THE CROSS PLATFORM MOBILE GAME

ITTI APINYANONT

MEESITH PITUCKVASUKARN

A SPECIAL PROJECT SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIRMENT FOR DEGREE OF BACHELOR OF SCIENCE

IN COMPUTER SCIENCE (INTERNATIONAL PROGRAM)

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

ACADEMIC YEAR 2011

Thesis Title	Cross Platform Mobile Game	
Student	Mr. Itti Apinyanont	Student ID: 51051218
	Mr. Meesith Pituckvasukarn	Student ID: 51051210
Degree	*Bachelor of Science*	
Program	Computer Science (International Program)	
Year	2011	
Thesis Advisor	Dr. Rungrat Wiangsripanawan	

ABSTRACT

This special project aim is to study and develop the cross-platform mobile game called "Nyorion". Nyorion is a two-player turn based board game which the player who arrives at the destination first will win. The player's number of moves is determined randomly by the system. During their journeys, players either face the randomly chosen event or have to compete with each other in playing the mini-game. There are three events and two mini-games. Players' devices are communicated through WiFi connection. Xcode version 4.3.2 and Android SDK version 3.7.2 are used to develop the game on iOS and android devices respectively. Since our main objective is indeed to study how to develop a game which can be played between iOS and android devices from scratch, problems and suggestions of how to implement this special type of game based on our experience are also provided.

Keywords : Android, Xcode, Game, Tablet, Board game, Cross platform, iOS

ACKNOWLEDGEMENTS

Our special project would not have been possible without the help and advice from many people.

Special thanks to our parents for their loves and supports. Thanks for their encouragements and useful advice.

Special thanks to Dr. Rungrat Wiangsripanawan for her advice and support throughout our project. We would like to thank for all her comments and her time for helping us correcting the project report.

Special thanks to Assoc.Prof.Dr. Jeeraporn Werapun and Asst. Prof.Dr. Korakot Prachumrak for their suggestions to improve our project.

Special thanks to Pisut Kasinpila and his mother for a very great suggestion for Xcode programming technique and Eakawat Tantamjarik for his help and suggestions on Android programming.

Special thanks to stackoverflow.com community for many great advice and suggestions for iOS programming and socket programming.

Special thanks to our friends in KMITL Com-Sci inter #9. Thanks for always helping each other when needed.

Last but not least, special thanks to Mrs. Pranee Apinyanont, Mr.Itti Apinyanont's mother who had passed away while we were doing the last stage of our project. She helped us very much in editing the project report and gave us a lot of encouragement even when she was sick and stayed in the hospital. She will always in our memory forever.

Itti Apinyanont

Meesith Pituckvasukarn

Contents

Contents (Cont.)

Contents (Cont.)

Contents (Cont.)

Contents (Cont.)

List of Table

List of Figure

List of Figure (Cont.)

List of Figure (Cont.)

List of Figure (Cont.)

List of Figure (Cont.)

List of Figure (Cont.)

List of Figure (Cont.)

Chapter 1

Introduction

This chapter explains the cause of our project. It also covers the scope of this work, the benefits from creating this game, tools that will be using and the plan to complete the project.

1.1 Importance and cause of problem

Smartphones and tablets have become so popular nowadays. Not only that their sizes are small which allow people to carry them everywhere, but also the performance is much more powerful than before. The devices capacities and performance are almost as powerful as a personal computer. Hence, several new applications are emerged due to the development in the device features and technology. Game itself is not a new feature on the mobile device. However, with the growth of the mobile technology in memory, screen, processor etc. and the popularity of games on smartphones, there is a need for a game that is more challenging and can be played by several players at the same time on their own devices. Some might argue that this can be done through the gaming application on the web. However, playing game through web browser on smart phones or tablets have several limitations. One of them is it needs the Internet connection while playing. Also, some games may need to download the game before playing them. Nevertheless, developing a game application on the mobile device relies on the device platform. That is, the android games can be played only on mobiles or tablets that run Android OS. Similarly, the iPhone or iPad games can only be played on devices that run iOS. Therefore, for our project, we are interested in developing a two-player mobile game which each player can play on his own phone or tablet. In addition, it is also a cross-platform game which allows players with the iOS devices and players with Android devices to play this game together. Our game will be based on a turn based party board game with some real time functions and will use Wi-Fi for its players' connections.

1.2 Purposes of the special topic

1) To develop a party board game for smart phones and tablets.
2) To develop a cross-platform mobile application between the iOS and Android devices
3) To study how to program an application for android OS device
4) To study how to program an application for iOS device.
5) To study how to design an application protocol and write a socket programming.

1.3 Scope

Our project focuses on the multiplayer 2-dimentional game that can be played between any two android or iOS tablets or smart phones via Wi-Fi connection. There are three types of connections. First, two players are on android devices. Second, two players are on iOS devices. The last one is the highlight of our work, the game can be played by two players from different platforms, one of them uses iOS while another one uses android.

1.4 Equipment

Hardware

a) iPad with iOS 5.0.1
b) Galaxy Tab 10.1 with android OS 3.1
c) Computer notebook and desktop PC
d) MacBook Pro

Software

a) Eclipse
b) Android SDK tool
c) Xcode 4.3 (for iOS programming)
d) Windows 7 Paint
e) Adobe Photoshop CS4

1.5 Project Planning

1) Plan scope and purpose of the project
2) List tasks of the project
3) Study android OS and iOS
4) Study socket programming
5) Design and plan interfaces and functions of the game
6) Design the game system
7) Implement the game according to the plan
8) Debug and test the game
9) Prepare and write the game documentation and manual.

1.6 Expected benefits

1) Able to program an iOS or MAC OS X application using Xcode.
2) Able to program an Android application using Android SDK.
3) Understand network game application's design and implementation concept.
4) Understand essential requirements to implement a cross-platform iOS and Android application

Chapter 2

Background

This chapter begins with the brief description of game and its genres. Then, Android OS and iOS, which are two of the most popular mobile devices platforms, backgrounds will be given. Next, TCP/IP concept will be described. Finally, Android SDK, Xcode and Eclipse which are all tools that we use to develop this project will be introduced and described.

2.1 Video game[1][2]

Video game is an electronic game that uses user interaction from user interface to generate visual feedback on a video device. The input device that used to interacting with video games is called a game controller which varies across platforms such as joystick, keyboard, etc. Video games typically use additional means of providing interactivity and information to the player. Audio is almost universal, using sound reproduction devices, such as speakers and headphones. Other feedback may come, such as vibration or force feedback, with vibration sometimes used to simulate force feedback.

Video game genres are used to categorize video games based on their game play interaction. A video game genre is defined by a set of game play challenges. They are classified independently from their setting or game-world content such as:

2.1.1 Action games

Action game requires players to use quick reflexes, accuracy, and timing to overcome obstacles. It is perhaps the most basic of gaming genres, and certainly one of the broadest. Action games tend to have game play with emphasis on combat. There are many subgenres of action games such as fighting games.

2.1.2 First-person shooting (FPS)

Emphasize shooting and combat from the perspective of the character controlled by the player. This perspective is meant to give the player the feeling of "being there", and allows the player to focus on aiming. Most FPSs are very fast-paced and require quick reflexes on high difficulty levels. The fast-paced and 3D elements required to create an effective looking FPS made the genre technologically unattainable for most consumer hardware systems

2.1.3 Third-person shooting (TPS)

Emphasize shooting and combat from a camera perspective in which the player character is seen at a distance. This perspective gives the player a wider view of their surroundings as opposed to the limited viewpoint of first-person shooters. Furthermore, third-person shooters allow for more elaborate movement such as rolling or diving, as opposed to simple jumping and crouching common in FPS games. Greater interaction with the player's environment is often possible.

2.1.4 Adventure games

This kind of game assumes player the role of protagonist in an interactive story driven by exploration and puzzle-solving. Nearly all adventure games are designed for a single player, since this emphasis on story and character makes multi-player design difficult.

2.1.5 Role-playing games

The player in RPG game takes role in one of the character, or control several adventuring party members, fulfilling one or many quests.

2.1.5 Simulation games

Generally simulation game designed to closely simulate aspects of a real or fictional reality by using the simulation technique or the complex mathematic algorithm to generate the core of the game.

2.1.6Real-time Strategy

Game time progresses continuously according to the game clock. Players perform actions simultaneously as opposed to in sequential units or turns. Players must perform actions with the consideration that their opponents are actively working against them in real time, and may act at any moment.

2.1.7 Turn-based Strategy

Game flow is partitioned into well-defined and visible parts, called turns. A player of a turn-based game is allowed a period of analysis before committing to a game action, ensuring a separation between the game flow and the thinking process, which in turn presumably leads to better choices. Once every player has taken his or her turn, which round of play is over, and any special shared processing is done. This is followed by the next round of play. In games where the game flow unit is time, turns may represent such things as years, months, weeks or days.

2.1.8 Music games

Most commonly challenge the player to follow sequences of movement or develop specific rhythms. Some games require the player to input rhythms by stepping with their feet on a dance pad, or using a device similar to a specific musical instrument, like a replica drum set. These games have changed the way players' interact with their consoles by making the gaming experience more active and sociable.

2.1.9 Party games

Video games developed specifically for multiplayer games between many players. Normally, party games have a variety of mini-games that range between collecting more of a certain item than other players or having the fastest time at something.

2.1.10 Board games

Board game is a game in which counters or pieces are placed, removed, or moved on a marked surface or "board" according to a set of rules. Games may be based on pure strategy, chance or a mixture of the two and usually have a goal which a player aims to achieve. Early board games represented a battle between two armies and most current board games are still based on beating opposing players in terms of counters, winning position or accrual of point.

2.1.11 Puzzle games

Require the player to solve logic puzzles or navigate complex locations such as mazes. This genre frequently crosses over with adventure and educational games. Some arcade games, in particular game play depends on hand/eye coordination and quick reflexes, rather than thought and logic.

2.1.12 Sports games

Emulate the playing of traditional physical sports. Some emphasize actually playing the sport, while others emphasize the strategy behind the sport.

2.2 Smartphone[3]

Smartphone is a high-end mobile phone. A Smartphone combines the functions of a personal digital assistant (PDA) and a mobile phone. The term Smartphone is usually used to describe phones with more advanced computing ability and connectivity than a contemporary feature phone .Smartphone also serve as portable media players and camera phones with high-resolution touch screens, web browsers that can access and properly display standard web pages rather than only mobile-optimized sites, GPS navigation, Wi-Fi and mobile broadband access.

2.3 Tablet personal computer [4]

Tablet personal computer or tablet PC is a tablet-sized computer that also has the key features of a full-size personal computer. A tablet PC is essentially a small laptop computer, equipped with a rotatable touch screen as an additional input device, and running a standard PC operating system like Windows or Linux.

2.4 Android[5]

Android is operating system for mobile devices such as mobile phones and tablet computers. Android developed under Google. Programming is mainly in XML, C Java and C++.

2.4.1 Android Architecture

The following diagram shows the major components of the Android operating system.

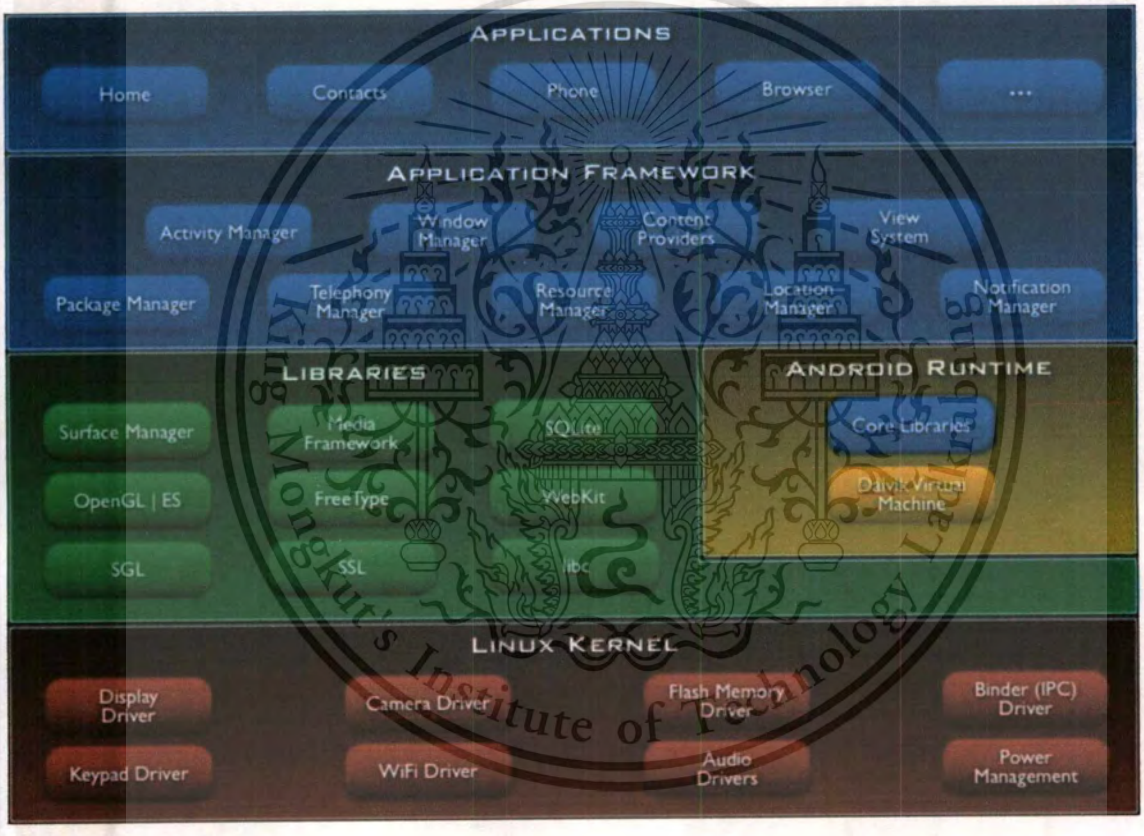


Figure 2.1 Android Architecture

2.4.2 Applications

Android will ship with a set of core applications including an email client, SMS program, calendar, maps, browser, contacts, and others. All applications are written using the Java programming language.

2.4.3 Application Framework

By providing an open development platform, Android offers developers the ability to build extremely rich and innovative applications. Developers are free to take advantage of the device hardware, access location information, run background services, set alarms, add notifications to the status bar, and much, much more.

Developers have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This same mechanism allows components to be replaced by the user.

Underlying all applications is a set of services and systems, including:

- A rich and extensible set of Views that can be used to build an application, including lists, grids, text boxes, buttons, and even an embeddable web browser.

- Content Providers that enable applications to access data from other applications (such as Contacts), or to share their own data.

- A Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files.

- A Notification Manager that enables all applications to display custom alerts in the status bar.

- An Activity Manager that manages the lifecycle of applications and provides a common navigation back stack.

2.4.4 Libraries

Android includes a set of C/C++ libraries used by various components of the Android system. These capabilities are exposed to developers through the Android application framework. Some of the core libraries are listed below:

- System C library - a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices.
- Media Libraries - based on PacketVideo'sOpenCORE; the libraries support playback and recording of many popular audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG.
- Surface Manager - manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications.
- LibWebCore - a modern web browser engine which powers both the Android browser and an embeddable web view.
- SGL - the underlying 2D graphics engine.
- 3D libraries - an implementation based on OpenGL ES 1.0 APIs; the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer.
- FreeType - bitmap and vector font rendering.
- SQLite - a powerful and lightweight relational database engine available to all applications.

2.4.5 Android Runtime

Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language.

Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.

The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

2.4.6 Linux Kernel

Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

2.4.7 Android Feature

- The platform supports, VGA, 2D graphics library, 3D graphics library based on OpenGL ES 2.0.

- Using lightweight relational database or SQLite for data storage.

- Android supports connectivity technologies such as GSM/EDGE, Bluetooth, Wi-Fi, LTE, NFC, WiMAX, etc.

- The web browser on Android is based on the open-source WebKit layout engine with Chrome's V8 JavaScript engine.

-Dalvik act as virtual machine designed specifically for Android to run java, optimized battery usage and limited memory and CPU in mobile devices.

- Android supports many media formats WebM, H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, MP3, MIDI, OggVorbis, FLAC, WAV, JPEG, PNG, GIF, BMP.

- Android also supports additional hardware such as cameras, touch screens, GPS, accelerometers, gyroscopes, magnetometers, proximity and pressure sensors, thermometers, etc.

- Android has native support for multi-touch.

- Multitasking of applications is available.

2.5 iPhone OS[6]

iPhone OS or iOS is Apple's mobile operating system which programming mainly in C, C++ and Objective-C. Now the device that supports IOS is iPhone, iPod Touch, and iPad. The user interface of iOS is based on the concept of direct manipulation, using multi-touch gestures. Interface control elements consist of sliders, switches, and buttons. The response to user input is immediate and provides a fluid interface. Interaction with the OS includes gestures such as *swipe*, *tap*, *pinch*, and *reverse pinch*, all of which have specific definitions within the context of the iOS operating system and its multi touch interface. Internal accelerometers are used by some applications to respond to shaking the device (one common result is the undo command) or rotating it in three dimensions (one common result is switching from portrait to landscape mode).

2.5.1 iOS Features

-The home screen displays application icons and a lay at the bottom of the screen where users can pin their most frequently used application.

- iOS contains default applications which are phone, mail, safari and iPod.

- iOS supports 2G, 3G, Wi-Fi, Bluetooth and EDGE.

- With iOS 4, 3rd-generation and newer iOS, supports multitasking through seven background APIs which are Background audio, Voice over IP, Background location, Push notifications, Local notifications, Task finishing and Fast app switching.

- Game Center is an online multiplayer in iOS "social gaming network" which allows users to invite friends to play a game, start a multiplayer game through matchmaking, track their achievements, and compare their high scores on a leader board.

-iOS has Retina display which increases pixel density to make the display sharper.

2.6 Computing network programming[7]

Computing network programming, socket programming or client–server programming is the way to write computer programs that communicate with other programs across a computer network. The program that initiating the communication is called a **client** and the program waiting for the communication to be initiated is the **server**. The communication process can be either connection-oriented (TCP) or connectionless (UDP). These are examples of functions or methods typically provided by the API library:

- *socket()* creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- *bind()* is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- *listen()* is used on the server side, and causes a bound TCP socket to enter listening state.
- *connect()* is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
- *accept()* is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- *send()* and *recv()*, or *write()* and *read()*, or *recvfrom()* and *sendto()*, are used for sending and receiving data to/from a remote socket.
- close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

2.6.1 Transmission Control Protocol[8]

The Transmission Control Protocol (TCP) is one of the core protocols of the Internet Protocol Suite. TCP is one of the two original components of the suite, complementing the Internet Protocol (IP), and therefore the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer. TCP is the protocol that major Internet applications such as the World Wide Web, email, remote administration and file transfer rely on.

