J. Software Engineering & Applications, **2008, 1: 20-25**
Published Online December 2008 in SciRes (www.SciRP.org/journal/jsea)

A New Communication Framework for Networked Mobile Games

Chong-wei Xu

Computer Science and Information Systems, Kennesaw State University, USA
Email: cxu@kennesaw.edu

Received November 27th, 2008; revised November 30th, 2008; accepted December 2nd, 2008.

ABSTRACT

This paper introduces a two-layer UDP datagram-based communication framework for developing networked mobile games. The framework consists of a physical layer and a data-link layer with a unified interface as a network communication mechanism. A standalone two-player mobile game, such as a chess game and the like, can be easily plugged on to the communication framework to become a corresponding networked mobile game.

Keywords: *Software Framework, Games, Networked Mobile Games, Network Programming, Games in Education*

1. Introduction

The game industry is growing very rapidly with a speed of "a near doubling in size in a two-year period" [1]. The mobile devices, especially cell phones, are getting popular and have been a solid part of our daily life. In turn, mobile games are growing even faster than desktop games. According to Informa Telecoms and Media, the worldwide market for mobile games will grow from \$2.41 billion in 2006 to \$7.22 billion by 2011. Juniper Research projects that global revenues of mobile games will grow from \$3 billion in 2006 to \$17.5 billion by 2010 [2].

In addition to the growing and the demand of industry market, technically games including mobile games are the integration of Humanities, Mathematics, Physics, Graphics, Multimedia (images and audios) technologies, Artificial Intelligence, Visualization and Animation, Network Structures and Distributed Computing, programming knowledge and skills, and so on. They provide rich teaching materials and engage students for learning. The demands of the game job market and the special features of gaming itself promote a new pedagogical method by using games for educations [3–7].

We have studied the approach for teaching Object-Oriented Programming (OOP) and Component-Oriented Programming (COP) via gaming [8,9]. Furthermore, we have extended the teaching contents to the field of networked gaming. From the technical point of view, the major difference between the standalone games and networked games is the network communication. Considering the special environment of the networked mobile games, they usually are preferred to be based on the peer-to-peer communication. Thus, the UDP protocol is widely used.

Since a networked game consists of the client site and the server site, which are connected by a communication mechanism, usually the development of a networked game starts the discussion of network programming and applies the client-server model to divide the networked game into two parts. Consequently, the traditional way for developing a networked mobile game is emphasizing on the separation of client and server at the early stage as shown in Figure 1 (a). It is the result that both the client and the server usually are a mixture of the gaming code with the communication code [10–14].

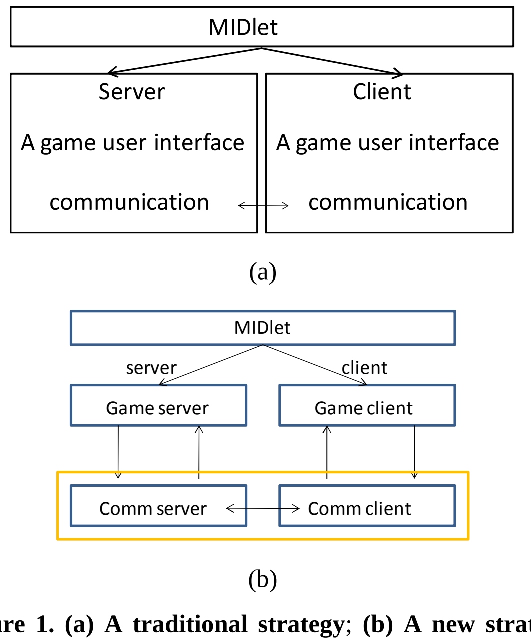


Figure 1. (a) A traditional strategy; (b) A new strategy for developing a networked mobile game

2. A New Problem Solving Strategy

In fact, usually we have had a standalone game already, and then we would like to develop the standalone game to be a networked game. That is, the gaming code and the communication code are the results of two stages of development. Furthermore, if the communication mechanism can be modulated as an independent attachable unit that performs the functionality of passing messages between the client and the server, then it not only increases the reusability and maintainability of the communication code but also makes the transition from the standalone game to the networked game easier.

Following this strategy, we have modulated the communication mechanism as an independent attachable unit with a simple unified interface. Then, two game graphical user interfaces of a standalone game, which represent the client and the server, can be plugged on to the independent communication mechanism through the unified interface for structuring a networked mobile game as depicted in Figure 1 (b). It clearly separates the gaming code from the communication code and allows the communication mechanism can be completely reused for any networked mobile game.

