decodeControlPanelMessage(final String msg){
    int temp = Integer.valueOf(msg);
    int playerMsg = temp % 1000;
    switch(playerMsg){
    case 1:
        // call 3 tanks
        break;
    case 10:
        // super missile fire
        break;
    case 100:
        // cannon fire
        break;
    case 111:
        // end message
        break;
    }
}
```

Game registration and networking section

```
// import all necessary classes form the package
import java.io.*;
import java.net.*;

public class TCPcommunication implements Runnable {
   private ServerSocket ss = null;
   private boolean terminateTheard = false;
   private final int MAX_CONNECTION = 2;
   private int connectionCounter = 0;

   public void run() {
         try {
              ss = new ServerSocket(SERVER_LISTEN_PORT);
         } catch (IOException e) {
              e.printStackTrace();
         }
         while (!terminateThread) {
              Socket s = null;
              DataOutputStream dos = null;
              DataInputStream dis = null;

             // check UDP port status
              if (connectionCounter < MAX_CONNECTION) {
                   try {
                        s = ss.accept();
                        dos = new DataOutputStream(s.getOutputStream());
                         // disturb the port
                        } catch (Exception e) {
                        e.printStackTrace();
                   }
                   try {
                        dis = new DataInputStream(s.getInputStream());
                         // program code
                         // player setup
                        connectionCounter++;
                   } catch (Exception e) {
                        e.printStackTrace();
                   }
              } else {
```

```
                    // program code
                     // reject the registration request
                }
                if (s != null) {
                    try {
                        s.close();
                        s = null;
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
}

public class UDPcommunication implements Runnable{
    private boolean terminateTheard = false;
    private DatagramPacket dp = null;
    private DatagramSocket ds = null;
      public void run() {
            dp = new DatagramPacket(BUF, BUF_LENGTH);
            try {
                ds = new DatagramSocket(UDP_PORT);
            } catch (SocketException e) {
                e.printStackTrace();
            }
            while(!terminateThread){
                try {
                    ds.receive(dp);
                    // program code
                    // handle receive data
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(ds != null){
                ds.disconnect();
                ds.close();
                ds = null;
            }
        }
}
```

Sound and singleton pattern section

```
public class GameSoundManager {
      // before constructor called, create the object
      private static final GameSoundManager gsm = new GameSoundManager();

     // private constructor
      private GameSoundManager(){
      }

      // static method
      public static GameSoundManager getInstance(){
            return gsm;
      }

     // program code
     // other methods
}
```

Implementation of active floor section

```
import socket

# host name and port number
c_host = str(HOST)
c_port = PORT

def sendGameFloorMessage(MESSAGE):

	# create a socket for UDP
	s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
	try:
		s.sendto(str(MESSAGE), (c_host, c_port))
	except:
		# exception
		pass
	#close socket
	s.close()

#	min		max
# step 1, 2, 3, ..... 11, 12
c_floor_minVal = 0.2
c_floor_maxVal = 2.9
c_floor_unitVal = 0.225

def calculateStep(playerNumber, location):
	y = location[1]

	if(y < c_floor_minVal):
		step = 1
	elif(y >= c_floor_minVal and y <= c_floor_maxVal ):
		step = int(y /c_floor_unitVal)
	else:
		step = 12

	msg = playerNumber + step

	return msg
```

Sound and low level user interface section

```
// monitor event when key released
protected void keyReleased(int keyCode) {
        int key = getGameAction(keyCode);
        switch (key) {
              case Canvas.FIRE:
                    weaponFire();
                    break;
              case Canvas.LEFT:
                    tatusLeftShift();
                    break;
              case Canvas.RIGHT:
                    statusRightShift();
                    break;
        }
}

// weapon left switch
private void statusLeftShift(){
            for(int i = 0; i < status.length; i++){
                  if(status[i] == getCurrentStatus()){
                        if(i == 0){
                              i = status.length - 1;
                        }else {
                              i--;
                        }
                        setCurrentStatus(status[i]);
                        break;
                  }
            }
      }

   // weapon right switch
      private void statusRightShift(){
            for(int i = 0; i < status.length; i++){
                  if(status[i] == getCurrentStatus()){
                        if(i == status.length - 1){
                              i = 0;
                        } else {
                              i++;
```

```
                    }
                    setCurrentStatus(status[i]);
                    break;
                }
            }
        }

        // weapon fire
        private void weaponFire(){
            switch(getCurrentStatus()){
                case 1:
                    //  play cannon sound
                    // cannon fire
                    break;
                case 2:
                    //  play super missile sound
                    //  launch super missile
                    break;
                case 3:
                    //  play thank sound
                    //  call three more zombie thanks
                    break;
            }
        }
```

Application server section

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">

      <servlet>
         <servlet-name>TankWarBridge</servlet-name>
         <servlet-class>TankWarBridgeServlet</servlet-class>
      </servlet>

      <servlet-mapping>
            <servlet-name>TankWarBridge</servlet-name>
            <url-pattern>/TankWar</url-pattern>
      </servlet-mapping>

</web-app>
```

APPENDIX 2: TANKWAR GAME USER GUIDE

Before Play

The game server is running, and there is an available place to play. A standard smart phone is needed with the K Virtual Machine (KVM) supports, e.g. Nokia Nseries. Download and install the client side application of the game on the smart phone, and if you meet any security warning, skip it. After that, the client side application should be available in the application list.