TCP provides a communication service at an intermediate level between an application program and the Internet Protocol (IP). That is, when an application program desires to send a large chunk of data across the Internet using IP, instead of breaking the data into IP-sized pieces and issuing a series of IP requests, the software can issue a single request to TCP and let TCP handle the IP details.

IP works by exchanging pieces of information called packets. A packet is a sequence of octets and consists of a header followed by a body. The header describes the packet's destination and, optionally, the routers to use for forwarding until it arrives at its destination. The body contains the data IP is transmitting.

Due to network congestion, traffic load balancing, or other unpredictable network behavior, IP packets can be lost, duplicated, or delivered out of order. TCP detects these problems, requests retransmission of lost data, rearranges out-of-order data, and even helps minimize network congestion to reduce the occurrence of the other problems. Once the TCP receiver has reassembled the sequence of octets originally transmitted, it passes them to the application program. Thus, TCP abstracts the application's communication from the underlying networking details.

TCP is a reliable stream delivery service that guarantees delivery of a data stream sent from one host to another without duplication or losing data. Since packet transfer is not reliable, a technique known as positive acknowledgment with retransmission is used to guarantee reliability of packet transfers. This fundamental technique requires the receiver to respond with an acknowledgment message as it receives the data. The sender keeps a record of each packet it sends, and waits for acknowledgment before sending

the next packet. The sender also keeps a timer from when the packet was sent, and retransmits a packet if the timer expires. The timer is needed in case a packet gets lost or corrupted.

TCP consists of a set of rules: for the protocol, that are used with the Internet Protocol, and for the IP, to send data "in a form of message units" between computers over the Internet. At the same time that IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual units of data transmission, called segments that a message is divided into for efficient routing through the network. For example, when an HTML file is sent from a Web server, the TCP software layer of that server divides the sequence of octets of the file into segments and forwards them individually to the IP software layer (Internet Layer). The Internet Layer encapsulates each TCP segment into an IP packet by adding a header that includes (among other data) the destination IP address. Even though every packet has the same destination address, they can be routed on different paths through the network. When the client program on the destination computer receives them, the TCP layer (Transport Layer) reassembles the individual segments and ensures they are correctly ordered and error free as it streams them to an application.

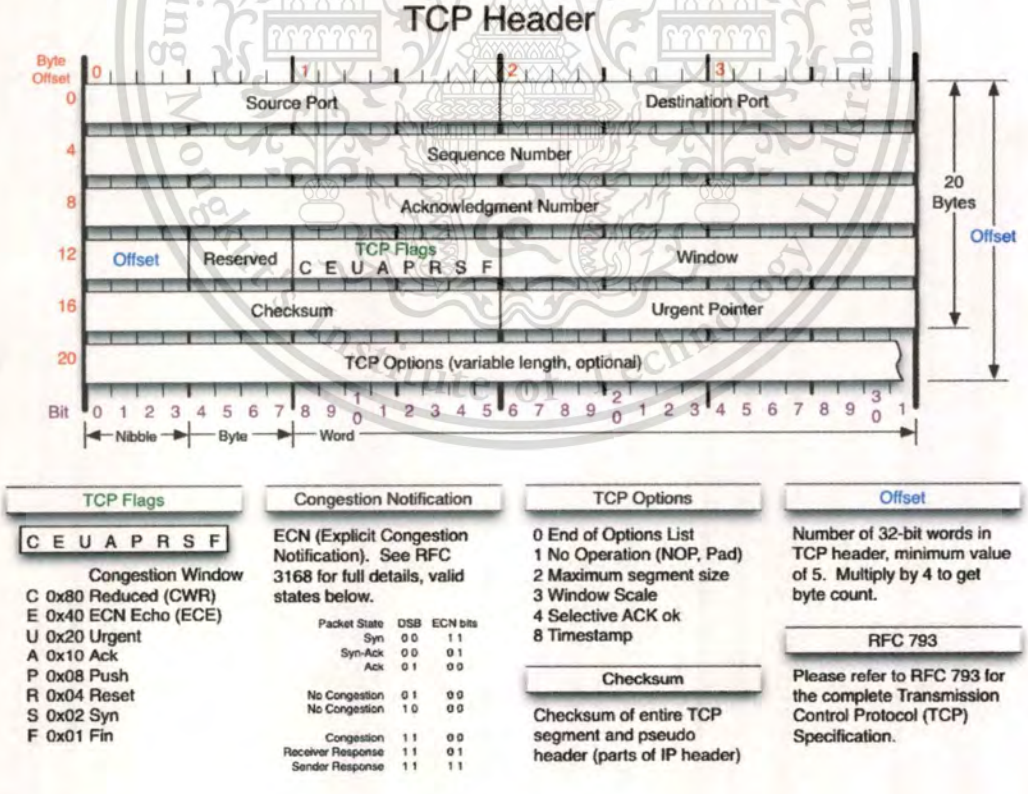


Figure 2.2 TCP header

2.6.2 User Datagram Protocol[9]

The User Datagram Protocol (UDP) is one of the core members of the Internet Protocol Suite, the set of network protocols used for the Internet. With UDP, computer applications can send messages, in this case referred to as data grams, to other hosts on an Internet Protocol (IP) network without requiring prior communications to set up special transmission channels or data paths.

UDP uses a simple transmission model without implicit handshaking dialogues for providing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and data grams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

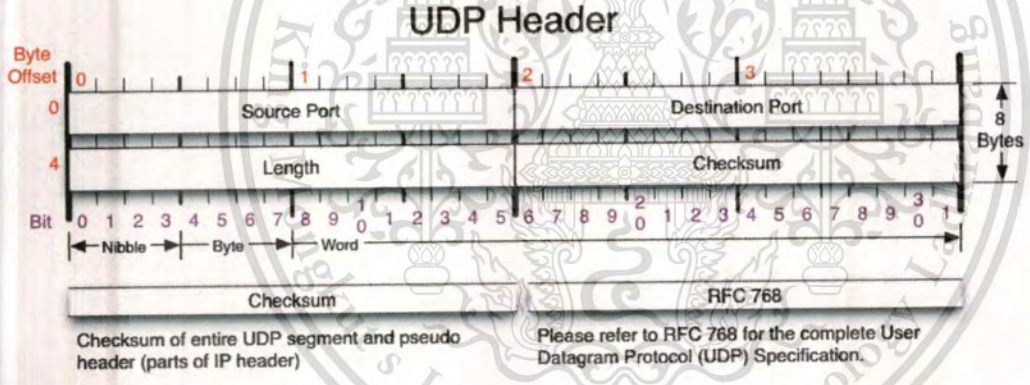


Figure2.3UDP header

2.6.3 Comparison of UDP and TCP[9]

Transmission Control Protocol is a connection-oriented protocol, which means that it requires handshaking to set up end-to-end communications. Once a connection is set up user data may be sent bi-directionally over the connection.

- Reliable – TCP manages message acknowledgment, retransmission and timeout. Multiple attempts to deliver the message are made. If it gets lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.
- Ordered – if two messages are sent over a connection in sequence, the first message will reach the receiving application first. When data segments arrive in the wrong order, TCP buffers the out-of-order data until all data can be properly re-ordered and delivered to the application.
- Heavyweight – TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.
- Streaming – Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries.

UDP is a simpler message-based connectionless protocol. Connectionless protocols do not set up a dedicated end-to-end connection. Communication is achieved by transmitting information in one direction from source to destination without verifying the readiness or state of the receiver.

- Unreliable – When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission or timeout.
- Not ordered – If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.
- Lightweight – There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.
- Datagrams – Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent.
- No congestion control – UDP itself does not avoid congestion, and it's possible for high bandwidth applications to trigger congestion collapse, unless they implement congestion control measures at the application level.

2.7 Cross-platform[10]

Cross-platform, or multi-platform, is an attribute conferred to computer software or computing methods and concepts that are implemented and inter-operate on multiple computer platforms. Cross-platform software may be divided into two types; one requires individual building or compilation for each platform that it supports, and the other one can be directly run on any platform without special preparation, e.g., software written in an interpreted language or pre-compiled portable byte code for which the interpreters or run-time packages are common or standard components of all platforms.

2.7.1 Cross-platform software

2.7.1.1 Web applications

Web applications are typically described as cross-platform because, ideally, they are accessible from any of various web browsers within different operating systems. Such applications generally employ a client–server system architecture, and vary widely in complexity and functionality. This wide variability significantly complicates the goal of cross-platform capability, which is routinely at odds with the goal of advanced functionality.

2.7.1.2 Video games

Cross-platform is a term that can also apply to video games released on a range of video game consoles, specialized computers dedicated to the task of playing games.

The characteristics of a particular system may lengthen the time taken to implement a video game across multiple platforms. So, a video game may initially be released on a few platforms and then later released on remaining platforms. Typically, this situation occurs when a new gaming system is released, because video game developers need to acquaint themselves with the hardware and software associated with the new console.

2.8 Android SDK[11]

The Android software development kit (SDK) includes a comprehensive set of development tools. These include a debugger, libraries, a handset emulator (based on QEMU), documentation, sample code, and tutorials. Currently supported development platforms include computers running Linux (any modern desktop Linux distribution), Mac OS X 10.4.9 or later, Windows XP or later. The officially supported integrated development environment (IDE) is Eclipse (currently 3.5 or 3.6) using the Android Development Tools (ADT) Plug in, though developers may use any text editor to edit Java and XML files then use command line tools (Java Development Kit and Apache Ant are required) to create, build and debug Android applications as well as control attached Android devices (e.g., triggering a reboot, installing software package(s) remotely).

Enhancements to Android's SDK go hand in hand with the overall Android platform development. The SDK also supports older versions of the Android platform in case developers wish to target their applications at older devices. Development tools are downloadable components, so after one has downloaded the latest version and platform, older platforms and tools can also be downloaded for compatibility testing.

Android applications are packaged in .apk format and stored under /data/app folder on the Android OS (the folder is accessible only to root user for security reasons). APK package contains .dex files (compiled byte code files called Dalvik executables), resource files, etc.

2.9 Xcode[12]

Xcode is a tool for developing software for Mac OS X. Xcode developed by Apple and the latest version is Xcode 4.2 which available on the Mac App Store for Mac OS X 10.7.

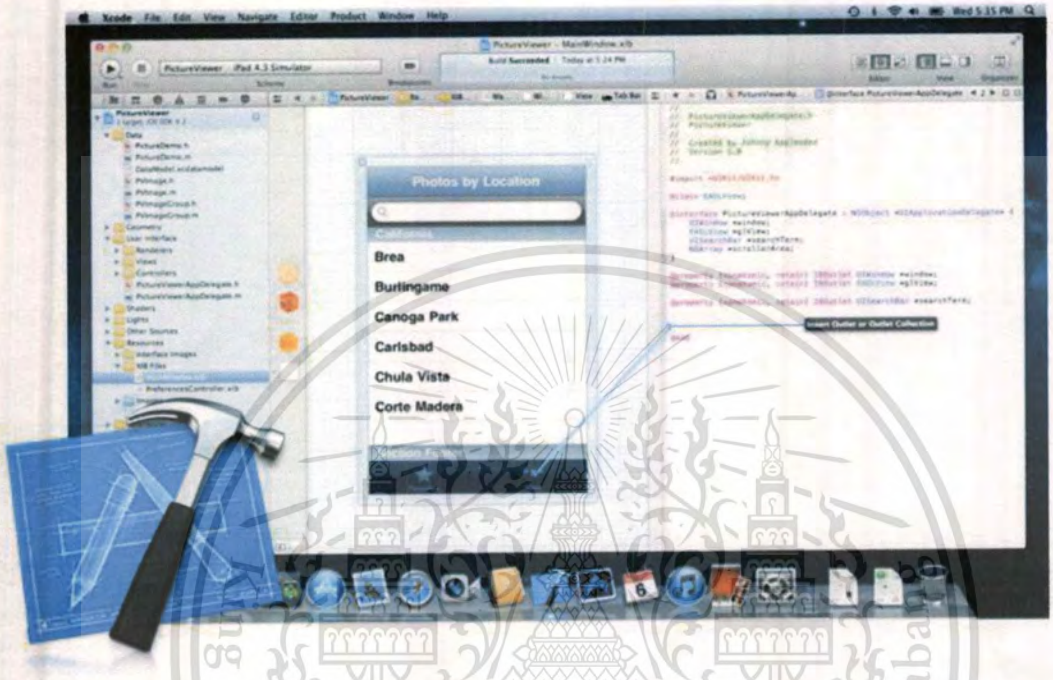


Figure 2.4 Xcode interface

2.9.1 Xcode IDE

The main application of Xcode is the integrated development environment (IDE), also named Xcode. The Xcode suite also includes most of Apple's developer documentation, and Interface Builder, an application used to construct graphical user interfaces.

The Xcode suite includes a modified version of free software GNU Compiler Collection, and supports C, C++, Objective-C, Objective-C++, Java, AppleScript, Python and Ruby source code with a variety of programming models, including but not limited to Cocoa, Carbon, and Java. Third parties have added support for GNU Pascal, Free Pascal, Ada, C#, Perl, Haskell, and D. The Xcode suite uses the GNU Debugger as the back-end for its debugger.

2.9.2 Apple LLVM Compiler

Apple's next generation compiler technology, the Apple LLVM compiler, does more than build application. Apple LLVM technology is integrated into the entire development experience. The same parser used to build C/C++ and Objective-C powers Xcode's indexing engine, providing accurate code completions. As working, Apple LLVM is constantly evaluating and identifying coding mistakes that Xcode shows as Live Issues, and thinking ahead for ways to Fix-it.

2.9.3 iOS Simulator

The iOS Simulator can run application in the quite a similar way to an actual iOS device. Because it is quick to launch and debug, the iOS Simulator is used for testing, to make sure your user interface works the way intended, the network calls are correct, and that the views change correctly when the phone rotates. It can even simulate touch gestures by using the mouse.

2.10 Eclipse[13]

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, other programming languages including Ada, C, C++, COBOL, Perl, PHP, Python, R, Ruby (including Ruby on Rails framework), Scala, Clojure, Groovy and Scheme. It can also be used to develop packages for the software Mathematica. The IDE is often called Eclipse ADT (Ada Development Toolkit) for Ada, Eclipse CDT for C/C++, Eclipse JDT for Java, and Eclipse PDT for PHP.

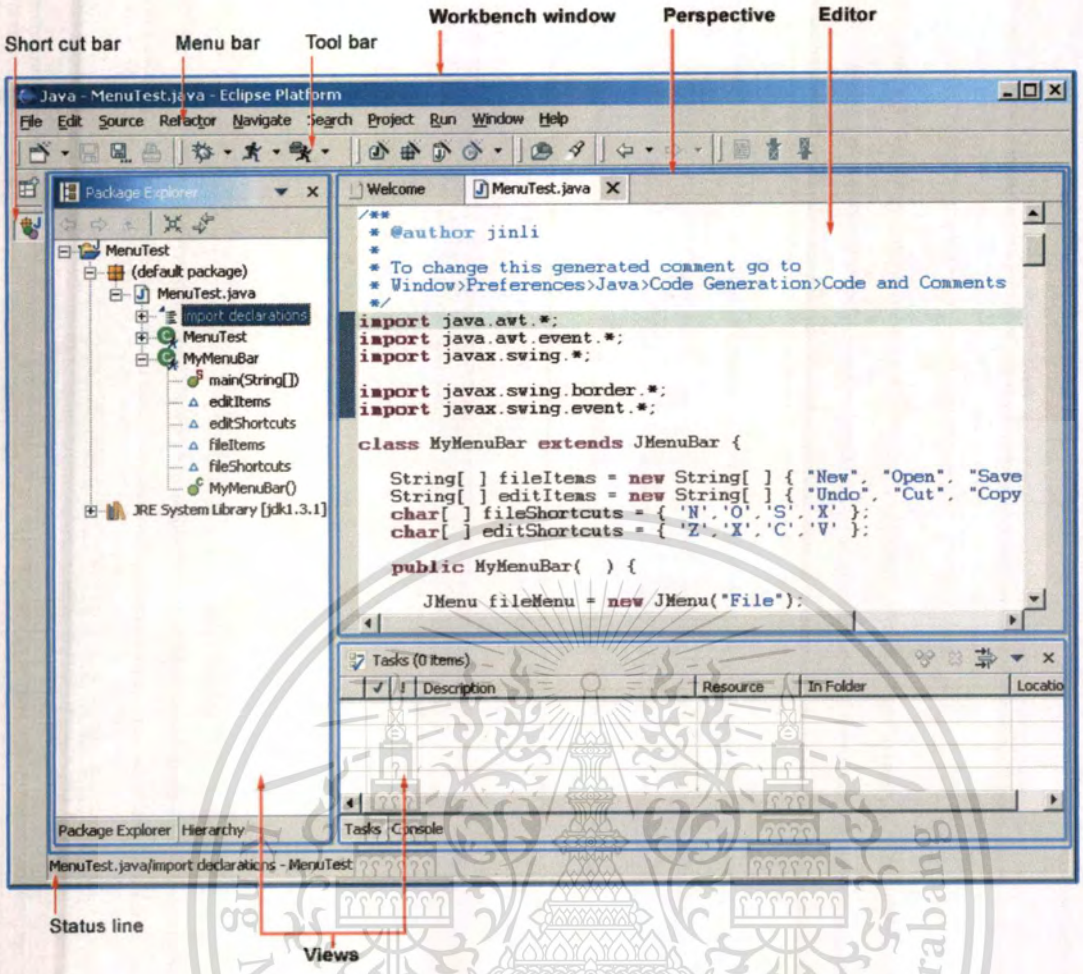


Figure2.5 Eclipse interface

Chapter 3

System Design

This chapter will give the design and analysis of our game.

3.1 Game Overview

Basically, our game consists of two players. When each player enters the game, he has to set-up his character such as choosing his name and his spaceship. Because it is a network game, one player must choose to perform the host duty where another will perform the join function. In each of the player turn, he will roll the dice, move his token on the board according to the numbers he has got and perform either an event's action or enter a battle mode in mini-game depending on the system. Once the player finishes his turn, he stops and waits for his next turn. The player will not make any move until his turn comes except his opponent drags him into the fight in the Mini-games. For the normal event the current player will resolve his event first and then the update form the current player's event will send to waiting player. The winner of this game is determined from the player who reaches the finishing line first.

The system uses the random method to decide whether it is going to choose an event or a mini-game. The event actions are actions which the player himself has to perform. The event can be either good or bad. The mini-game is a game that the player has to fight with another player. The loser of the mini-game will get punishment such as move back or stop for x turn. The winner of the mini-game has not received any reward yet. Instead, he has to perform another random event chosen by the system. Similarly, this event can be either good or bad.