3. Manipulating the UDP Programming Template

For implementing the new strategy, we apply the UDP datagram protocol for making a peer-to-peer environment. By manipulating the UDP datagram communication mechanism in the following steps, the independent attachable communication mechanism has been structured.

First of all, a UDP programming template is derived for depicting its communication mechanism. As we know that J2ME network programming is based on the Generic Communication Framework (GCF) that is illustrated as the connection hierarchy shown in Figure 2. The connection hierarchy has three major interfaces: Content Connection for accessing web data; Datagram Connection for packet-oriented communication; and Stream Connection for stream-based communication. No matter which interface, a foundation class named Connector is used to establish a MIDlet network connection. For mobile games, the more realistic network option is the UDP protocol based on the Datagram Connection because of the limited bandwidth of the mobile phone networks. The programming template of the UDP protocol can be depicted as in Figure 3.

Where, sdc stands for server datagramConnection; cdc stands for client datagramConnection. The server builds up a sdc and prepares an empty datagram packet dg for receiving an input message. And then calls sdc.receive(dg). Whenever the receive() method is invoked, the server process is blocked waiting for the incoming message from the client site. When the client builds up its cdc, it creates a datagram to contain its out-message and issues send() call to send the message out. The server, then, gets the in-message and stores it in the empty datagram packet. This programming template establishes the connection from the client to the server.

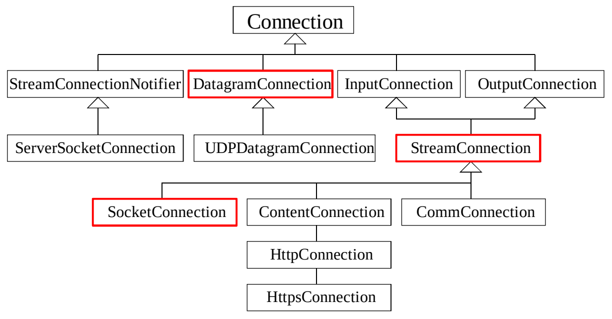
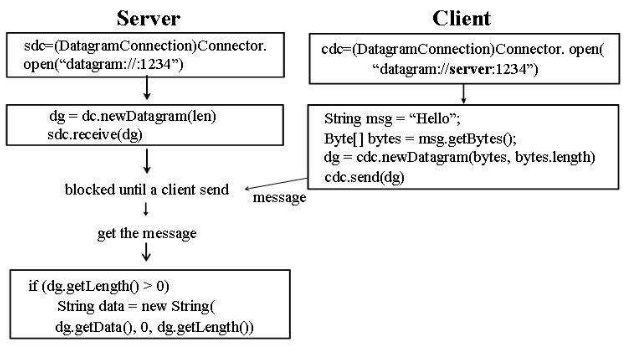


Figure 2. The connection hierarchy of MIDP



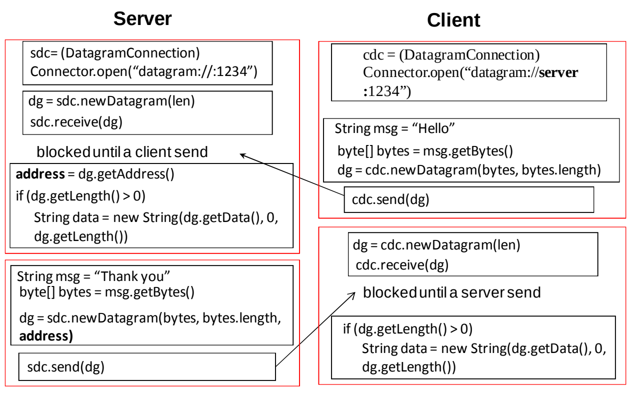
```
sdc=(DatagramConnection)Connector.
open("datagram://:1234")

dg = dc.newDatagram(len)
sdc.receive(dg)

if (dg.getLength() > 0)
    String data = new String(
    dg.getData(), 0, dg.getLength())
```

```
cdc=(DatagramConnection)Connector. open(
    "datagram://server:1234")

String msg = "Hello";
Byte[] bytes = msg.getBytes();
dg = cdc.newDatagram(bytes, bytes.length)
cdc.send(dg)
```

Figure 3. The programming template of UDP protocol

After the server receives the message sent by the client, the server should be able to echo the message back to the client. That is, the client needs to prepare for receiving a message and the server needs to send the message that it just received to the client. The complete programming template is shown as Figure 4. This bi-directional communication mechanism establishes the communication channel and reveals a very symmetric communication system. The only asymmetric codes are referring to the addresses passing, which are marked with the bold face in the figure.