Player Setup

When you have started the client application, on the phone screen there will be a front form that contains the game name and a tank image as shown in Figure 1. Press ‘Exit’ to quit this application, and by pressing ‘Options’, you will get two choices to select. One is starting the game and the other one is the ‘Help’ option which contains the basic information of the application.

Figure 1. Front form of the client application

After you having pressed ‘Start game’, the application will require the user input the game server IP address, and every text field will only allow to give 3 digits from 0 to 255. The user can enter his/her name (nick name) and choose one type of color for his/her game objects. Figure 2 and Figure 3 are shown as example of this.

When the player setup step is completed, the user can press ‘Register’ to send a registration request to the game server and wait for the server’s response.

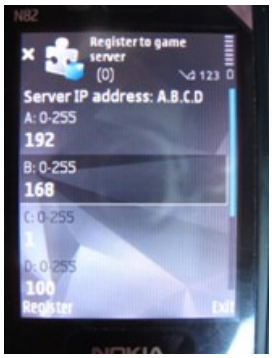

Figure 2. Input IP address

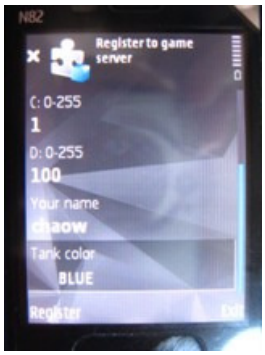

Figure 3. Input user name and choose color

Synchronization with the active floor

The registration can either succeed or fail, and the reason causing registration to be failed may include no place any more (there are already two players in the game), giving an incorrect server IP address, or a network problem. If in the case of meeting a failed registration, try it again. If the registration is successful, the phone will display the information that requires the user enter one zone of the active floor for sync, as shown in Figure 4. The purpose of that is that the floor will inform the server that the player is ready to play the game. The total time for sync will be less than two seconds.

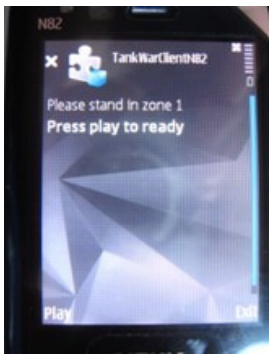

Figure 4. Game requiring to be synchronized

The zone division is shown in Figure 5, and by default, the first registered user will be required to be synchronized in zone 1 and play in zone 2. The second user will need zone 3 for synchronization and zone 4 for playing.

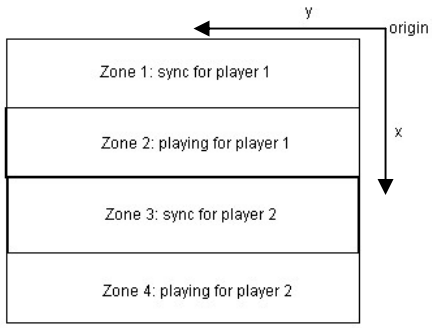

Figure 5. Zone division of the active floor for game

Play Game

After the floor's synchronized, the user can press 'play' and wait for the other player. If both players are ready, the game will be started. On the phone, there only are three buttons needed for playing the game. Figure 6 is shown as an example. In the game the user only has three different weapons to fight: a cannon, a super missiles and a zombie tank. To switch the weapon, the user can press either 'left' or 'right', and 'fire' means the same as its name.

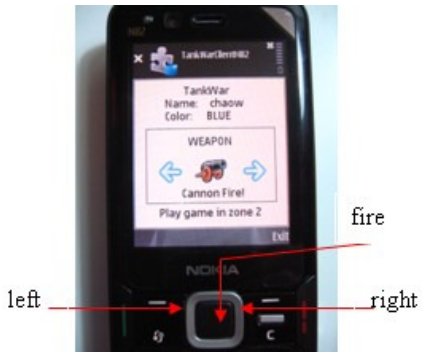

Figure 6. Phone keys division

At the beginning of the game, the game screen displays nothing except waiting for messages by default, as shown in Figure 7. While two users are synchronized successfully, the game will be started immediately. In the game, the user will only control the horizontal movement of the cannon via the active floor, and the way to allow the cannon to move is by walking on the appointed zone, which was already described in Synchronization with the active floor section and showed in Figure 5. When the user's weapon is 'cannon', pressing 'fire' will lead to the cannon fire.