In our system, there are normal events that affect the players on the board and mini-games event which each player must compete to get the reward.

3.1.1 Events

The event is an action that will affect the position or condition of each player. After the event has been selected the event will affect the value of the event.

The examples of the event are:

1) Randomly move backward or forward. This type of event will move player's token forward or backward for several space

2) Randomly move vector and movement point

3.1.2 Mini-games

Mini-games is the augmented game in our main game which will be called when the system chooses Mini-games which each player needs to compete in the Mini-games to get reward which can be good or bad to either player depend on the choice of reward they choose.

The mini-games that will put in our game are:

1) Evasive Mini game: The Evasive mini-game assumes both players caught in the ion storm and need to evade the hit from the storm by evasion it. In the game we assume evasion by tap the button sequentially in the limited time
2) Energy Charger Mini-games: The Energy Charger will assume that the player's ship has encountered the power failure situation and needs to complete power charging before their rival does it. Each player needs to tap as fast as possible. After 5 second passed, the tab counter will be compared by the game system. The winner will have his chance to gain advantage in their journey or to make a disadvantage to his rival.

3.2 Use Case Diagrams of Nyorion game

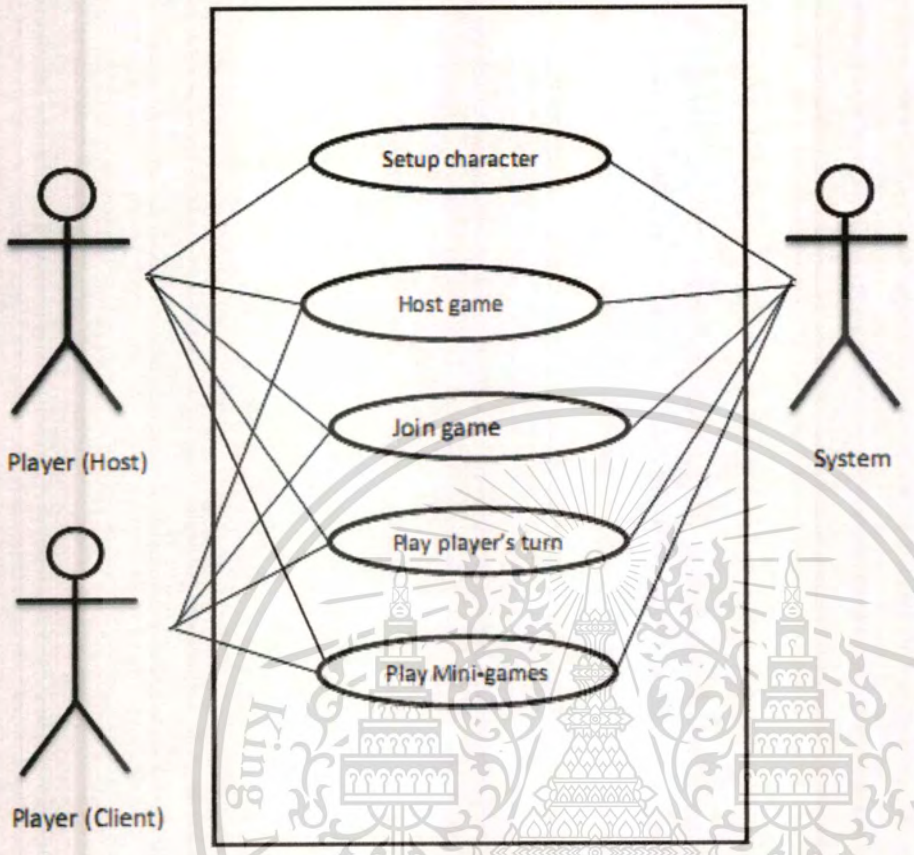


Figure 3.1 Game overall Use case diagrams.

We will use the Use case diagram to describe actions that each player can interact with the system. The main action that every player has to execute consists of: Setup character, Host game, Join game, Play their turn, and Play mini game. The descriptions of each action will be given in the use case description Table 3.1 to Table 3.16.

3.2.1 Setup Character

Setup Character is an action in which a player sets up his name and chooses a spaceship before starting the game.

Table 3.1 Use Case Description Set up character.

Use case name	Set up character	
Scenario	Function to set up character	
Triggering event	Open game	
Brief description	Player sets his name and chooses a spaceship before start playing the game	
Actor	Player	
Precondition	-	
Postcondition	Show result	
Flow of event	Actor	System
	1.Set name and choose spaceship	2.Save information
Alternative flow	-	

3.2.2 Host game

Host game is a part where a player chooses to be a host of the game. Hence, its system needs to open a connection waiting for other player's system to connect.

Table 3.2 Use Case Description Host game.

Use case name	Host game	
Scenario	Function for host game	
Triggering event	Host	
Brief description	Player chooses function to host the game (to be a server) and wait for client to connect to server	
Actor	Host	
Precondition	-	
Postcondition	Wait for client to connect	
Flow of event	Actor	System
	1.Choose function to be a host of the game.	2.Open connection and wait for client to connect
Alternative flow	Client side cannot find the server	

3.2.3 Join Game

Join game is a section where the player who decides to be the client make a connection to the host player.

Table 3.3 Use Case Description Join game.

Use case name	Join game	
Scenario	Function for player(client) to join the game	
Triggering event	Client	
Brief description	Player choose function to join the game to connect to the host	
Actor	Client	
Precondition	-	
Postcondition	Wait for the server(host) response	
Flow of event	Actor	System
	1.Choose function to join the game	2. Create connection to the server(host)
Alternative flow		

3.2.4 Play player's turn

This is the part that describes each player actions when he plays his turn which are rolling dice, walking and either doing an event or playing a mini game.

Table 3.4 Use Case Description Play player's turn.

Use case name	Play player's turn	
Scenario	Roll dice, walk and do system's randomly chosen decision (event or mini game)	
Triggering event	Player choose to move	
Brief description	Player roll dice to specify number of move	
Actor	Player, System	
Precondition	Start game	
Postcondition	Play mini-game or event	
Flow of event	Actor	System
	1. Roll the dice	2.Random number of move 3.Move player token
Alternative flow	-	

3.2.5 Play Mini-game

The current player's system must also randomly pick which mini-game the player will play. Then, this mini-game will be started on both player's devices. Each player must with his opponent fight to win.

Table 3.5 Use Case Description Play Mini-game.

Use case name	Play Mini-game	
Scenario	Start Mini-game	
Triggering event	System randomly chooses mini-game	
Brief description	Start Mini-game for Host and Client	
Actor	Host and Client	
Precondition	Player move to space	
Postcondition	Show result	
Flow of event	Actor	System
		1. Start game
Alternative flow	One of play connection is drop out	

At this stage, there are six mini-games. Use case diagrams in Figure 3.2 - 3.7 describe actions that the player interacts with the system when he plays mini-game.

3.2.5.1 Energy Charger Mini Game

There are two interactions: Tab and Choose action.. See section3.3.4.3for more details.

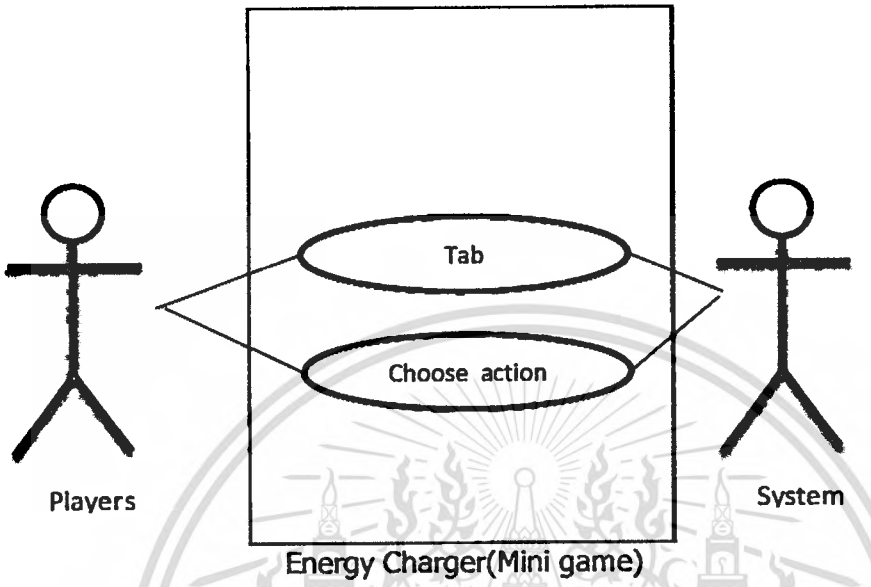


Figure 3.2 Energy charger mini game use case diagram.

Table 3.6 Use Case Description Tab.

Use case name	Tab	
Scenario	Function for tab	
Triggering event	Players start mini-game	
Brief description	Player tab screen to win other play.	
Actor	Players	
Precondition	Players move to space	
Postcondition	Choose action	
Flow of event	Actor	System
	1.Player tab screen	2.System count tab 3.System determine who is winner
Alternative flow		

Table 3.7 Use Case Description Choose action.

Use case name	Choose action	
Scenario	Function for Choose action	
Triggering event	Winner	
Brief description	Winner can Choose reward from mini-game	
Actor	Winner	
Precondition	Win mini-game	
Postcondition	Show result	
Flow of event	Actor	System
	1.Winner choose action	2.System do action 3.Show result of action
Alternative flow		

3.2.5.2 Space Race Mini Game

There are two interactions: Tab and Choose action.. See section 3.3.4.5for more details.

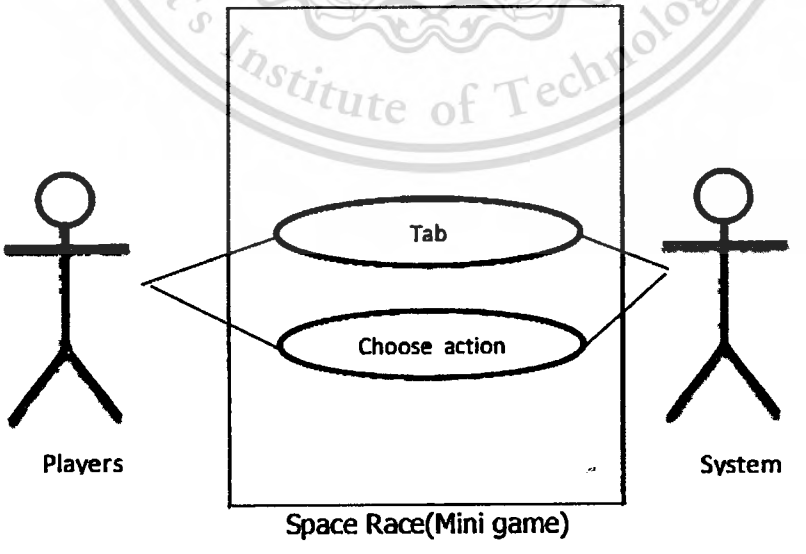


Figure 3.3 Space race mini game use case diagram.

Table 3.8 Use Case Description Tab.

Use case name	Tab	
Scenario	Function for tab	
Triggering event	Players start mini-game	
Brief description	Player *tab screen* to win other play.	
Actor	Players	
Precondition	Players move to space	
Postcondition	Choose action	
Flow of event	Actor	System
	1.Player tab screen	2.System count tab 3.System determine who is winner
Alternative flow		

Table 3.9 Use Case Description Choose action.

Use case name	Choose action	
Scenario	Function for Choose action	
triggering event	Winner	
brief description	Winner can choose reward from mini-game	
actor	Winner	
Precondition	Win mini-game	
Postcondition	Show result	
Flow of event	Actor	System
	1.Winner choose action	2.System do action 3.Show result of action
Alternative flow		

3.3 Flowcharts Activity Diagrams and Game Descriptions

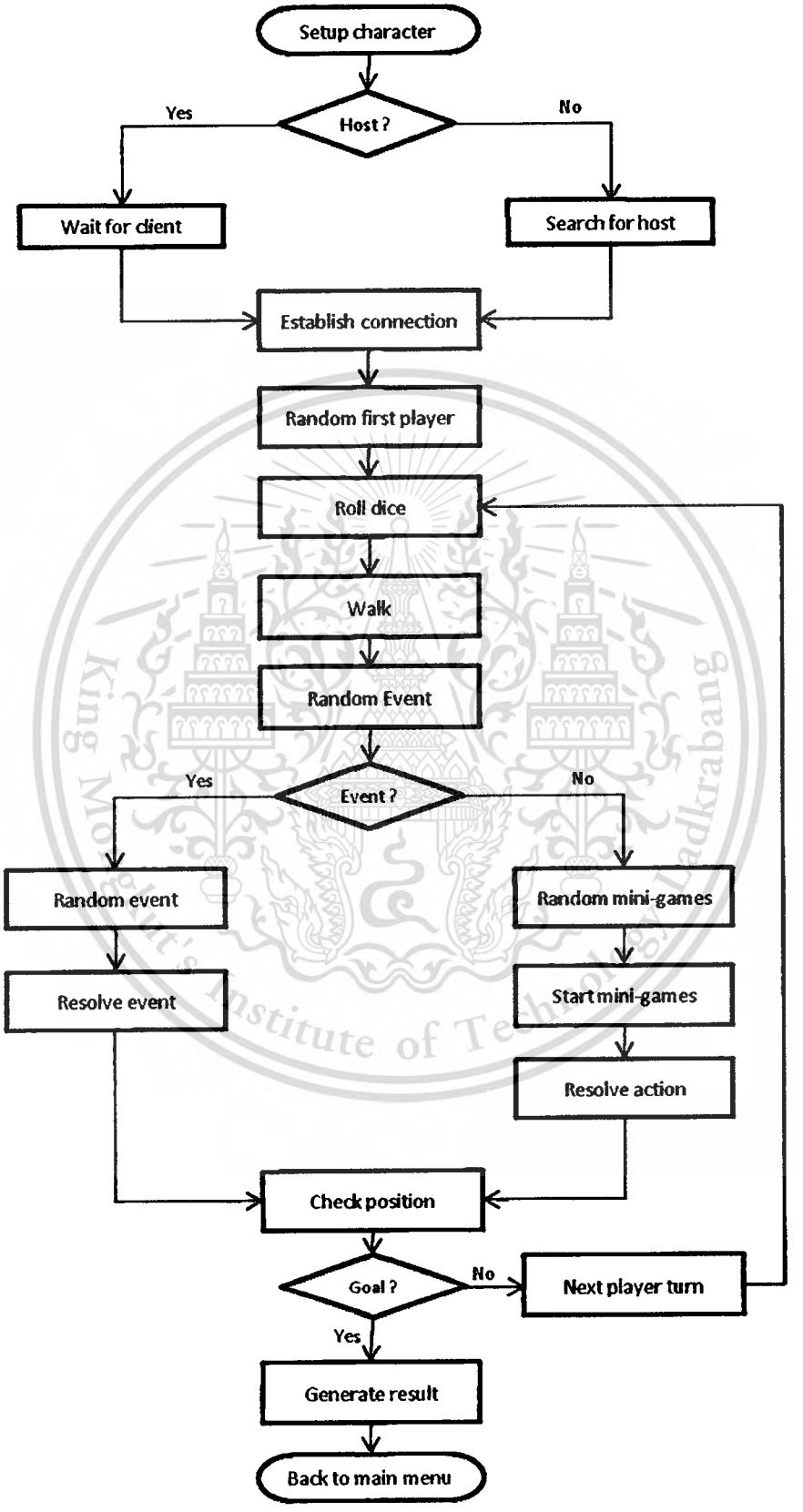


Figure 3.4 Game Flowcharts.

The flowchart in Figure 3.8 describes the overview of the game processes. After both players enter the character setup menu, which they can choose names and tokens to represent themselves in the game, one of the players must choose to host the game. Hence, another player will join the game as a client. After their connection established, the host device will randomly choose the first player. If the client device is the first player, the signal will be sent to the client device to activate his turn. The first player plays his turn by rolling the dice to obtain the point to walk. The player token will be moved according to their movement point. Then, the system decision model will randomly make a decision. The decision can be either the event or the mini-game. If the system decision is the event, the event will be randomly generated and shown to both players for them to resolve it on the current player first then the current player system will send the change update to another player. The event can be good or bad for either player or for both of them. If the system decision chooses the mini-game, the system will send mini-game triggering signal to tell another player's system to start this mini-game session. Both players will compete in the mini-game to gain advantage in their journey or make a disadvantage to their rival, which only the winner of the mini-game can choose this action. After the action has been resolved, the system of the current player will check the position of both player's token. If either player token is on the finishing space, the system will generate the result: who is the winner and who is the loser and will send the signal to notify the result to another player system. If there is still no winner, the system will end the player turn and send signal to trigger another player to start his turn.

3.3.1 Host Activity Diagram

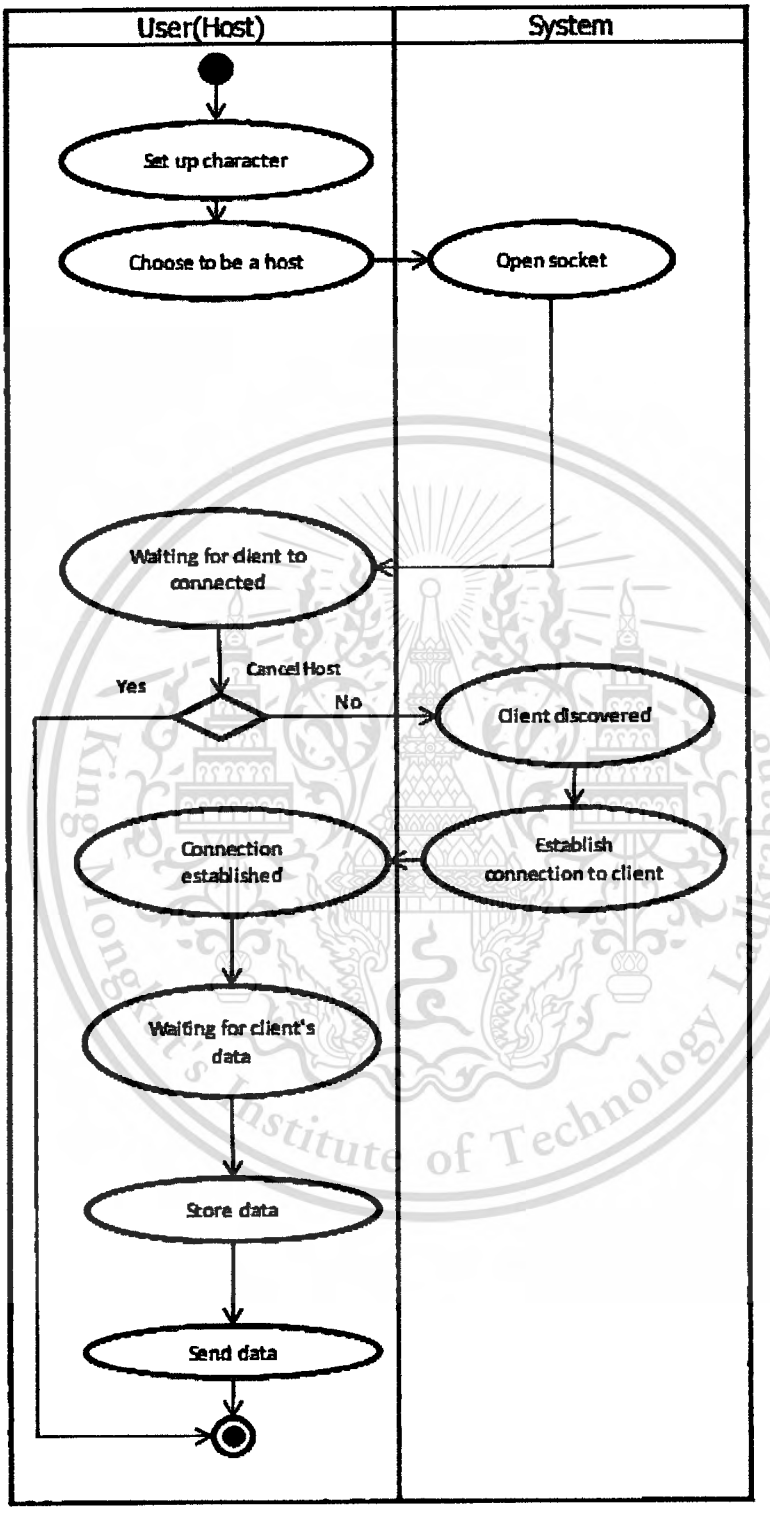


Figure 3.5 Host Activity diagram.