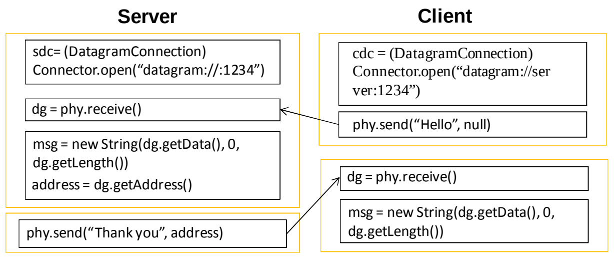
Considering the symmetric scenario, the receiving and sending functions can be moved to a physical layer so that the details of the receiving and sending operations can be hidden. The added physical layer changes Figure 4 to Figure 5.



```
sdc= (DatagramConnection)
Connector.open("datagram://:1234")

dg = sdc.newDatagram(len)
sdc.receive(dg)

address = dg.getAddress()
if (dg.getLength() > 0)
    String data = new String(dg.getData(), 0,
    dg.getLength())

String msg = "Thank you"
byte[] bytes = msg.getBytes()
dg = sdc.newDatagram(bytes, bytes.length,
address)

sdc.send(dg)
```

```
cdc = (DatagramConnection)
Connector.open("datagram://server
:1234")

String msg = "Hello"
byte[] bytes = msg.getBytes()
dg = cdc.newDatagram(bytes, bytes.length)

cdc.send(dg)

dg = cdc.newDatagram(len)
cdc.receive(dg)

if (dg.getLength() > 0)
    String data = new String(dg.getData(), 0,
    dg.getLength())
```

Figure 4. A programming template of the bi-directional communication



```
sdc= (DatagramConnection)
Connector.open("datagram://:1234")

dg = phy.receive()

msg = new String(dg.getData(), 0,
dg.getLength())
address = dg.getAddress()

phy.send("Thank you", address)
```

```
cdc = (DatagramConnection)
Connector.open("datagram://ser
ver:1234")

phy.send("Hello", null)

dg = phy.receive()

msg = new String(dg.getData(), 0,
dg.getLength())
```

Figure 5. A physical layer for sending and receiving (phy.send() and phy.receive())

Obviously, in order to test the communication mechanism shown in Figure 5, an application should be developed. The simple chat application is selected as an example. Its user interface only needs a TextField component for the user to type in out-messages and a StringItem component for displaying the in-messages. Definitely, the chat communication should be a continuous process until one of partners stops the chatting. For that purpose, a loop is added to keep the chatting process continuous and a sending command is used by the users whenever they make their messages available for sending.

Unfortunately, this version of the chat application experienced both deadlock and duplicate message sending problems. The problems are caused by the structure of the communication mechanism, which uses the physical layer to contain both the phy.receive() and the phy.send() calls. The codes of the phy.receive() and the phy.send() are as follows.

```
public synchronized void send(String msg, String address) {
   byte [] bytes = msg.getBytes();
   try {
      if (address == null) {
         dg = cdc.newDatagram(bytes, bytes.length);
         cdc.send(dg);
      } else {
         dg = sdc.newDatagram(bytes, bytes.length, address);
         sdc.send(dg);
      }
   } catch (IOException ex) {
      ex.printStackTrace();
   }
}

public synchronized Datagram receive(String name)
{
   try {
      if (name.equals("Client")) {
         dg = cdc.newDatagram(100);
         cdc.receive(dg);
      } else if (name.equals("Server")) {
         dg = sdc.newDatagram(100);
         sdc.receive(dg);
      }
   } catch (IOException ex) {
      ex.printStackTrace();
   }
   return dg;
}
```

Due to the fact that two methods are shared by both the client and the server, they form critical sections. In order to protect these two critical sections, both methods should be a synchronized method. That is, only one process can enter the methods at a time. Unfortunately, both methods contain sdc (server's datagram connection object) and cdc (client's datagram connection object). As we know that when one process, say the server process, invokes the receive() call, it should be blocked until the other process, the client process, issues a send() call. Therefore, when the server invokes the method phy.receive(), not only the server process itself will be blocked but also the other process, the client process, will be blocked too due to the synchronized protection blocks both resources sdc and cdc inside the phy.receive(). That makes the client process unable to invoke the send() method for sending a message to release the server process since the cdc is blocked. All these together cause a deadlock as depicted in Figure 6.