Figure 7. Screenshot at the game beginning

The user has ten zombie tanks in the game beginning by default, but the user can not control the tanks' movement and fire action. The zombie tank will attack the other user's wall as the enemy. If the number of the user's zombie tank is less than five,the user can switch the weapon to 'tank' and press to call for another three zombie tanks. The weapon 'super missile' is more powerful than 'tank', and it can hurt the enemy's wall much more. By default, every user has three super missiles, and the user will obtain one more super missile automatically in the game. To launch 'super missile', the user will switch the weapon on that and press 'fire'.

Game Screen Structure

The screen structure is divided into three parts, which are shown in Figure 8. It includes war zone 1, war zone 2 and player information and game statistics. The player 1 cannon and super missile will be placed in the latter half of war zone 1, and its zombie tanks will be placed in the first half of war zone 2. At the front of cannon, there is the player's wall, and in the last of the war zones is the wall strength indicator. No zombie tank can cross the player's wall.

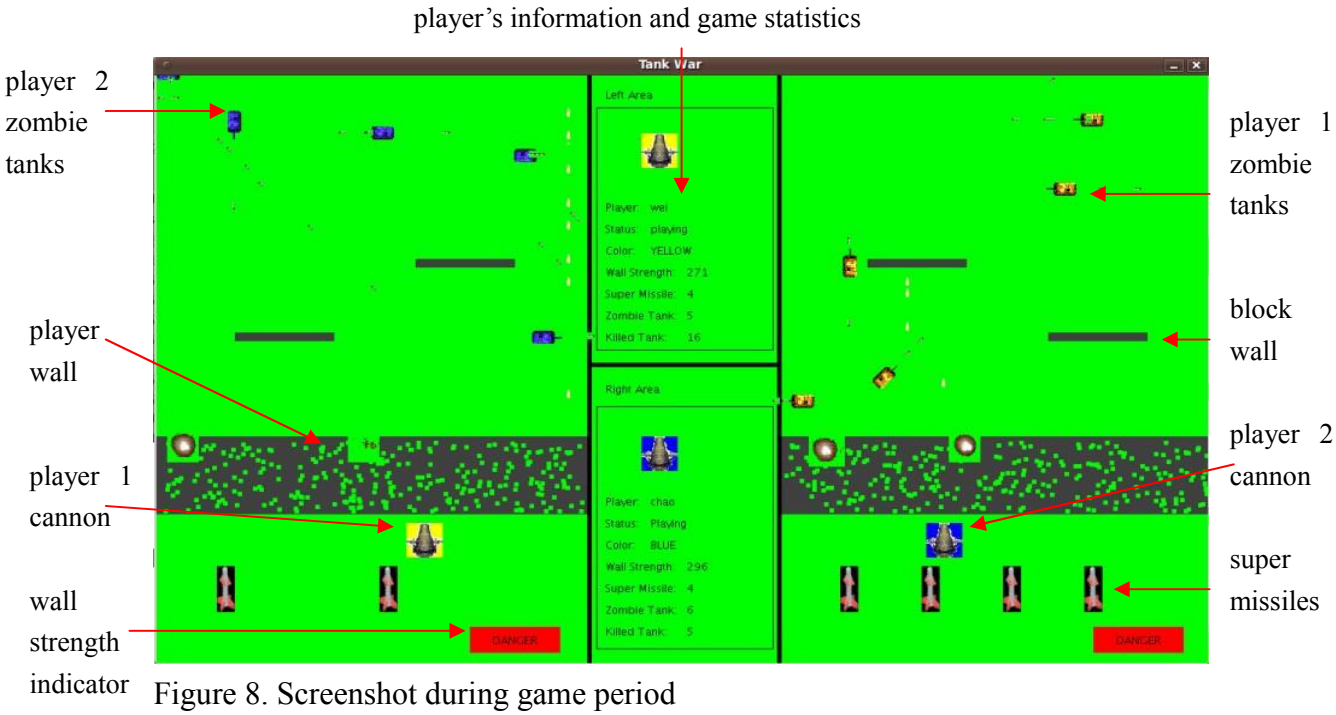

Figure 8. Screenshot during game period

Condition to Win

The only way to win the game is keeping the wall strength greater than zero for as long a time as the player can. The attack target of the zombie tank and the super missile is the enemy's player wall. To reduce the zombie tank's hurt, the user needs to drive the cannon to kill the enemy's zombie tanks, but for the super missile, there is no way to prevent its hurt. In the game period, the wall strength indicator will be always shown, and it has four different levels, including "SAFE", "ATTENTION", "SERIOUS" and "DANGER" with the colors white, orange, magenta and red. "DANGER" is the highest level.

After Play

When the game has a winner, the game screen will display the name, and then the screen will display the messages that inform the client side to quit the game. At last, the game screen will turn back to the beginning phase and wait for the next game period.