This diagram will show how host device wait for the client's connection. First the player sets up his character and chooses his token. Then, the player chooses to be a host of the game. Hence, his system will listen for the client connection. While the host is still waiting, the player can choose to cancel the wait for the client. After discovering the client, the connection will be established and the host will wait for the client data (name and token). After receiving this data, the host will store client data in his device and send his data (name and token) to the client device.

3.3.2 Client Activity Diagram

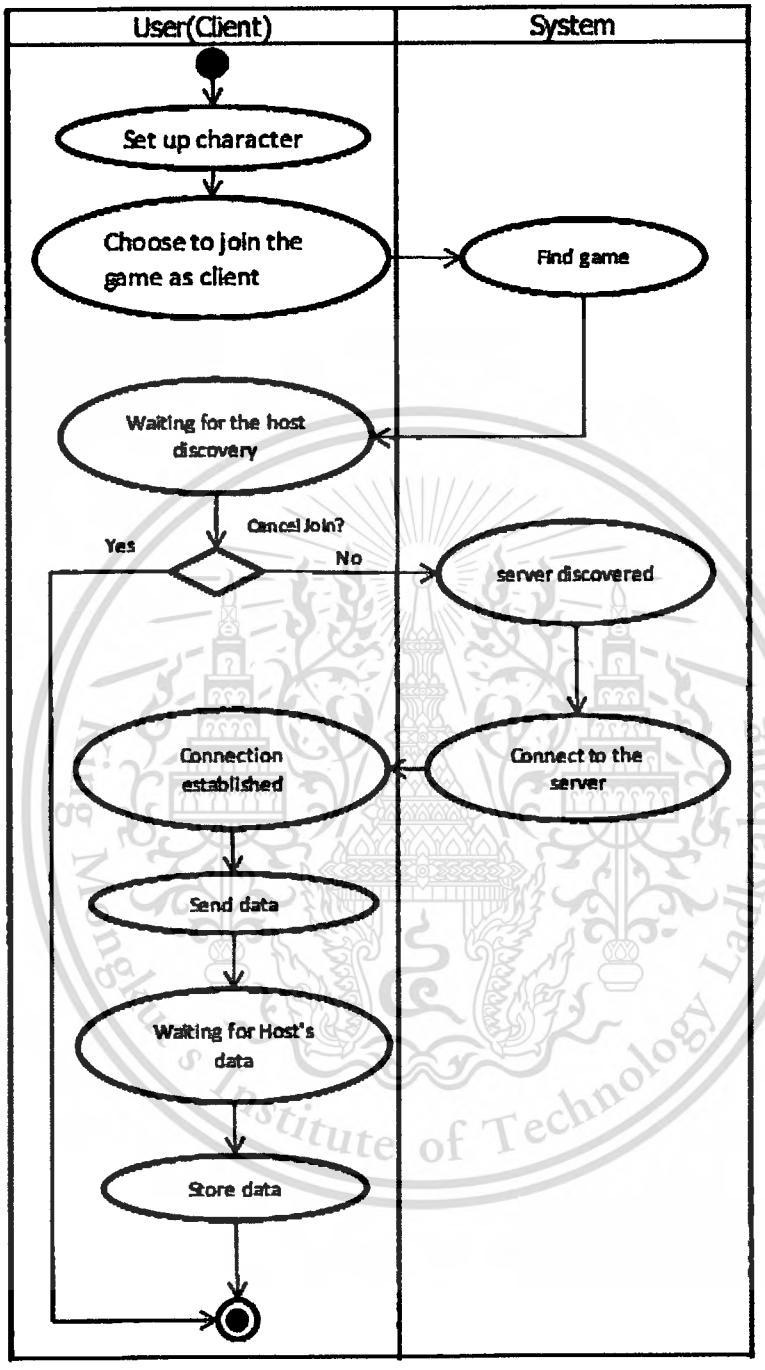


Figure 3.6 Client Activity diagram.

This diagram will show how the client device finds the host. First the player sets up his character and chooses his token. The player chooses to join the game so his system will try to connect to the host. While the client tries to connect to the host, the player can choose to cancel it. After the connection is

established, the system will send his data (name, token) to the host device and then it will wait for the host data (name, token). After receiving the host data, the system will store in the client device.

3.3.3 Core Game Activity Diagram

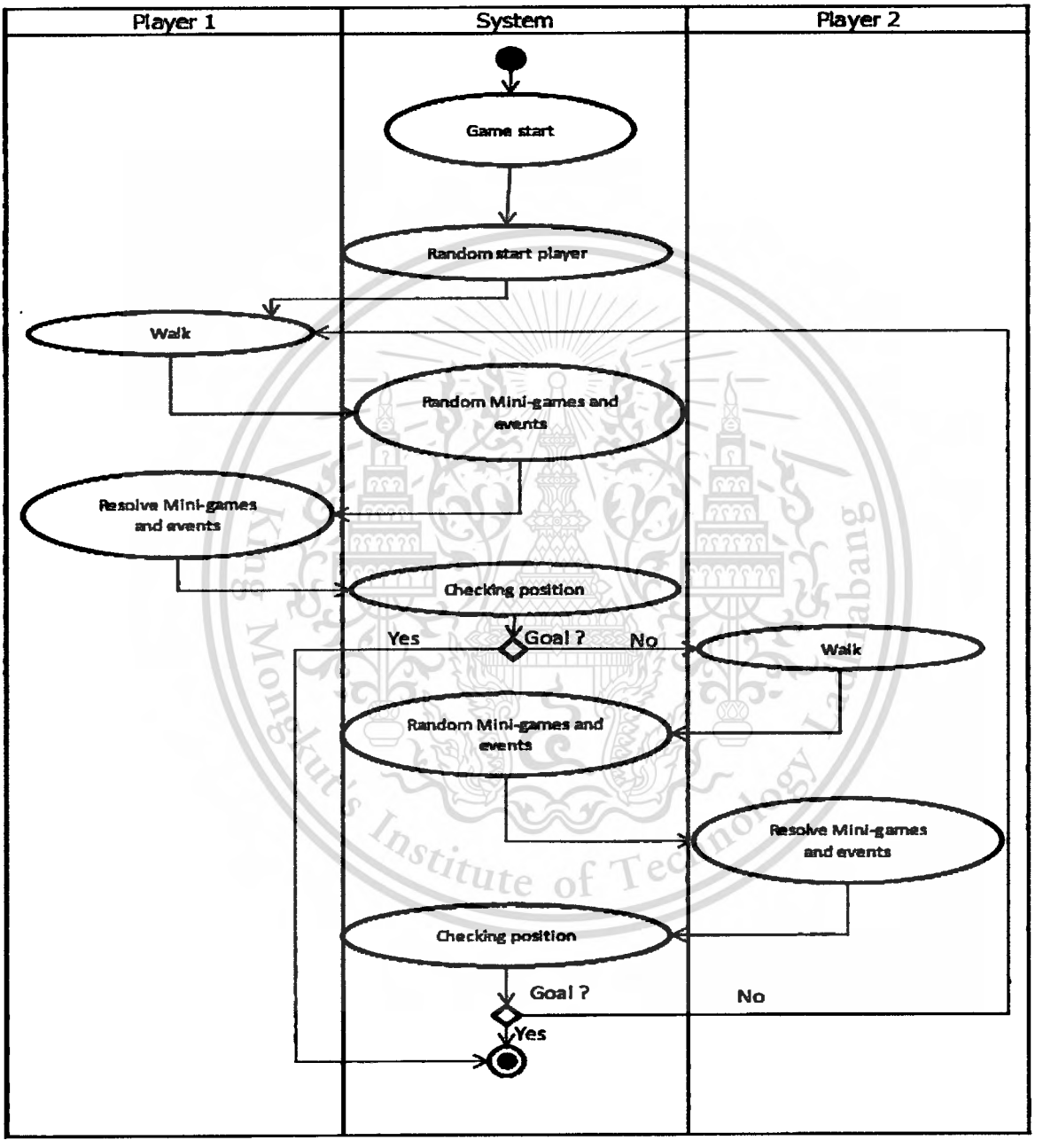


Figure 3.7 Core game Activity diagram.

The diagram in Figure 3.11 is the core game activity diagram which will explain the flow of the system. After the game starts, the system will randomly determine the first player and sends the signal with flag to notify another player system for this order of the play (player 1, player 2). Then, the player 1 will move first. After the player 1 moves, the system of the player 1 will randomly generate an event or a mini-game for both player and will send the event or mini-game flag to identify system decision to another player. While resolving the event, the current player will update his game position in his device first then will send the updated position to another player. For the mini-game the winner of that mini-game will update his position first then will send the updated position to another player. After resolving the event or mini-game, the current player system will check both the token positions. If there is a token on the finishing space, the game will end. Otherwise, the game will continue to the next player turn by sending the turning flag to that player.

Form the Figure 3.12 – 3.17 will show the activity diagrams of the mini-games and explain how to play each game.

3.3.4 Mini Game Activity Diagrams

3.3.4.1 Energy Charger Mini-game Activity diagram.

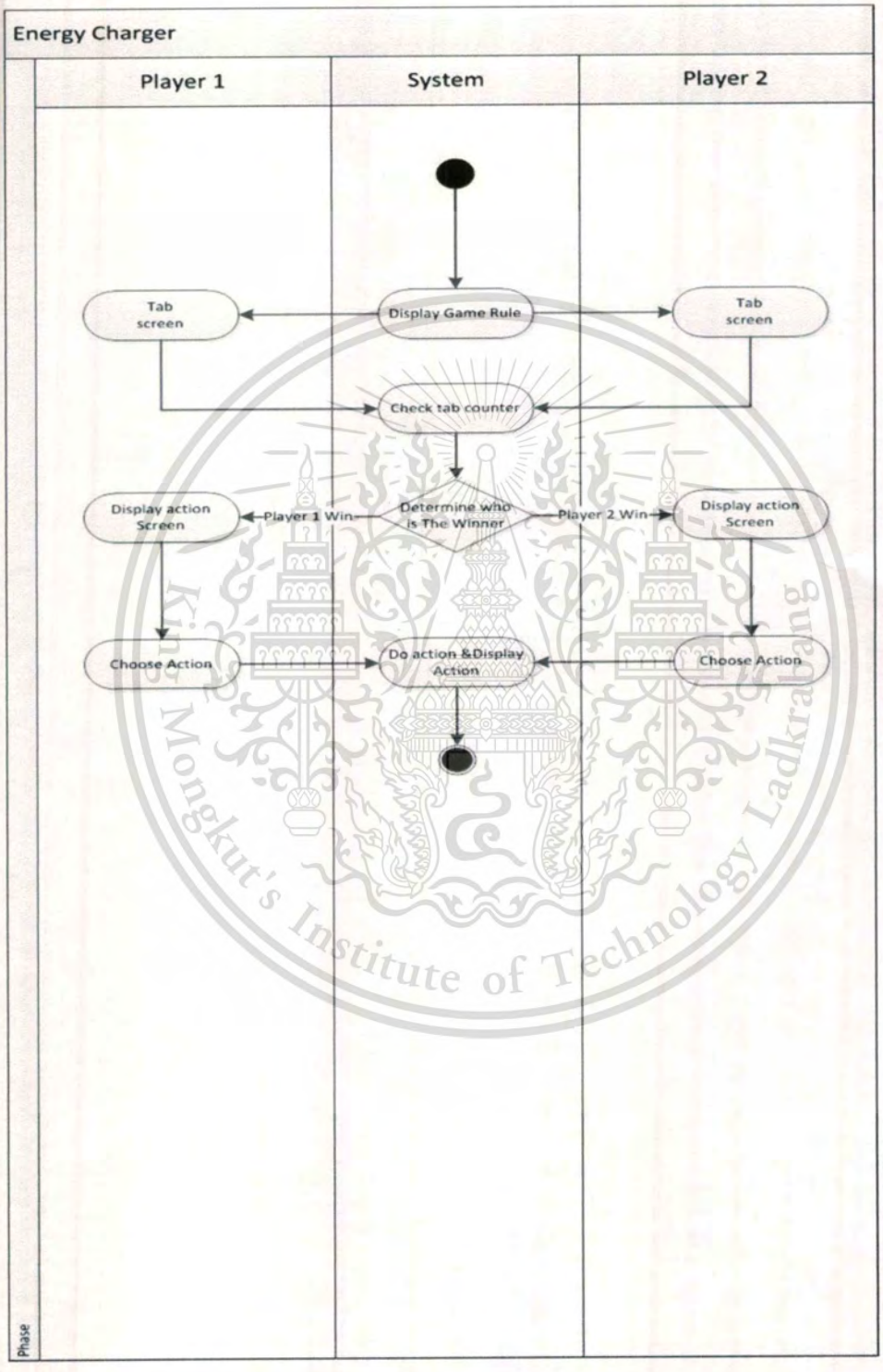


Figure 3.8 Energy charger Mini-game Activity diagram.

3.3.4.4 Space Race Mini-game Activity diagram.

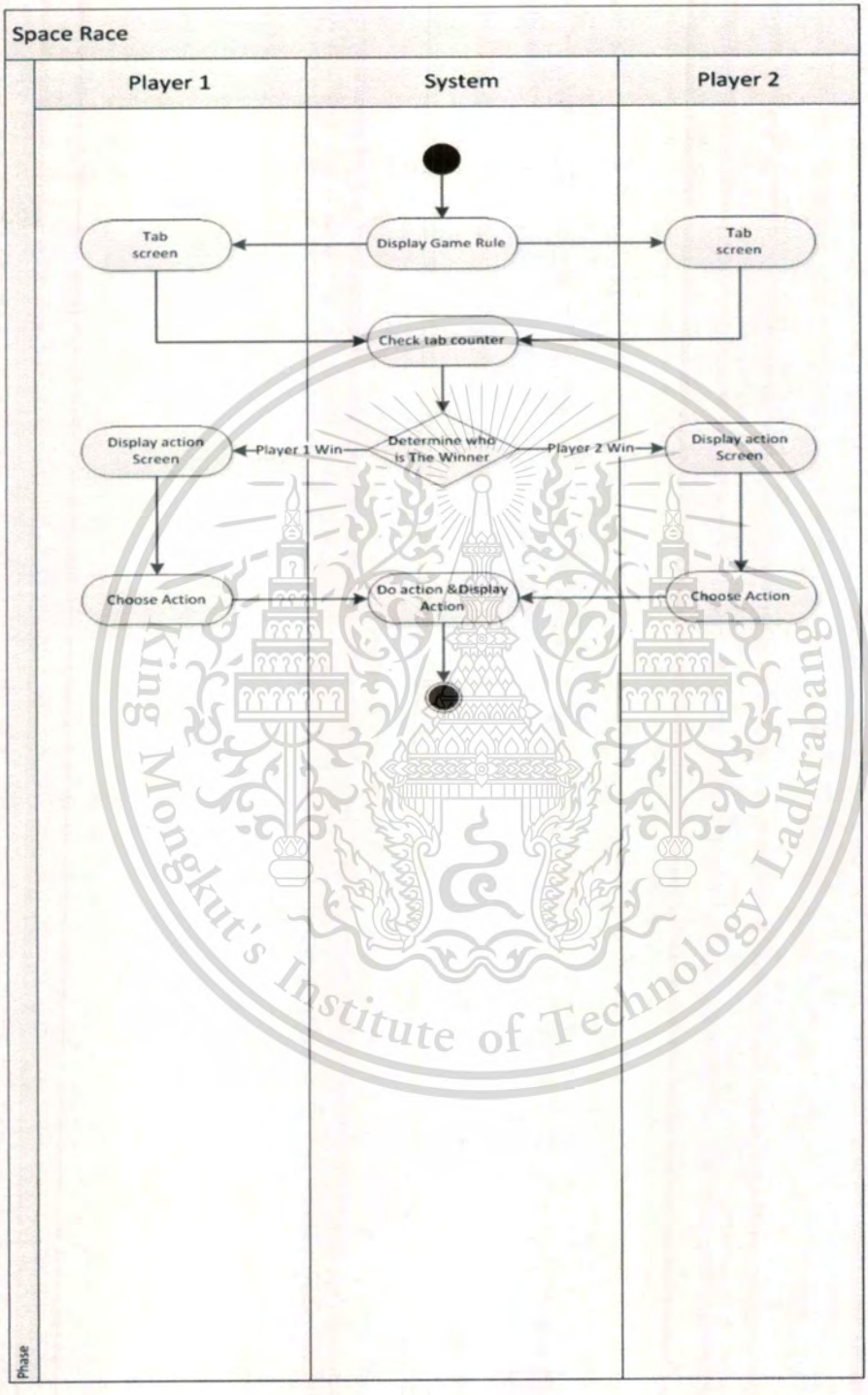


Figure 3.9 Space race Mini-game Activity diagram.

3.4 Game Interface

Figure 3.10 Example of game interface

The game will be simple which player don have to do much. The game flows sequentially by game model. First player need to roll the movement point and then move their token then the events or mini-games will randomly generate. After resolve the events or mini-games the game will send the update to the other player.