For overcoming this problem, the synchronized requirement for the phy.receive() has to be released. But, this allows both processes to enter the phy.receive() at the same time and it causes a duplicate message sending.

These two phenomena forced us to move the receive() method out of the physical layer and place it back to the original position and only keep the send() method in the physical layer as Figure 7 shows. This continuous communication mechanism keeps the chat application working. Clearly, it makes both the client and the server consists of three layers: the user interface layer on the top, the physical layer on the bottom, and a layer in the middle, which we gave a name to it as "data link layer".

Based on this layered structure, the user interface layer could be replaced by any game graphical user interface. However, the send Command designed for the chat application cannot be used for games since the players of a game should be able to use key presses for playing the game. Thus, between the user interface layer and the data link layer, a unified interface that consists of two methods: userinterface.receiveMessage(String inMsg)

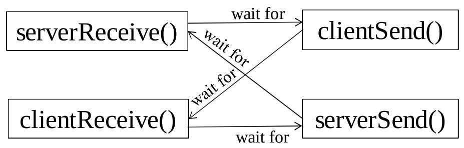
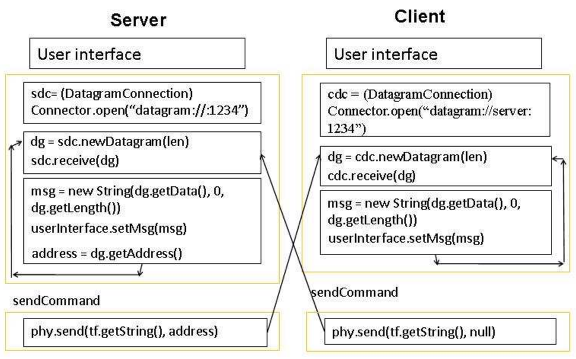


Figure 6. The deadlock scenario



```
sdc= (DatagramConnection)
Connector.open("datagram://:1234")
dg = sdc.newDatagram(len)
sdc.receive(dg)
msg = new String(dg.getData(), 0,
dg.getLength())
userInterface.setMsg(msg)
address = dg.getAddress()
phy.send(tf.getString(), address)
```

```
cdc = (DatagramConnection)
Connector.open("datagram://server:
1234")
dg = cdc.newDatagram(len)
cdc.receive(dg)
msg = new String(dg.getData(), 0,
dg.getLength())
userInterface.setMsg(msg)
phy.send(tf.getString(), null)
```

Figure 7. A continuous communication mechanism

and datalink.sendMessage (String outMsg) is inserted. This unified interface plays a role of bridge between the user interface layer and the data link layer. When a player of a game triggers an action that causes the change of the states of the game at one site, the new states will be sent to the other site. The new states carried by the inMsg will be further interpreted by an overloaded method setParameters(inMsg) in the game user interface for controlling the scene of the game. Through this unified interface, the graphical user interface of any standalone game can be easily plugged onto the communication framework as summarized in Figure 8.

4. A Networked Mobile Game Connect 4

We take the Connect4 networked mobile game as an example to demonstrate the application of the framework. This networked game has been described in [10] and implemented according to the traditional method. We have re-designed and re-implemented it by using the new framework. The same game implemented in different strategies enables us to compare the two different strategies for designing and implementing networked mobile games.

For using the framework, a standalone Connect4 game should be developed first, and then add its game graphical user interface on the top of the data link layer in the framework through the unified interface. Because both the client and the server will display the same game user interface, we only need one game user interface for both the client and the server with their own different names, respectively. The standalone game Connect4 that we have developed is described by the simplified UML diagram shown in Figure 9.