3.5 Xcode Programming[14]

Xcode is the latest iteration of Apple's integrated development environment (IDE), a complete toolset for building Mac OS X and iOS applications. The Xcode IDE includes a powerful source editor, a sophisticated graphical UI editor, and many other features from highly customizable builds to support for source code repository management. Xcode can help you to identify mistakes in both syntax and logic, and will even suggest how to fix those mistakes. Xcode features a single window, called the workspace window as shown **Figure 3.11.**

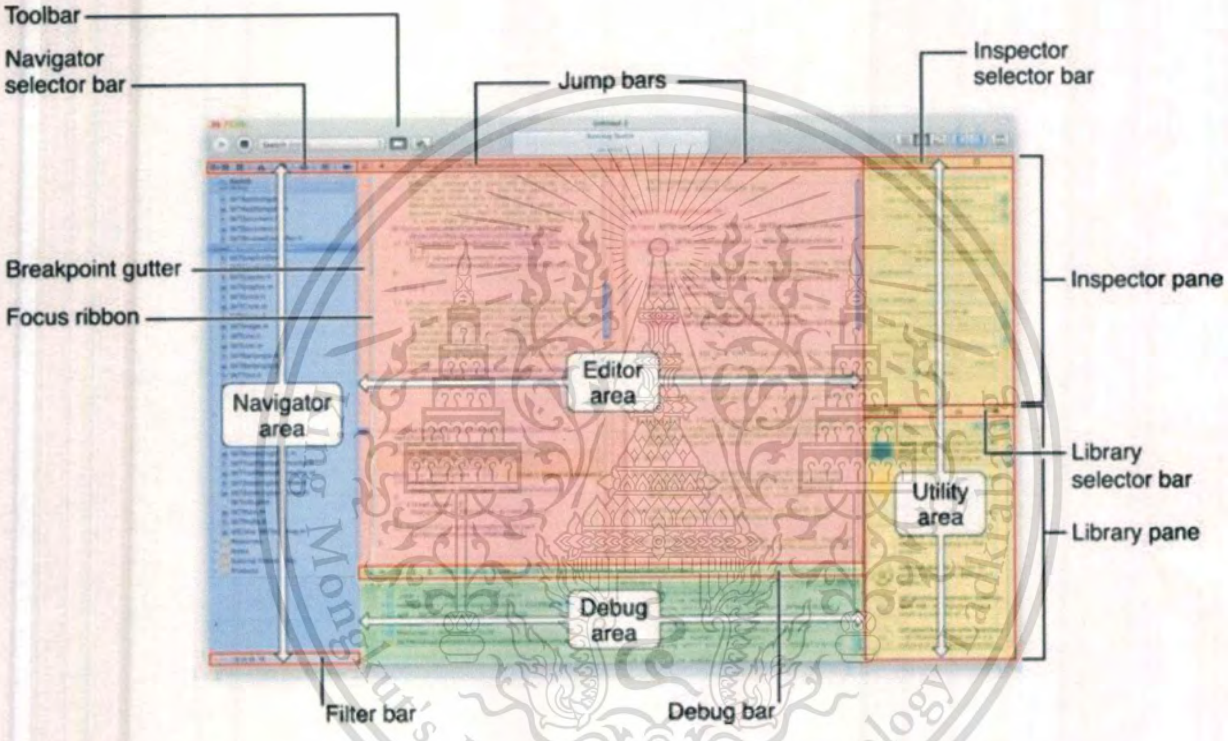


Figure 3.11 Workspace Window

Interface Builder is a fully integrated graphical tool for designing user interfaces.

When creating a new Xcode project, you can name the project, identify the company identifier and the device you wish to implement. There are also options which you can choose like unit tests, storyboard or an automatic reference counting as in **Figure 3.12**

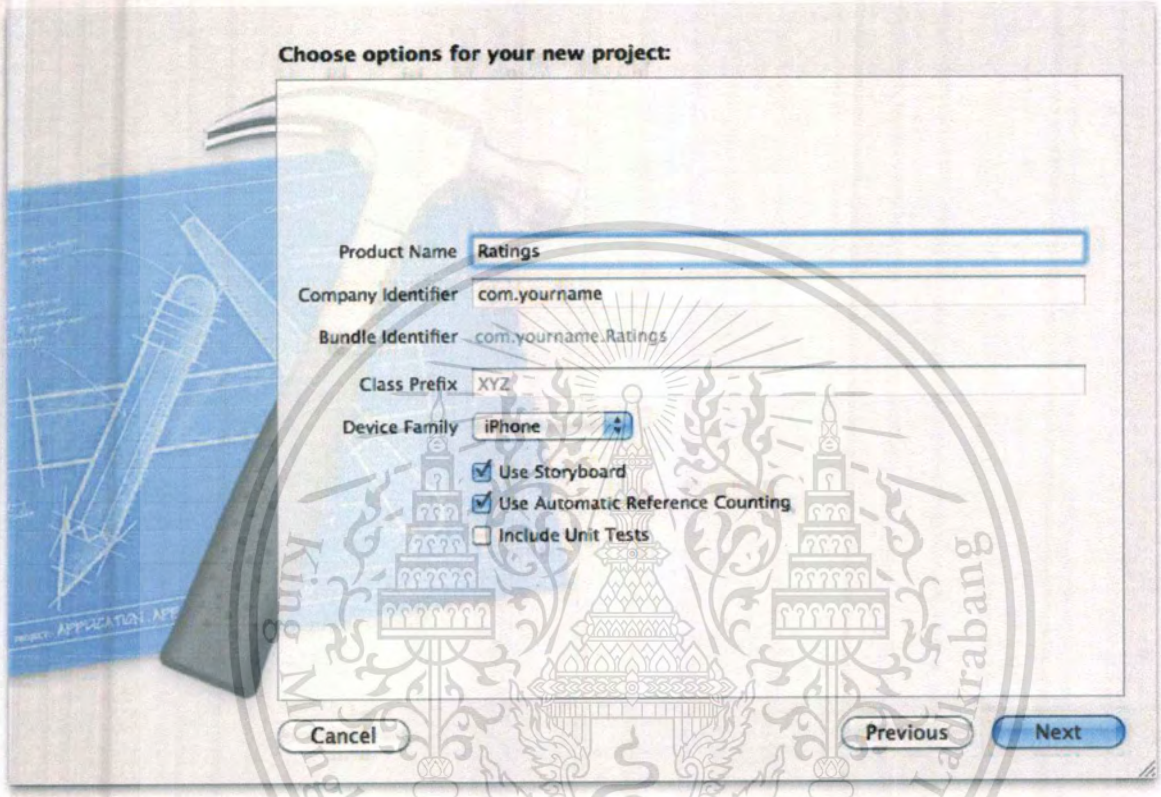


Figure 3.12 Options of newly created project

When you create a project a workspace will be generated for you. On the right side of a Figure is the editor area. It can edit many types of information, including source code, property lists (*plists*), core data models, and user interface (*nib* or storyboard) files, and you can view many more.

Figure 3.13 Options of newly create project

On the left side of the window in **Figure 3.14** is the navigator area. The project navigator shows the project's contents or workspace.

Figure 3.14 Navigator area

To supplement the information in the editor area, you can open a utility area at the right of the workspace window as in **Figure 3.15** , which includes inspectors and libraries. Use the view selector ( ) in the toolbar to open and close the navigator, debug, and utility areas.

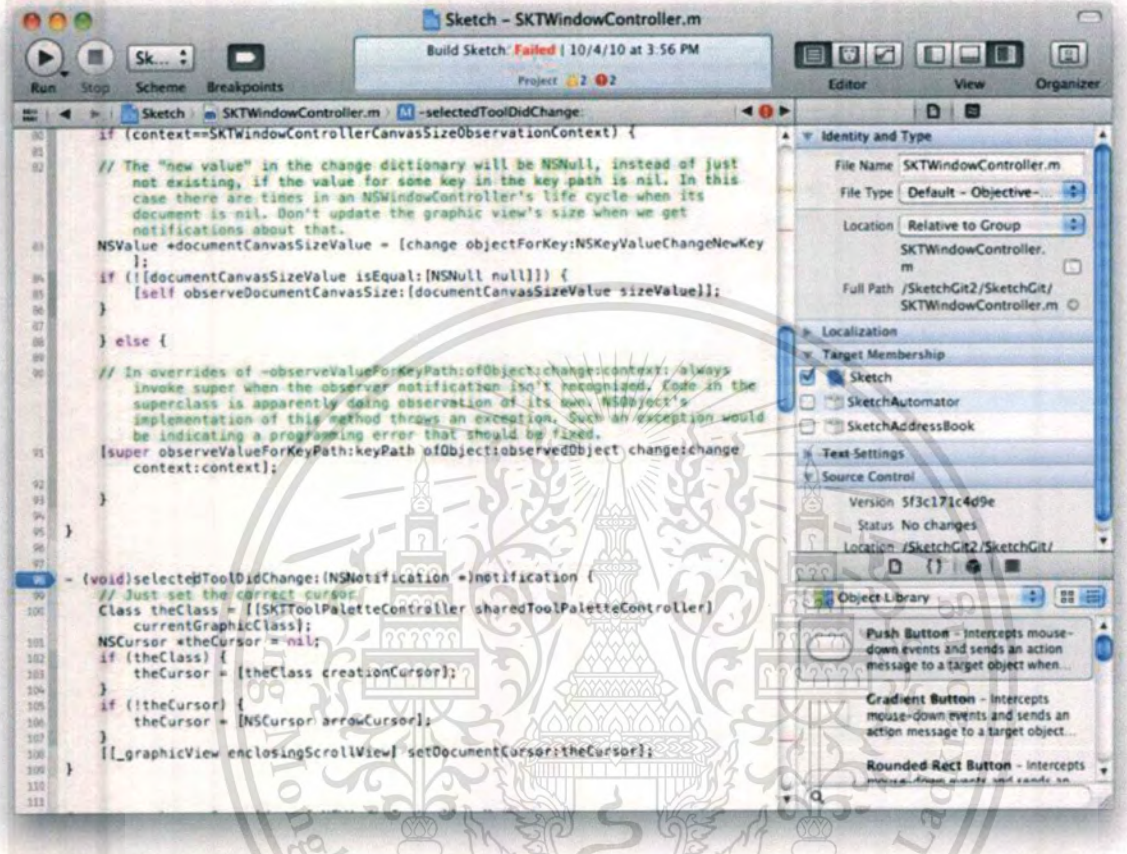


Figure 3.15 The utility area

When running the code, the debug area opens as shown in **Figure** 3.16. When the code stops at a breakpoint, the debug navigator opens as well.

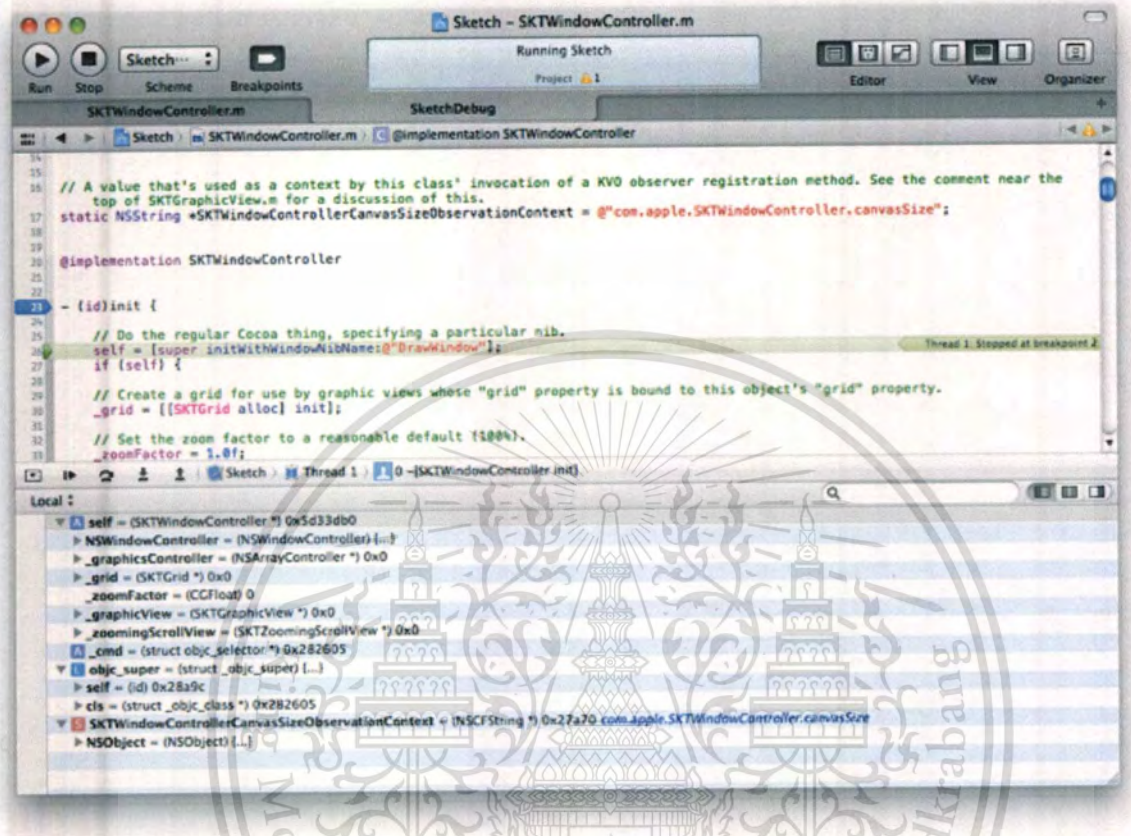


Figure 3.16 The debug area

Xcode has two main windows: the workspace window which is used for editing, debugging, and interface design, and the *Organizer window*, which is used to display documentation, source control, project organization, and for iOS, access to your mobile devices. To display the Organizer window, choose *Window* > *Organizer*, or click the Organizer button in the toolbar **Figure** 3.17.

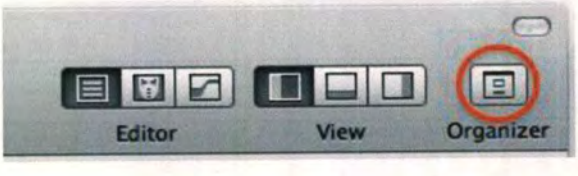


Figure 3.17 The Organizer button

The Xcode is an Objective-C programming tool which uses C based language. In the basic view base project, there are three basic types of files that are the core of the Xcode programming the *.h* ,.*m* and *.xib* files. The *.h* files which is also called the header file is where the variable, constants, and functions that are used by other files within a programming project are declared and also to declare the property of them. Next is *.m* file which is the Class implementation file used by programs written in Objective-C, and begins with the @implementation directive to initialize variables and functions that can be referenced by other Objective-C source file or header (*.h*) files. Last the *.xib* file. It is an application interface created by the Interface Builder, a graphical editor for designing and testing user interfaces.

One of the core things in Xcode programming is *Appdelegate.Appdelegate* is an application delegate or helper object that takes responsibility or aids in some of the tasks that an object is responsible to the *UIApplication*(which is the iOS singleton application base class)object. It is normally used to perform tasks on application startup and shutdown, handling URL open requests and similar application-wide tasks.

3.5.1 Server Socket

3.5.1.1 Server Socket overview[15]

Steps to generate a server socket can be done as follows:

1. Constructs a BSD server socket through calling the *listen* function.

2. Creates a *CFSocket*, using the BSD native server socket, with a *CFRunLoop* (i.e. thread) to accept and notify the method *acceptConnection* when connection is made.

3. Starts the *CFRunLoop*. This means that the server is waiting for a connection.

4. When connection is made, *acceptConnection* is called, which creates a *CFSocket* for sending and receiving data, with a *CFRunLoop* (i.e. thread) to notify the method *receiveData* when data is received.

5. Starts the *CFRunLoop*, the server is waiting for data in the background thread and change the view to the mainboard game

6. When data is received from a client, *receiveData* will be called while the current view is in the *WaitingTurn* of server to avoid the repeating of allocating memory and cause the data to lose.

```
-(IBAction)hostGame :(UIButton *)sender;{
structsockaddr_in sin;
        int sock, yes = 1;
        CFSocketRef s;
        CFRunLoopSourceRef source;
        sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        memset(&sin, 0, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_port = htons(6666);
        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
        setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &yes, sizeof(yes));
        bind(sock, (structsockaddr *)&sin, sizeof(sin));
        listen(sock, 5);
        s = CFSocketCreateWithNative(NULL, sock, kCFSocketAcceptCallBack, acceptConnection, NULL);
        source = CFSocketCreateRunLoopSource(NULL, s, 0);
        CFRunLoopAddSource(CFRunLoopGetCurrent(), source, kCFRunLoopDefaultMode);
        CFRelease(source);
        CFRelease(s);
        CFRunLoopRun();
WaitingServerTurn *WaitingServer = [[WaitingServerTurnalloc] initWithNibName:nilbundle:nil];
   [selfpresentModalViewController:WaitingServeranimated:YES];
}
```

Figure 3.18 Action of Host Button1

3.5.1.2 Server Socket detail

From the fist line, the *IBAction* means that we have declared a method called *host Game*, and type of the message which will be sending to the *host Game* as a *UIButton* type. Then, we declare the variable named *sock* with the socket address *struct type* which creates the reference of the socket. Next, we declare a source as the *CFRunLoopSource* to tell the abstract of the object that can put to run loop. Then, the socket reference will be created using "socket(AF_INET, SOCK_STREAM,0);". AF_INET in Xcode is the family of IP address which normally is the IPV4. *htons(6666)* means set this connection port to 6666.SOCK_STREAM tells that the connection type is TCP. *bind(sock, (structsockaddr *)&sin, sizeof(sin))*and *listen(sock, 5)*tell the program to bind the socket and start to listen for a connection on *sock* and socket number 5. The *CFSocketRef* will generate required data using the sock and set to accept the call back type with the *acceptConnection* method which will be called when there is a connection. Then, the source object will put to run loop to start to listen to the connection.

```
voidreceiveData(CFSocketRef s, CFSocketCallBackType type,
CFDataRef address, const void *data, void *info)
{
(//do logic check for event)
NSString *buffer = [[NSStringalloc] initWithData:(NSData *)data encoding:NSUTF8StringEncoding];
WaitingServerTurn *waiting = [[WaitingServerTurnalloc] initWithNibName:nilbundle:nil];
   [waitingactivateButton:buffergetAddress:addresssetSockRef:s];
}

vòidacceptConnection(CFSocketRef s, CFSocketCallBackType type, CFDataRef address, const void *data,
void *info)
{
CFRunLoopStop(CFRunLoopGetCurrent());
        CFSocketNativeHandlecsock = *(CFSocketNativeHandle *)data;
        CFSocketRefsn;
        CFRunLoopSourceRef source;
        NSLog(@"Server socket %d received connection socket %d\n", CFSocketGetNative(s), csock);
        sn = CFSocketCreateWithNative(NULL, csock, kCFSocketDataCallBack, receiveData, NULL);
        source = CFSocketCreateRunLoopSource(NULL, sn, 0);
        CFRunLoopAddSource(CFRunLoopGetCurrent(), source, kCFRunLoopDefaultMode);
        CFRelease(source);
        CFRelease(sn);}
```

Figure 3.19 *acceptConnection* and receiveData method

When there is a connection, the *accepConnection* will be called which then will generate the socket reference of the incoming connection, address and data. Next, another loop will be created in order to

wait for the incoming data from the client. This socket (having *CFSocketRef*type) is used as a pipe to send and receive data.

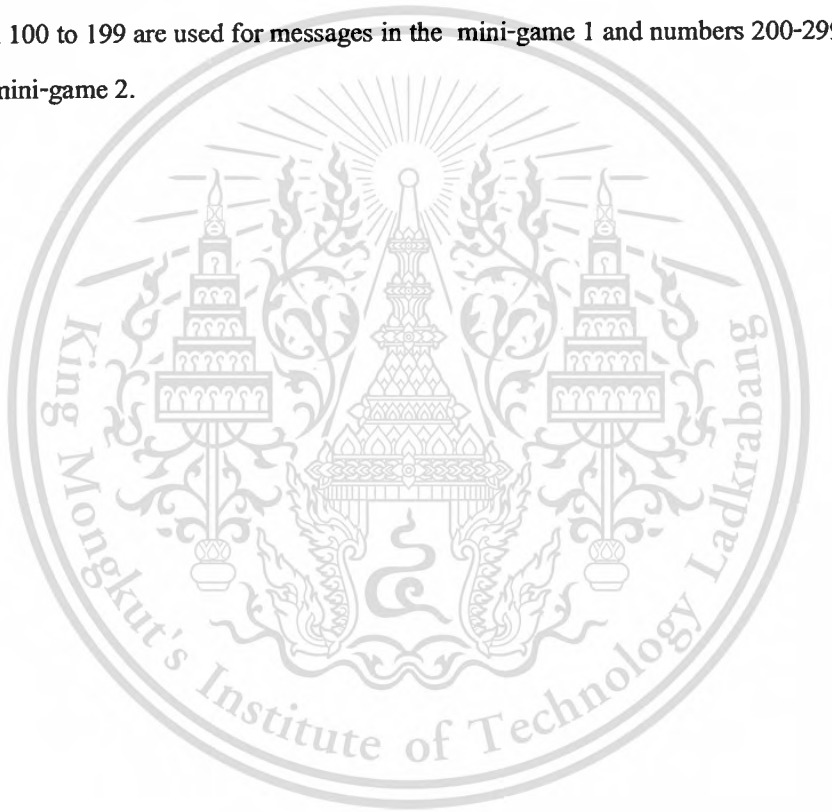
When the data is received, the data will be first kept as the *CFData*, which is a mutable type of data object wrapper for byte buffer. Then, the buffer of type *UINT8* will be created according to the length of data which will be converted to an integer and used to trigger the event or mini-game in our application as explained next.

```
NSString *buffer = [[NSStringalloc] initWithData:(NSData *)data encoding:NSUTF8StringEncoding];
intintBuff = [buffer intValue];
appDelegate.dataa= buffer;
appDelegate.addr = address;
   appDelegate.sss= s;
if (intBuff>0 &&intBuff<= 29) {
      appDelegate.p1AccMove = intBuff;
appDelegate.activePlayer = 1;
      [waiting alert];
   }
if (intBuff>= 50 &&intBuff< 99) {
inttempcal = intBuff - 50;
      appDelegate.p1AccMove = tempcal;
   }
if (intBuff>= 100) {
if (intBuff == 100) {
         appDelegate.miniGame1Flag = 1;
         [waitinggamealert];}
if (intBuff>= 100 &&intBuff< 199) {
         appDelegate.minigame1Data = intBuff - 100;
appDelegate.minigameServerGetDataFlag =1;}
   }
```

Figure 3.20 Data logic checking

When there is data coming from the *receivedData*, the data will be put in *NSData* object which will be changed to string object type before it can be used. With our programming algorithm, we design that the data that we will receive will be only numbers which we will use to generate the required data for the game.

We use 0 to 29for the position of the player. So when the data is in the range of 0 to 29, our application will update the position of the player from the client side. Then, the program sets the flag to the server turn. Numbers from 50 to 99 are used by the client informing the server for his latest position so that the server can use them to calculate the client's new position after the mini-game ends each time. Number 100, 200 and 300 are used to trigger the players for a mini-game's play (set the flag to the mini-game play). Numbers from 100 to 199 are used for messages in the mini-game 1 and numbers 200-299 are used for messages in the mini-game 2.



```
appDelegate = (AppDelegate*)[[UIApplicationsharedApplication] delegate];
move = arc4random() % 6+1;
appDelegate.p1AccMove = move;
NSString *strMoveValue = [NSStringstringWithFormat:@"%d",move];
   [gotMovesetText:strMoveValue];



-(void) sendDataToServer:( NSString  *)sender{
NSString *configStr = [sender stringByAppendingString:@"\n"];
   UInt8 *message = (UInt8 *)[configStr UTF8String];
CFDataRef data = CFDataCreate(NULL, message, sizeof(message));
CFSocketSendData(appDelegate.TempSock, NULL, data, 0);
CFRelease(data);
}

-(void) sendDataToClient:( NSString  *)sender{
NSString *configStr = [configStrstringByAppendingString:@"\n"];
   UInt8 *message = (UInt8 *)[sender UTF8String];
CFDataRef data = CFDataCreate(NULL, message, sizeof(message));
CFSocketSendData(appDelegate.sss, appDelegate.addr, data, 0);
CFRelease(data);
}
```

Figure 3.21 random and sending data

In an example, when the client moves after finishing his turn, the game will send the number of his current position to the server using *sendDataToServer* or *sendDataToClient* which will convert the data to string then to *UINT8* before sending.

For *sendDataToServer*, the data will be converted to a data object with the data message and the size of message. Then it will be sent according to the socket reference that has been created during the *acceptionConnection.*

For*sendDataToClient,* it is similar to *sendDataToServer* but needs extra information which an address to send the data to.

3.5.2 Client Socket

3.5.2.1 Client Socket overview [15]

1. Creates a *CFSocket* with a callback to notify the method *receiveData* when data is received.

2. Connects to the server via the socket.

3. Creates a *CFRunLoop* to wait for receive data on the socket.

4. Starts the *CFRunLoop*, the client is waiting for socket data.

5. When data is received from the server, *receiveData* will be called while the current view is in the *WaitingTurn* of the client to avoid the repeating of allocating memory and cause the data to lose.

```
    s = CFSocketCreate(NULL, PF_INET,
                       SOCK_STREAM, IPPROTO_TCP,
kCFSocketDataCallBack,
receiveDataCilent,
                       NULL);
structsockaddr_in     sin;
structhostent         *host;
memset(&sin, 0, sizeof(sin));
host = gethostbyname("192.168.1.114");
memcpy(&(sin.sin_addr), host->h_addr,host->h_length);
sin.sin_family = AF_INET;
sin.sin_port = htons(6666);
CFDataRef address;
CFRunLoopSourceRef source;
address = CFDataCreate(NULL, (UInt8 *)&sin, sizeof(sin));
CFSocketConnectToAddress(s, address, 0);
CFRelease(address);
source = CFSocketCreateRunLoopSource(NULL, s, 0);
CFRunLoopAddSource(CFRunLoopGetCurrent(),
source,
kCFRunLoopDefaultMode);
CFRelease(source);
     [selfsendDataToServer:@"-1"];
CFRunLoopRun();
```

Figure 3.22 *Join game Button*

3.5.2.2 Client Socket detail

For the client connection, the actual connection of the client side will be called in the main game board so that the iOS server can connect to the android device to avoid some bugs. This is slightly different from the process on Android client. To make the application compatible with the view changing of the iOS device, the player must press the Next button on the *mainboard* view. Pressing this button will create the socket reference used to connect to the host using the default data of the device like IP address, connection type, IP family and port. The client side then is able to connect to the server and run the *receiveDataClient* to wait for incoming data.

```
voidreceiveDataCilent(CFSocketRef s, CFSocketCallBackType type, CFDataRef address, const void *data,
void *info)
{
NSString *buffer = [[NSStringalloc] initWithData:(NSData *)data encoding:NSUTF8StringEncoding];
WaitingTurnClient *waitingc = [[WaitingTurnClientalloc] initWithNibName:nilbundle:nil];
  [waitingcsetData:buffersetSockRef:s];
}
```

Figure 3.23 *receiveDataCilent* method

The *receiveDataCilent* method is used to get data from the server and transform the received data into the integer type for later use. This is similar to the server's *receiveData*.

```
if (intBuff>= 1000 &&intBuff< 1999) {
        appDelegate.minigame1Data = intBuff - 1000;
appDelegate.minigameServerGetDataFlag =1;}
```

Figure 3.24 Data logic checking client side

Data logic checking on the client side is also similar to the server side. However, the client side uses the data range from 1000 to 1999 for the client token position in case of the client winning the mini-game. In summary, if the client receives data in range from 0 to 29, this means that the server wins the mini-game. The client's program then will update the server token. If the received data ranges from 1000 – 1999, this means that the client wins and the program will update the client token.

3.5.3 Mini game

3.5.3.1 Evasive game

```
-(IBAction)leftButton:(id)sender{
leftButton.enabled = NO;
rightButton.enabled = YES;
appDelegate.counter +=1;
}

-(IBAction)rightButton:(id)sender{
leftButton.enabled = YES;
rightButton.enabled = NO;
}
```

Figure 3.25 Evasive mini-game

The concept of this game is to tap the button on the left sequentially one by one. The score counter will be on the left button. Each time the left button is pressed the score will be increased by 1. In Xcode programming, the button can be linked with the outlet feature and can set property. With the outlet feature, Our program enables and disables other button(s) when one button has been pressed. The time counter will be counted down from 6 to 0with the interval of 1.0 second. Hence, the player needs to press the buttons as many as possible before the time runs out.

3.5.3.2 Energy Charger mini-game

```
-(IBAction)button:(id)sender{
appDelegate.counter2 +=1;
}
```

Figure 3.26 Energy Charger mini-game

In Energy Charger mini-game, the player needs to tap only one button as fast as he can in 6 seconds. Each time the button being pressed, it will call the code in **Figure 3.26** and start counting the score.

3.5.3.3Mini-game Timer

```
- (void)viewDidLoad

{

appDelegate = (AppDelegate*)[[UIApplicationsharedApplication] delegate];

count = 6;

appDelegate.counter = 0;

timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(counter)
userInfo:nilrepeats:YES];

rightButton.enabled = NO;

   [superviewDidLoad];}

-(void)counter{

count -= 1;

second.text = [NSStringstringWithFormat:@"%i",count];

if (count <= 0) {

      [timer invalidate];

timer = nil;}
```

Figure 3.27 Mini-game Timer code

When the mini-game view is started, the *viewDidLoad* method will be called which will set the counter (*count*) to 6 and reset the score counter(*appDelegate.counter*) to 0. Then, the timer object will be created and initialized by setting the count time interval to 1.0 second in which every one second, the program will tell *selector* to call method *counter* until *count* is equal to zero. After that the timer will be stopped and the timer object will be released. Finally, the view will be changed.

3.5.3.4Mini-game data sending

```
if (count <= 0) {
    [timer invalidate];
timer = nil;
if(appDelegate.turn == 1){
intsendResult = appDelegate.counter2 + 200;
NSString *resultStr = [NSStringstringWithFormat:@"%d", sendResult];
intcurrentPos = appDelegate.p1AccMove + 50;
NSString *resultStr2 = [NSStringstringWithFormat:@"%d", currentPos];
    [self sendDataToServer:resultStr2];
    [self sendDataToServer:resultStr]; // then change view}}
```

Figure 3.28 Mini-game data sending code

The game will send the current position of the client token added up with 50 before sending this value to the server. For example, if the current position is 17, the data will be 67 (17+50). After that, the game will send the score added up with 200 before sending it. For example, if the current score is 37, the game will add the score with 200 and send 237(37 + 200).

3.5.4 Change view

```
gameboard *mainboard = [[gameboardalloc] initWithNibName:nilbundle:nil];

[selfpresentModalViewController:mainboardanimated:YES];
```

Figure 3.29 View changing code

The Xcode programming has implemented the concept of view to help developer in programming iOS application. To change the view in the Xcode, first programmers need to allocate the memory for that view object with or without the initial value. Then, the object of that view will be changed via the animated changing method. To allocate memory in Xcode, programmers must be very careful not to allocate the same object more than one time; otherwise the pointer reference will be lost.

3.5.5Global Variable

```
@interface AppDelegate :UIResponder<UIApplicationDelegate>{

CFSocketRefsockFromClient;

CFDataRefaddr;

NSString *dataa;

NSString *dataFromClient;

int p1AccMove;

int p2AccMove;}


@property (assign) CFSocketRefsockFromClient;

@property (assign) CFDataRefaddr;

@property (retain, nonatomic) NSString *dataa;

@property (retain, nonatomic) NSString *dataFromClient;

@property (assign) int p1AccMove;

@property (assign) int p2AccMove;
```

Figure 3.30 Class.h@interfacecode

The concept of global variable sin Xcode programming is different from other programming languages. It must be declared on the *appdelegate* class as a normal variable. However, to use it globally, it must be shared first. In the example in **Figure 3.30**, we declare some variables in *AppDelegate.h* like normal variables and we declare the property of the variables such as *retain*,*strong*, *weak*, *nonatomic*, *assign*, etc.

```
#import "AppDelegate.h"

#import "ViewController.h"


@implementation AppDelegate


@synthesize window = _window;

@synthesize viewController = _viewController;

@synthesize addr;

@synthesize dataa;

@synthesize dataFromClient;

@synthesize sockFromClient;

@synthesize p1AccMove;

@synthesize p2AccMove;
```

Figure 3.31 Class.m@implementation code

Then, Xcodecreates*getter and setter*methodusing*synthesize* as shown in **Figure 3.31**

```
// gameboard.h

#import <UIKit/UIKit.h>

#import "AppDelegate.h"

@interface gameboard :UIViewController{

AppDelegate *appDelegate;}


// gameboard.m


appDelegate = (AppDelegate*)[[UIApplicationsharedApplication] delegate];

appDelegate.hasConnect = 1;

appDelegate.androidFlag =0;
```

Figure 3.32 Class.h and class.m

In **Figure3.32**, to use the global variables, first the object AppDelegatemust be created which can be created either in .h file or in .m file. [[UIApplicationsharedApplication] delegate] will create the pointer to the AppDelegateobject so that it can be accessible by shared objects. Because the appdelegate is a helper object as mentioned in Chapter 3, it will be automatically created when the application is started. Most of the variables in appdelegate will be kept in the appdelegate thread.

3.5.6 Game Board

```
- (void)viewDidLoad

{appDelegate = (AppDelegate*)[[UIApplicationsharedApplication] delegate];

   appDelegate.miniGame1Flag = -1;

   P1XArraypos = [[NSArrayalloc]
initWithObjects:@"48",@"48",@"48",@"48",@"48",@"48",@"48",@"48",@"48",@"142",@"234",@"234",
@"234",@"234",@"234",@"234",@"234",@"234",@"234",@"326",@"420",@"420",@"420",@"420",@"42
0",@"420",@"420",@"420",@"420",@"508",nil];

   P1YArraypos = [[NSArrayalloc]
initWithObjects:@"44.5",@"128.5",@"209.5",@"291.5",@"373.5",@"456.5",@"538.5",@"620.5",@"703.5"
,@"703.5",@"703.5",@"620.5",@"538.5",@"456.5",@"373.5",@"291.5",@"209.5",@"128.5",@"44.5",@"4
4.5",@"44.5",@"128.5",@"209.5",@"291.5",@"373.5",@"456.5",@"538.5",@"620.5",@"703.5",@"419.5",
nil];

   P2XArraypos = [[NSArrayalloc]
initWithObjects:@"976",@"976",@"976",@"976",@"976",@"976",@"976",@"976",@"976",@"883",@"79
2",@"792",@"792",@"792",@"792",@"792",@"792",@"792",@"792",@"699",@"605",@"605",@"605",@
"605",@"605",@"605",@"605",@"605",@"605",@"508",nil];

   P2YArraypos = [[NSArrayalloc]
initWithObjects:@"703.5",@"620.5",@"538.5",@"456.5",@"373.5",@"291.5",@"209.5",@"128.5",@"44.5"
,@"44.5",@"44.5",@"128.5",@"209.5",@"291.5",@"373.5",@"456.5",@"538.5",@"620.5",@"703.5",@"70
3.5",@"703.5",@"620.5",@"538.5",@"456.5",@"373.5",@"291.5",@"209.5",@"128.5",@"44.5",@"283.5",
nil];

   p1Int = appDelegate.p1AccMove;

   p2Int = appDelegate.p2AccMove;

if (p1Int > 29) {

      p1Int = 29; }

if (p1Int < 0) {

      p1Int = 0;}

if (p2Int > 29) {

    p2Int = 29;}
```

Figure 3.33 *gameboard* code

In the game board's codes in **Figure 3.33,** first we prepare the array of the token positions which stores positions of the token when it is moved to each space on the board. Since, there are 30 spaces in the board, the array index values are from 0 to 29. This value must be checked that it is within the range.

3.5.7 Token positipon update

```
-(void) p1drawPosition:(int)number;{
NSLog(@"Number in p1draw Position is %d",number);
xPos = [[P1XArraypos objectAtIndex:number]doubleValue];
yPos = [[P1YArraypos objectAtIndex:number]doubleValue];
NSLog(@"xPos in p1draw %f",xPos);
NSLog(@"yPos in p1draw %f",yPos);
image.center= CGPointMake(xPos,yPos);}
```

Figure 3.34 Update position code

This code will update the position of the token by call the index of the array by the received data then the array data will be read position X and position Y and draw the image according to the data.

3.6 Android programming

3.6.1 Game Framework

Framework is used to help developers to develop game easier and more manageable. Framework is separated into many modules:

File I/O: this module will handle reading and writing resources(files) in the *asset* folder.

Input: This module will handle user's input such as keystrokes, touch event, accelerometer as shown in **Figure 3.37**

Graphics: this module deals with loading and drawing images on the screen **Figure 3.36**

Game Framework: this module will combine all modules together for game developer to write a game application more easily.