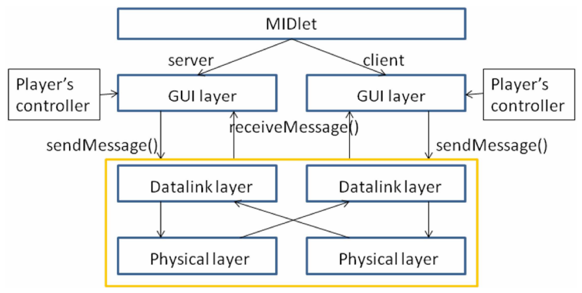


Figure 8. The framework for developing UDP datagram based networked mobile games

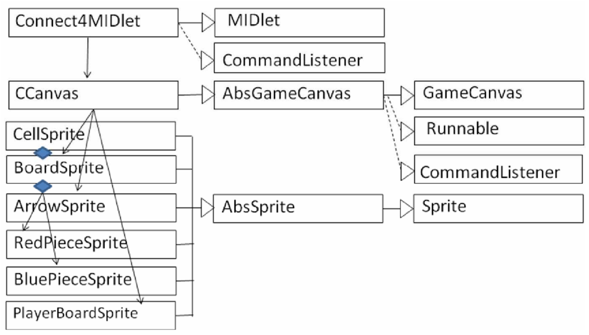
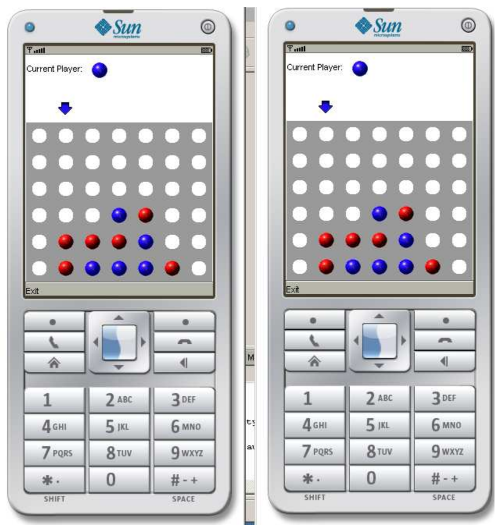
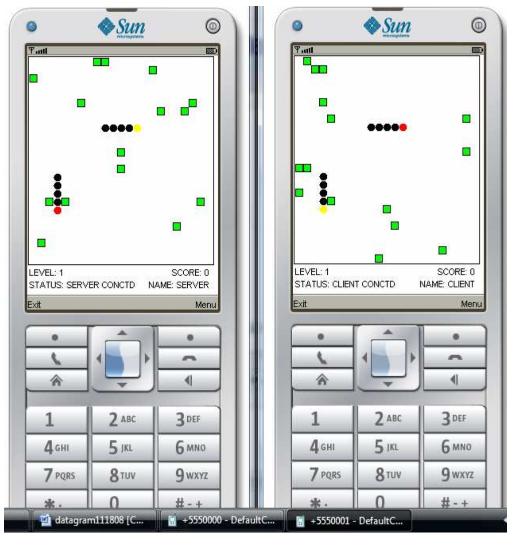


Figure 9. The simplified UML diagram of the standalone game Connect 4

By plugging two game user interfaces with the communication mechanism, the networked mobile game Connect4 is built up as shown in Figure 10 (a).



(a)


LEVEL: 1 SCORE: 0
STATUS: SERVER CONCTD NAME: SERVER
Exit Menu
LEVEL: 1 SCORE: 0
STATUS: CLIENT CONCTD NAME: CLIENT
Exit Menu

(b)

Figure 10. (a) The turn-based networked mobile game Connect4; (b) The event-based networked mobile game Worm. (The left is the server; the right is the client)

The players can control the networked mobile game Connect4 by using the right and the left keys to move the arrow for indicating the target column, and then the players can press the fire key to drop the piece on to the target column. They will take a turn to drop their own pieces with different colors (red and blue). Who will link the four pieces with the same color together either along the horizontal, vertical, or diagonals, who will be the winner of the game.

One of the important design considerations of a networked game is what information should be passed between the client and the server. For the networked mobile game Connect4, there are two kinds of message should be sent. One kind of message only contains a column value, which corresponds to the right or the left key pressing, for synchronizing the arrows' movements in two sites. The second kind of message contains two values: the column number and the current color value, which corresponds to the fire key pressing, for synchronizing the piece dropping. No matter which kind of messages, the user interface layer of the sender site can call the unified interface method datalink.sendMessage() to send out a string to the other site. When the receiver site receives the message, its data link layer can use the unified interface method userinterface.receiveMessage() to move the received message up to the user interface layer. The user interface layer calls the overloaded method setParameters() to interpret the received message for controlling the actions on the receiver site. Due to the fact that both sites have the exact same game logic and under the control of the same parameters, the game user interface layer will display everything the same in both sites, which is the same as the standalone game graphical user interface.