3.6.1.1 Examples of game framework

```
Public abstract class AndroidGame extends Activity implements Game {
      Public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            requestWindowFeature(Window.FEATURE_NO_TITLE);
            getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN);
            booleanisLandscape =
            getResources().getConfiguration().orientation ==
            Configuration.ORIENTATION_LANDSCAPE;
            intframeBufferWidth = isLandscape ? 1280 : 800;
            intframeBufferHeight = isLandscape ? 800 : 1280;
      Bitmap frameBuffer = Bitmap.createBitmap(frameBufferWidth,
            frameBufferHeight, Config.RGB_565);
      Public void onResume() {
            super.onResume();
            wakeLock.acquire();
            screen.resume();
            renderView.resume();}
      Public void onPause() {
            super.onPause();
            wakeLock.release();
            renderView.pause();
            screen.pause();
            if (isFinishing())
                  screen.dispose();}
      Public Input getInput() {
            return input; }
      Public void setScreen(Screen screen) {
            if (screen == null)
            throw new IllegalArgumentException("Screen must not be null");
            this.screen.pause();
            this.screen.dispose();
            screen.resume();
            screen.update(0);
            this.screen = screen;
      }
```

Figure 3.35 *AndroidGame* class

AndroidGame class is a class to control everything on the screen in the application such as set width, set height and change setting in the screen. *setScreen* method is used for changing the current screen to other screen of games.

```
public class AndroidGraphics implements Graphics {
      publicPixmapnewPixmap(String fileName, PixmapFormat format) {
            Configconfig = null;
            if (format == PixmapFormat.RGB565)
                  config = Config.RGB_565;
            else if (format == PixmapFormat.ARGB4444)
                  config = Config.ARGB_4444;
            else
                  config = Config.ARGB_8888;
      Options options = new Options();
            options.inPreferredConfig = config;
            InputStream in = null;
      Bitmap bitmap = null;
            try{
                  in = assets.open(fileName);
                  bitmap = BitmapFactory.decodeStream(in);
            if (bitmap == null)
                  throw new RuntimeException("Couldn't load bitmap from
                  asset '"+ fileName + "'");
      } catch (IOException e) {
                  throw new RuntimeException("Couldn't load bitmap from
                  asset '"+ fileName + "'");
      } finally {
            if (in != null){
                  try{in.close();}catch(IOException e){}}}
            if (bitmap.getConfig() == Config.RGB_565)
                  format = PixmapFormat.RGB565;
            else if (bitmap.getConfig() == Config.ARGB_4444)
                  format = PixmapFormat.ARGB4444;
            else
                  format = PixmapFormat.ARGB8888;
            return new AndroidPixmap(bitmap, format);}
      public void drawPixmap(Pixmappixmap, int x, int y) {
            canvas.drawBitmap(((AndroidPixmap)pixmap).bitmap, x, y, null); }
      public void drawPixmap(Pixmappixmap, int x, int y, intsrcX, intsrcY,
            intsrcWidth, intsrcHeight) {
            srcRect.left = srcX;
            srcRect.top = srcY;
            srcRect.right = srcX + srcWidth - 1;
            srcRect.bottom = srcY + srcHeight - 1;
            dstRect.left = x;
            dstRect.top = y;
            dstRect.right = x + srcWidth - 1;
            dstRect.bottom = y + srcHeight - 1;
            canvas.drawBitmap(((AndroidPixmap) pixmap).bitmap, srcRect,
            dstRect,
            null);
      }
```

Figure 3.36 *AndroidGraphics* class

This class *AndroidGraphics* is used to handle pixmap pictures such as loading picture from *asset*, drawing picture and drawing a pixel. *drawPixmap* is used for drawing picture on screen.

```
PublicclassAndroidFileIOimplementsFileIO {
        AssetManager assets;
        String externalStoragePath;

        PublicAndroidFileIO(AssetManager assets) {
                this.assets = assets;
                this.externalStoragePath =
                Environment.getExternalStorageDirectory()
                   .getAbsolutePath() + File.separator;
        }
        PublicInputStreamreadAsset(String fileName) throwsIOException {
        Returnassets.open(fileName);
        }


        PublicInputStreamreadFile(String fileName) throwsIOException {
        ReturnnewFileInputStream(externalStoragePath + fileName);
        }


        PublicOutputStreamwriteFile(String fileName) throwsIOException {
        ReturnnewFileOutputStream(externalStoragePath + fileName);
        }
}
```

Figure 3.37 *AndroidFileIO* class

AndroidFileIO class is used for handling input and output files. *readAsset* is used for reading files in *asset* folder. *readFile* is used for reading files from the selected path. *writeFile* is use for writing files to selected path.

3.6.2 Socket programming

3.6.2.1 Server side

```
        PublicstaticfinalintSERVERPORT = 6666;
        ServerSocketserverSocket;
        Socket client;
        Publicvoidrun() {
                if (SERVERIP != null){
                        handLer.post(new Runnable() {publicvoid run(){}});
                        serverSocket = newServerSocket(SERVERPORT);
                        connected = true;
        while (connected) {
                if(client == null){
                        client = serverSocket.accept();}
                handLer.post(new Runnable() {
        publicvoidrun(){}});
                BufferedReaderHin
                =newBufferedReader(newInputStreamReader(client.getInputStream()));
                PrintWriterHout
                =newPrintWriter(newBufferedWriter(newOutputStreamWriter(client.getOutputStr
                eam())), true);
String line = null;
                Inti=1;
                if(Settings.startgame==0&&i==1){
                        line = Hin.readLine();
                        if(line != null){
                        check= Integer.parseInt(line);
                        line = null;}
                if(check == -1){
                        Settings.setstartgame(1);
                        Settings.store();
                        i=0;
                        Hin.close();
                        serverSocket.close();
                        client.close();
                        connected = false;
                }}}}
```

Figure 3.38 *starthost* class

This thread is used when the android tablet acts as the server. *serverSocket* class is used to create the server socket, and *Socket* class is used for creating the client's socket. To receive data from the client, the server uses *InputStreamReader* class which converts byte stream into character stream by using *getInputStream* method. *In* is a variable in the *BufferedReader* class used for keeping data in the buffer which the server later can read by using the *readLine* method .*Settings.startgame* is the flag to control the server while waiting data from the client. If data received from the client is -1, the server will start the game. To send data to the client, the server will use *PrintWriter* class which writes data by using *println* method. *BufferedWriter* class keeps data in the buffer and writes to socket using *OutputStreamWriter* class to convert character stream into byte stream before sending data.

```
if(Settings.receivedatap1==0){
        line = Hin.readLine();
if(line != null){
        check= Integer.parseInt(line);
        line = null;
}
if(check >=0 && check <= 29){
        Settings.setpositionp1(check);
        Settings.setendp2(0);
        Settings.setstartplayer2(1);
        Settings.setupdatep1(1);
        Settings.setreceivedatap1(1);
}
if(check>=50 &&check<=99){
        intdatain=0;
        datain=check-50;
        Settings.setpositionp1(datain);
        Settings.setupdatep1(1);
        Settings.store();
}
if(check == 100){
        Settings.setstartminigame02(1);
        Settings.store();
}
if(check == 200){
        Settings.setstartminigame01(1);
        Settings.store();
}
```

Figure 3.39 *threadhost* class

Threadhost class in **Figure 3.39** contains codes for sending and receiving data from the client while playing the game. *Settings.receivedatap1* is the flag which controls the server state whether it will receive data from the client or not. If the value is 0, the server will wait to receive data; otherwise, the server will not wait for data but do other tasks such as sending data instead. When the server received data, it checks the value of data for which condition the game state should be performed. If the value of the data is between 0 and 29, the server will update the position of the client token; then the server will set its state to playing state. If the value of the data is between 50 and 99, the server will only use the data to update the position of the client token. If the data is between 100 or 200, the server will start playing mini-game.

```
if(check >100 &&check < 200){
        dataout= check-101;
        Settings.setdatagame02p1(dataout);
}
if(Settings.datagame02p2>Settings.datagame02p1){
        dataoutp2 +=Settings.setpositionp2;
        Settings.setpositionp2(dataoutp2);
        Settings.setupdatep2(1);
        Settings.setreceivedatap1(1);
        Settings.setsenddatap2(1);
}
if(Settings.datagame02p1>= Settings.datagame02p2){
        dataoutp1 +=Settings.setpositionp1;
        Settings.setpositionp1(dataoutp1);
        Settings.setupdatep1(1);
        Settings.setreceivedatap1(1);
        Settings.setresultp2(1);
}
if(check>200&&check<300){
        dataout2= check-201;
        Settings.setdatagame01p1(dataout2);
        if(Settings.datagame01p2> Settings.datagame01p1){
                datawin+=Settings.setpositionp2;
                Settings.setpositionp2(datawin);
                Settings.setupdatep2(1);
                Settings.setreceivedatap1(1);
                Settings.setsenddatap2(1);
        }
        if(Settings.datagame01p1>= Settings.datagame01p2){
                datalose+=Settings.setpositionp1;
                Settings.setpositionp1(datalose);
                Settings.setupdatep1(1);
                Settings.setreceivedatap1(1);
                Settings.setresultp2(1);
        }
}
```

Figure 3.40 *threadhost* class

If the data is between 101 and 199 or between 201 and 299, the server will use this received data to calculate the mini-game result. 101 to 199 are for minigame1(minigame01) and 200 to 299 are for minigame2 (*minigame02*).In the mini games, the player that gets the highest score will be the winner which will get bonus position reward.

```
if(Settings.senddatap2==1) {
	Hout.println(Settings.setpositionp2);
	if(Settings.minigame01==0&&Settings.minigame02==0&&Settings.minigame03=0){
		Settings.setstartplayer2(0);
		Settings.setreceivedatap1(0);
		Settings.setsenddatap2(0);
		Settings.setplayerturn(0);
	}
	if(Settings.minigame01==1||Settings.minigame02==1||Settings.minigame03==1){
		if(Settings.playerturn==0) {
			Settings.setendp2(0);
			Settings.setreceivedatap1(1);
			Settings.setstartplayer2(1);
			Settings.setplayerturn(1);
		}else{
			Settings.setplayerturn(0);
			Settings.setstartplayer2(0);
			Settings.setreceivedatap1(0);
		}
		Settings.isminigame01(0);
		Settings.isminigame02(0);
		Settings.setdatagame01p1(0);
		Settings.setdatagame01p2(0);
		Settings.setdatagame02p1(0);
		Settings.setdatagame02p2(0);
		Settings.setsenddatap2(0);}
if(Settings.sendresultp2 ==1){
	dataoutp1=1000;
	if(Settings.minigame02==1 ||Settings.minigame01==1){
		dataoutp1+= Settings.setpositionp1;
	}
		Hout.println(dataoutp1);
	}
if(Settings.sendminigamep2 == 2){
		int test=50;
		test += Settings.setpositionp2;
		Hout.println(test);
		Settings.setsendminigamep2(1);
	}
if(Settings.sendminigamep2 == 1){
	if(Settings.minigame02==1){
		test =100;
	}
	if(Settings.minigame01==1){
		test =200
	}
	Hout.println(test);
}
```

Figure 3.41 *threadhost* class (Cont.)

For sending data to the client side, the serverusesSettings.senddatap2, Settings.sendresultp2 and Settings.sendminigamep2. Settings.senddatap2 is used for sending the server token's position to the client side in which there are two cases: after the server finishes playing his event and after the server wins the mini-game. Settings.sendminigamep2 is used for sending data to the client telling it to start playing the mini-game together with server side. Settings.sendresultp2 is used for sending the result (the server loses) of the mini game to the client.

3.6.2.2 Client side

```
      String serverIpAddress = ""
      boolean connected = false;
      static Handler handler = new Handler();
      Socket socket;
Publicvoidrun() {
      InetAddressserverAddr = InetAddress.getByName(serverIpAddress);
      try{
      socket = new Socket(serverAddr, SERVERPORT);
      }catch (IOException e){
      connected = true;}
      connected = true;
      while (connected) {
            PrintWriterJout
            =newPrintWriter(newBufferedWriter(newOutputStreamWriter(socket.getOutp
            utStream())), true);
            BufferedReaderJin =
            newBufferedReader(newInputStreamReader(socket.getInputStream()));
            String Jline=null;
            Inti=1;
            if (Settings.startgame == 0 &&i==1){
                  Jout.println(-1);
                  Settings.setstartgame(1);
                  Settings.store();
                  Settings.store();
                  i=0;
                  Jout.close();
                  socket.close();
                  connected =false;
            }}}}
```

Figure 3.42 *startjoin* class

To start joining the game, the program calls *startjoin* Class. The client uses *Socket* class for to connect to the server. Sending the data to the server can be done using the println method in the*PrintWriter* class. *Jin* is a variable in the *BufferedWriter* class used for keeping data in the buffer which the server later uses the *OutputStreamWriter* class to convert character stream to byte stream and sends this data to server side. *InputStreamReader* class is used to receive data from the server by using *getInputStream* method to get byte data into character data. *BufferedReader* keeps received data in the buffer for the client and can be read by using *readLine* method. If this thread is called, the client will sends-1 flag to indicates the server for its readiness to start the game. *Settings.startgame* is the flag to control the client when it should send the starting game flag value to the server.

```
If(Settings.sendminigamep1 == 1){
            int test=0;
      if(Settings.minigame02==1){
            test =100;
            Jout.println(test);
      }
      if(Settings.minigame01==1){
            test =200;
            Jout.println(test);
      }
      Settings.setsendminigamep1(0);
      Settings.store();
}
If(Settings.sendresultp1 ==2){
      intdatain;
      if(Settings.minigame02==1){
            datain =101;
            datain +=Settings.datagame02p1;
            Jout.println(datain);
      }
      If(Settings.minigame01==1){
            datain =201;
            datain +=Settings.datagame01p1;
            Jout.println(datain);
      }
      If(Settings.playerturn==1){
            Settings.setreceivedatap2(0);
      }
      Settings.setresultp1(0);
      Settings.store();
}
If(Settings.senddatap1 == 1  ){
      Jout.println(Settings.setpositionp1);
      Settings.setstartplayer1(0);
      Settings.setsenddatap1(0);
      Settings.setreceivedatap2(0);
      Settings.store();
}
```

Figure 3.43 *threadjoin* class

The client uses*Settings.sendminigamep1*, *Settings.sendresultp1*, and*Settings.senddatap1* to control which data will be sent to the server. *Settings.sendminigamep1* is for sending a flag for client to start the mini-game.*Settings.sendresultp1*is for sending the result of the mini-games(either minigame1 or minigame2). *Settings.senddatap1* is for sending the client token's position to the server side.

```
if(Settings.receivedatap2==0){
	Jline = Jin.readLine();
	if(Jline != null){
		check= Integer.parseInt(Jline);
		Jline=null;
	}
	if(check >=0 && check <= 29){
		Settings.setpositionp2(check);
		Settings.setendp1(0);
		Settings.setstartplayer1(1);
		Settings.setupdatep2(1);
		Settings.setreceivedatap2(1);}
	if(check>=50 &&check<=99){
		intdatain=0;
		datain=check-50;
		Settings.setpositionp2(datain);
		Settings.setupdatep2(1);
	}
	if(check == 100){
		Settings.setstartminigame02(1);}
	if(check == 200) {
		Settings.setstartminigame01(1);}
	if(check >= 1001&&check <= 1050){\
		int data;
		data = check-1001;
		if(Settings.minigame02==1||Settings.minigame01==1){
			Settings.setpositionp1(data);
			Settings.setupdatep1(1); }
			if(Settings.playerturn==1){
				Settings.setendp1(1);
				Settings.setreceivedatap2(1);
			}
		Settings.setdatagame01p1(0);
		Settings.setdatagame01p2(0);
		Settings.setdatagame02p1(0);
		Settings.setdatagame02p2(0);
		Settings.isminigame01(0);
		Settings.isminigame02(0);
	}
```

Figure 3.44 *threadjoin* class (Cont.)

To receive data from the server, the client uses*Settings.receivedatap2* class. If the received data is either 0 to 29 or 50 to 99, the received value will be used to update the token's position. If the value is 100, the minigame1 will start. If the value is 200, the minigame2 will start. If the value is from 1001to 1050, the client will wait for the mini-game result from the server.

3.6.3 Setting parameters for the game

3.6.3.1 Loading the game data

```
Public class Settings {
      Public final static String file = "info.txt";
      Public static voidload(FileIO files) {
      BufferedReaderin =
      newBufferedReader(newInputStreamReader(files.readFile(file)));setp
      ositionp1 = Integer.parseInt(in.readLine());
      setpositionp2 = Integer.parseInt(in.readLine());
      startplayer1 = Integer.parseInt(in.readLine());
      startplayer2 = Integer.parseInt(in.readLine());
      senddatap1 = Integer.parseInt(in.readLine());
      senddatap2 = Integer.parseInt(in.readLine());
      player1 = Integer.parseInt(in.readLine());
      player2 = Integer.parseInt(in.readLine());
      updatep1 = Integer.parseInt(in.readLine());
      updatep2 = Integer.parseInt(in.readLine());
      receivedatap1 =Integer.parseInt(in.readLine());
      receivedatap2 =Integer.parseInt(in.readLine());
      datagame= Integer.parseInt(in.readLine());
      playerturn = Integer.parseInt(in.readLine());
      }
```

Figure 3.45 *Settings*: *load* method

Info.txt is a text file which the game saves all global variables so that the data will not be lost when the screen is changed. Load method is used to load global variables from *info.txt* file. Examples of global Variables are *setpositionp, startplayer, senddatap, receivedatap, player, datagame* and *playerturn. setpositionp* is for storing the position of the player token. *startplayer* is used for controlling to start the player turn. *Senddatap* and *receivedatap* are used for controlling flow of sending and receiving data in the game. *Player* is used by system to control which player will be player number 1 or player number 2.*datagame* is used for storing mini-game's data for calculation. *playerturn* is used by the system for the player turn. 1 means it is the player turn and 0 means it is not the player turn.

3.6.3.2 Saving the game data

```
Publicstaticvoidsave(FileIO files) {
        BufferedWriterout =
        newBufferedWriter(newOutputStreamWriter(files.writeFile(file)));
        out.write(Integer.toString(setpositionp1));
        out.write("\n");
        out.write(Integer.toString(setpositionp2));
        out.write("\n");
        out.write(Integer.toString(startplayer1));
        out.write("\n");
        out.write(Integer.toString(startplayer2));
        out.write("\n");
        out.write(Integer.toString(senddatap1));
        out.write("\n");
        out.write(Integer.toString(senddatap2));
        out.write("\n");
        out.write(Integer.toString(player1));
        out.write("\n");
        out.write(Integer.toString(player2));
        out.write("\n");
        out.write(Integer.toString(updatep1));
        out.write("\n");
        out.write(Integer.toString(updatep2));
        out.write("\n");
        out.write(Integer.toString(receivedatap1));
        out.write("\n");
        out.write(Integer.toString(receivedatap2));
        out.write("\n");
        out.write(Integer.toString(playerturn));
        out.write("\n");
        out.write(Integer.toString(datagame));
        out.write("\n");
}
```

Figure 3.46 *Settings*: *save* method

Save method is used for saving all global variables in the application to *info.txt* file. Before changing to the new screen, all global variables will be saved.