In detail, the networked version needs additional two pieces of code in comparison with the standalone version. One is that the user interface layer needs to instantiate an object of data link layer for sending and receiving messages. The other piece is that when the user interface layer receives messages from the data link layer, it needs to interpret the receiving messages for controlling its own game user interface. In the networked mobile game Connect4, there are two kinds of message are passed so that the user interface layer needs two overloading methods setParameters() to interpret the different messages.

5. Conclusions and Future Work

This framework releases the burden for considering a totally different design and implementation between a standalone game with its corresponding networked version. Any take-turn based game can be easily plugged on to the network communication mechanism since it is designed and implemented by following the component-oriented programming philosophy. This structure of the framework allows the data link layer and the physical layer completely reusable. It also makes the standalone game reusable up to 90% when it will be developed to be a networked game. The game logic wouldn't be touched for both the standalone and the networked versions and all required parameters will be passed along the channel for communication.

Besides supporting networked mobile game development, this framework is also a practical tool for teaching network programming since the developing process of the framework is a manipulation of the UDP protocol. From the manipulation process, students can better understand the functionality of the protocol. It also promotes a sequence of analysis and synthesis processes and enhances students' problem solving ability. Going through the process for developing the framework, we guide students to explore the essential principles of network communication and enrich their foundation on object, module, and component oriented philosophy.

The networked mobile game Connect4 is a turn-based game. Many standalone mobile games played by two competitors, such as a tic-tac-toe, a chess game, an Othello game, and the like belong to this category. These games send and receive messages in a sequential order. The other category of networked games are event-based, where input events made by the players can occur at any time and any player can interact with the game at any time in any order. That is, the messages sent and received are in a concurrent matter. We have developed a networked version of the classic Worm game using the framework, which has two Worms. One player controls one worm for competing to eat the treats as shown in Figure 10 (b). Its functional behaviors need more deeply observations.

This framework is based on the UDP datagram protocol since it supports peer-to-peer model of communications. That limits the number of players to two. What if more players would like to join? Furthermore, the clients of networked mobile games are better to be a thin client since mobile devices have limited supports on their resources. For realizing a thin client, we'd better to move more codes, especially the game logic that is shared by both sites, to be resided on the server site so that two clients don't need to carry them. How to satisfy these requirements? These are the topics that we need to further explore.

6. Acknowledgement

This project was partially supported by the Scholarship of Teaching and Learning Team (STLT) fund, The Center for Excellence in Teaching & Learning (CETL), Kennesaw State University, 2007–2008.

REFERENCES

[1] M. Zyda, "Educating the next generation of game developers," Computer, IEEE, June 2006.
[2] F. Chau, "Mobile gaming aims for mass market," 2006. http://www.smackall.com/viewresource.php?resource=17.

[3] M. Mayo, “Games for science and engineering education,” CACM, Vol. 50, No. 7, July 2007.

[4] M. Zyda, “Creating a science of games,” CACM, Vol. 50, No. 7, July 2007.

[5] J. Schollmeyer, “Games get serious,” Bulletin of the Atomic Scientists, 2007. http://www.thebulletin.org/article.php?art_ofn=ja06schollmeyer_100.

[6] A. Phelps, K. Bierre, and D. Parks, “MUPPETS: Multi-user programming pedagogy for enhancing traditional study,” CITC4’03, Lafayette, Indiana, USA, October 16–18, 2003.

[7] K. Bierre and A. Phelps, “The use of MUPPETS in an introductory java programming course,” SIGITE’04, Salt Lake City, Utah, USA, October 28–30, 2004.

[8] C. W. Xu (2007), “A hybrid gaming framework and its applications,” The International Technology, Education and Development Conference 2007 (INTED2007), Valencia, Spain, pp. 30000_0001.pdf, March 7–9, 2007.

[9] C. W. Xu (2008), “Teaching OOP and COP technologies via gaming,” in book “Handbook of research on effective electronic gaming in education,” Edited by Richard E. Ferdig, University of Florida, pp. 508–524, IGI Global, 2008.

[10] M. Morrison, “Beginning mobile phone game programming,” Sams, 2005.

[11] C. Hamer, “J2ME games with MIDP2,” Apress, 2004.

[12] J. Fan, E. Ries and C. Tenitchi, “Black art of java game programming,” Waite Group Press, 1996.

[13] A. Davison, “Killer game programming in java,” O’Reilly, 2005.

[14] D. Brackeen, B. Barker, and L. Vanhelsuwe, “Developing games in Java,” New Riders, 2004.