3.6.4 Function in game

3.6.4.1 Input

```
Private Boolean inBounds(TouchEvent event, int x, int y, int width, int
height){
If (event.x> x &&event.x< x + width - 1 &&
       event.y> y &&event.y< y + height - 1)
       returntrue;
else
       returnfalse;
}
```

Figure 3.47 *inbounds* method

Inbound method is used by many screens for creating an area of the touch event so that when the player touches the screen, his touching position must be within this bound for the event to be execute.

3.6.4.2 Timer

```
Publicvoid present (floatdeltaTime) {

      Currenttime = System.nanoTime();
      Currenttime -= Starttime;
      Currenttime /=100000000;
}
Publicvoid resume () {

      Starttime = System.nanoTime();
}
```

Figure 3.48 Timer

Timer method is used for controlling time in the mini-games. *Starttime* is the time that mini-game is started. *System.nanoTime* is a method to get the current time from the system.

3.6.5 Assets

```
//Example of code
Publicclass Assets {
	PublicstaticPixmapbsh1;
	PublicstaticPixmappsh1;
	PublicstaticPixmapTitle;
	PublicstaticPixmapsetup;
	PublicstaticPixmaptapstart;
	PublicstaticPixmapHOSTBUTTON;
	PublicstaticPixmapJOINBUTTON;
	PublicstaticPixmapmovef;
	PublicstaticPixmapmoveb;
	PublicstaticPixmapran;
	PublicstaticPixmapranb;
	PublicstaticPixmapranf;
	PublicstaticPixmaplpress;
	PublicstaticPixmaplrelease;
	PublicstaticPixmaprpress;
	PublicstaticPixmaprrelease;
	Publicstatic Sound click;
}
PublicclassLoadingScreenextends Screen {
	Assets.bsh1 = g.newPixmap("bsh1.png", PixmapFormat.RGB565);
	Assets.psh1 = g.newPixmap("psh1.png", PixmapFormat.RGB565);
	Assets.Title = g.newPixmap("Title.jpg", PixmapFormat.ARGB4444);
	Assets.boardhigh = g.newPixmap("boardhigh.jpg", PixmapFormat.RGB565);
	Assets.setup = g.newPixmap("setup.jpg", PixmapFormat.RGB565);
	Assets.tapstart = g.newPixmap("tapstart.png", PixmapFormat.RGB565);
	Assets.HOSTBUTTON = g.newPixmap("HOSTBUTTON.png", PixmapFormat.RGB565);
	Assets.JOINBUTTON = g.newPixmap("JOINBUTTON.png", PixmapFormat.RGB565);
	Settings.load(game.getFileIO());
	Settings.setpositionp1(0);
	Settings.setpositionp2(0);
	Settings.setstartplayer1(0);
	Settings.setstartplayer2(0);
	Settings.setplayer1(0);
	Settings.setplayer2(0);
	Settings.setstartgame(0);
	Settings.save(game.getFileIO());
	game.setScreen(newMainMenuScreen(game));
}
```

Figure 3.49 *Assets* Class

Assets class is for representing all pictures and sound that will be used in game. *LoadScreen* will load all assets before starting the game and load all global variables from *Settings* class for setting data to be 0.

3.6.6 Game screen

3.6.6.1 Update

```
//Some example of code//
PublicclassGameScreenCextends Screen {
Publicvoidupdate(floatdeltaTime) {
	List<TouchEvent>touchEvents = game.getInput().getTouchEvents();
	game.getInput().getKeyEvents();
	if (Settings.updatep1 ==1){
		p1x=Settings.setpositionp1;
		Settings.setupdatep1(0);
	}
	if (Settings.updatep2 ==1){
		p2x=Settings.setpositionp2;
		Settings.setupdatep2(0);
	}
	if (flag ==1){
		this.event();
		this.waiting(1);}
	if(flag==2){
		this.waiting(2);
		Settings.setsenddatap2(1);
		game.setScreen(new Winner(game));}
	if(flag==3){
		this.waiting(1);
		game.setScreen(new Loser(game));}
	if(state == GameState.Ready){
		updateReady(touchEvents);}
	if(state == GameState.Running){
		updateRunning(touchEvents,deltaTime);}
	if(state == GameState.Paused){
		updatePaused(touchEvents);}
	if(state == GameState.GameOver){
		updateGameOver(touchEvents);}
	if(Settings.startminigame02==1){
		Settings.isminigame02(1);
		Settings.setstartminigame02(0);
		game.setScreen(new Minigame02(game));
	}
	if(Settings.startminigame01==1){
		Settings.isminigame01(1);
		Settings.setstartminigame01(0);
		game.setScreen(new Minigame01(game));
	}
```

Figure 3.50 *GameScreen*: *update* method

In *GameScreen* class handle different states of game. Each state has its own method. There are several methods such as *update*, *present*, *resume* and *dispose*. The *update* method is used for handling game flags

for performing different actions such as updating position of token in the board game, calling event, calling mini-game, changing game screen to the end phase, setting state of the game for client etc.

There are two *GameScreen* classes in our implementation: *GameScreenClient* and *GameScreenServer* for the client and the server respectively. These two classes are very similar. However, there are some differences. For example, when the mini-game ends the*GameScreenServer* will be in the waiting state for the updated data (score) from the client but the *GameScreenClient* will send the mini-game data (score) to the server.

3.6.6.2 States

```
Privatevoidupdateready(List<TouchEvent>touchEvents) {
      if(touchEvents.size() > 0){
            state = GameState.Running;}}
privatevoidupdateRunning(List<TouchEvent>touchEvents, floatdeltaTime) {
            intlen = touchEvents.size();
      for(inti = 0; i<len; i++) {
            TouchEvent event = touchEvents.get(i);
      if(Settings.startplayer1==1 || Settings.startplayer2 ==1){
      if(event.type == TouchEvent.TOUCH_DOWN){
      if(inBounds(event,320 ,200, 640,400 ) ){
            Assets.click.play(1);
            set01=1;
            next=0;
            state = GameState.Paused;
            return;}}}}
privatevoidupdatePaused(List<TouchEvent>touchEvents) {
      if(Settings.startplayer1==1 || Settings.startplayer2 ==1){
            intlen = touchEvents.size();
      for(inti = 0; i<len; i++) {
            TouchEvent event = touchEvents.get(i);
      if(event.type == TouchEvent.TOUCH_DOWN||event.type ==
      TouchEvent.TOUCH_UP){
      if(inBounds(event,575 ,650, 500,275 )&&next==0 ){
            set01=0;
            Assets.click.play(1);
      if(event.type == TouchEvent.TOUCH_UP){
            next=1;
            ran=random();
            oldp1x=p1x;
            p1x+=ran;
      }
      return;
      }}}
```

Figure 3.51 *GameScreen* states

*updateReady, updateRunning*and *updatePaused* methods are for handling states in the *GameScreen* class. *updateReady* is for the startup state. *updateRunning* is for handling the touch event ; while playing the game, if the player touches the screen on his turn, the game system will change the state of the game to the *Paused* state (which is the state that the game will get to the walk screen). When the player touches the walk button, the *updatePaused* will first generate a random number for the player's moving position and then change back to the running state (change screen to the game board screen).

3.6.6.3 Present

```
int positionpx1 [] =
{50,50,50,50,50,50,50,50,50,50+b,50+2*b,50+2*b,50+2*b,50+2*b,50+2*b,50+2*b,50+
2*b,50+2*b,50+2*b,50+2*b,50+3*b,50+4*b,50+4*b,50+4*b,50+4*b,50+4*b,50+4*b,50+4*b,50+4
*b,50+4*b,50+5*b};
int positionpy1 [] =
{15,15+a,15+2*a,15+3*a,15+4*a,15+5*a,15+6*a,15+7*a,15+8*a,15+8*a,15+8*a,15+7*a
,15+6*a,15+5*a,15+4*a,15+3*a,15+2*a,15+a,15,15,15,15+a,15+2*a,15+3*a,15+4*a,15
+5*a,15+6*a,15+7*a,15+8*a,15+4*a};
int positionpx2 []={1205,1205,1205,1205,1205,1205,1205,1205,1205,1205-b,1205-
2*b,1205-2*b,1205-2*b,1205-2*b,1205-2*b,1205-2*b,1205-2*b,1205-2*b,1205-
2*b,1205-3*b,1205-4*b,1205-4*b,1205-4*b,1205-4*b,1205-4*b,1205-4*b,1205-
4*b,1205-4*b,1205-4*b,1205-5*b};
int positionpy2[]=
{15+8*a,15+7*a,15+6*a,15+5*a,15+4*a,15+3*a,15+2*a,15+a,15,15,15,15+a,15+2*a,15
+3*a,15+4*a,15+5*a,15+6*a,15+7*a,15+8*a,15+8*a,15+8*a,15+7*a,15+6*a,15+5*a,15+
4*a,15+3*a,15+2*a,15+a,15,15+4*a};

Publicvoidpresent(floatdeltaTime) {
      Graphics g = game.getGraphics();
      g.drawPixmap(Assets.boardhigh, 0, 0);
if(p1x>=29){
      p1x=29;
      flag =3;
}
if(p2x>=29){
      p2x=29;
      flag = 2;
}
      g.drawPixmap(Assets.psh1,positionpx1[p1x],positionpy1[p1x]);
      g.drawPixmap(Assets.bsh1,positionpx2[p2x],positionpy2[p2x]);
if(state == GameState.Paused){
      drawPausedUI();}
}
```

Figure 3.52 *GameScreen*: *present* method

The *present* method in *GameScreen* class is for drawing and updating main screen board of the game and the position of the player's token. The *present* method checks the position of the player's token. If it is more than 29, the *present* method will set a flag to end the *game. Assets.boardhigh* is used for displaying the board game picture. *Assets.psh1* and *Assets.bsh1* are used for displaying the token of player 1 and player 2 respectively. *position1* and*position2*variablesare used to keep the positions of the player1 and player2 tokens in the board game.

3.6.6.4 walk screen

```
Privatevoiddrawpausedui(){
        Graphics g = game.getGraphics();
        if(next==0){
                g.drawPixmap(Assets.turn_phase, 0, 0);
                if(set01 == 1){
                        g.drawPixmap(Assets.walkbutton,g.getWidth()/2-175 , 650);
                }
                if(set01 ==0){
                g.drawPixmap(Assets.walkbuttonpush,g.getWidth()/2-175 , 650);
                }
        }
        if (next==1){
                g.drawPixmap(Assets.turn_phasegot, 0, 0);
                start=1;
                if (ran ==1){
                        g.drawPixmap(Assets.number, g.getWidth()/2-150, 690, 25,
 0, 22, 30);}
                if (ran ==2){
                        g.drawPixmap(Assets.number, g.getWidth()/2-150, 690, 40,
 0, 22, 30);}
                if (ran ==3){
                        g.drawPixmap(Assets.number, g.getWidth()/2-150, 690, 60,
 0, 22, 30);}
                if (ran ==4){
                        g.drawPixmap(Assets.number, g.getWidth()/2-150, 690, 80,
                0, 22, 30); }
                if (ran ==5){
                        g.drawPixmap(Assets.number, g.getWidth()/2-150, 690, 100,
                0, 22, 30);}
                if (ran ==6) {
                        g.drawPixmap(Assets.number, g.getWidth()/2-150, 690, 120,
                0, 22, 30);}
                }
        }
}
```

Figure 3.53 *GameScreen* UI

drawPausedUI method is used for drawing*Paused* state screen (walk screen) and the number of moving position on the walk screen.*Assets.turn_phase* and *Assets.turn_phasegot*are for displaying walk screen and walk result screen. *Assets.walkbutton* and *Assets.walkbuttonpush* are for displaying buttons in the walk screen.

3.6.6.5 Event

```
Public int random(){
	Random random= newRandom();
	int move;
	move=random.nextInt(6)+1;
	return move; }
Public void event(){
	Random Ran=newRandom();
	Intranflag,ranminigame,random;
	random = Ran.nextInt(2)+1;
	if(random == 1){
		Random Ranevent=newRandom();
		ranflag = Ranevent.nextInt(3)+1;
		if(ranflag == 1){
			ranflag=0;
			flag=0;
			game.setScreen(newEventran(game));}
		if(ranflag == 2){
			ranflag=0;
			flag=0;
			game.setScreen(newEventmovef(game));}
		if(ranflag == 3){
			ranflag=0;
			flag=0;
			game.setScreen(newEventmoveb(game));}}
	if(random == 2){
		Random Rangame=newRandom();
		ranminigame = Rangame.nextInt(2)+1;
		if(ranminigame == 1){
			ranminigame=0;
			flag=0;
			Settings.isminigame01(1);
			Settings.setsendminigamep2(2);
			game.setScreen(new Minigame01(game));}
		if(ranminigame == 2){
			ranminigame=0;
			flag=0;
			Settings.isminigame02(1);
			Settings.setsendminigamep2(2);
			game.setScreen(new Minigame02(game));}}
```

Figure 3.54 *GameScreen*: *event* method

A *random* method is used for generating a random number for moving positions in the walk screen when the player pushes walk button. An *event* method is used for triggering the event randomly or for starting the mini-game from flags received from the server and the client threads.

3.6.6.6 resume and dispose

```
Publicvoidresume() {
	this.Restore();
	if(Settings.player1=1){
		gsc = new Thread(newThreadJoin());
		gsc.start();
	}
	if(Settings.player2=1){
		gss = new Thread(newThreadHost());
		gss.start();
	}
}
Publicvoiddispose(){
	this.store();
}
publicvoid store(){
	Settings.setpositionp1(p1x);
	Settings.setpositionp2(p2x);
	Settings.save(game.getFileIO());
}
PublicvoidRestore(){
	Settings.load(game.getFileIO());
	if(Settings.setpositionp1< 0){
		Settings.setpositionp1(0);
	 }
	if(Settings.setpositionp2< 0){
		Settings.setpositionp2(0);
	 }
	if(Settings.setpositionp1>= 29){
		Settings.setpositionp1(29);
		flag=3;
	 }
	if(Settings.setpositionp2>= 29)
		Settings.setpositionp2(29);
		flag=2;
	}
}
```

Figure 3.55 *GameScreen*: *resume* and *dispose* methods

A *resume* method is called when *GameScreen* is started up. *resume* method will start the server or the client thread for communication and call a *Restore* method. The *Restore* method is used for loading all global variables from *Setting* class. A *dispose* method will be called when the *GameScreen* is changed to another screen; it calls *store* method. The *store* method is used for saving global variables in the *GameScreen* back to Setting class. The *GameScreen*on the client side will create and start *ThreadJoin* for communication while the server creates and starts *ThreadHost* for communication.

3.6.7 MainMenuScreen

```
PublicclassMainMenuScreenextends Screen {
Publicvoidupdate(floatdeltaTime) {
      Graphics g = game.getGraphics();
      List<TouchEvent>touchEvents = game.getInput().getTouchEvents();
      Intlen = touchEvents.size();
      for(inti = 0; i<len; i++) {
            TouchEvent event = touchEvents.get(i);
      if(event.type == TouchEvent.TOUCH_DOWN) {
            game.setScreen(new Setup(game));
      if(Settings.soundEnabled){
            Assets.click.play(1);
            return; }
publicvoid present(floatdeltaTime) {
      Graphics g = game.getGraphics();
      g.drawPixmap(Assets.Title, 0, 0);
      if(count < 12){
            g.drawPixmap(Assets.tapstart,g.getWidth()/2-120
            ,g.getHeight()/2+125);
      }
      if( count >15){
            count = 0;
      }
}
```

Figure 3.56 *MainMenuScreen*Class

The *MainMenuScreen* is the first screen of the game. *MainMenuScreen* class contains codes displaying the title of this game and codes to call the next screen (setup screen in the next section). The *update* method in the *MainMenuScreen* is for a touch event from the player; when the player touches the screen, the system will call setScreen method to change the screen. A *present* method is used for displaying pictures on the screen. *Assets.Title* displays the game's title and background. *Assets.tapstart* displays **Tab to start** picture on the screen.

3.6.8 Setup screen

```
Publicclass Setup extends Screen {
Publicvoidupdate(floatdeltaTime) {
      Graphics g = game.getGraphics();
      List<TouchEvent>touchEvents = game.getInput().getTouchEvents();
      game.getInput().getKeyEvents();
      intlen = touchEvents.size();
      for(inti = 0; i<len; i++) {
            TouchEvent event = touchEvents.get(i);
      if(event.type == TouchEvent.TOUCH_DOWN){
      if(inBounds(event,g.getWidth()/2-250, g.getHeight()/2+100, 425,125 ) ){
            game.setScreen(new Host(game));
      if(Settings.soundEnabled)
            Assets.click.play(1);
            return; }
      if(inBounds(event,g.getWidth()/2-250, g.getHeight()/2+200, 425,125 ) ){
            game.setScreen(new Join(game));
      if(Settings.soundEnabled)
            Assets.click.play(1);
            return;}}}}
publicvoid present(floatdeltaTime){
      Graphics g = game.getGraphics();
      g.drawPixmap(Assets.setup, 0, 0);
      g.drawPixmap(Assets.HOSTBUTTON, g.getWidth()/2-160,
      g.getHeight()/2+100);
      g.drawPixmap(Assets.JOINBUTTON, g.getWidth()/2-160,
      g.getHeight()/2+250);
      }
```

Figure 3.57 *Setup* screen

Setup screen is a screen for setting the player to be either the server or the client. An *update* method is for a touch screen event from the player. There are two bounds on this screen: host or join. When the player touches in the bounded area, the system will determine to change screen to be the server or the client. In this screen, *Assets.setup* is used for displaying the background of screen; *Assets.HOSTBUTTON* and *Assets.JOINBUTTON* are used for displaying buttons on screen that match the area of bound.

3.6.9 Mini-game and Event

3.6.9.1 Event

```
PublicclassEventmovebextends Screen {
Publicvoidupdate(floatdeltaTime) {
       Graphics g = game.getGraphics();
       List<TouchEvent>touchEvents = game.getInput().getTouchEvents();
       Intlen = touchEvents.size();
       for(inti = 0; i<len; i++) {
              TouchEventeventb = touchEvents.get(i);
       if(eventb.type == TouchEvent.TOUCH_DOWN) {
       if(inBounds(eventb,0 ,0, 1100,700 )&&set1==0){
              set1=1;
       if(set1==1){
              ranb=random(6)+1;
              if(Settings.player1==1){
                     Settings.setpositionp1-=ranb; }
              if(Settings.player2==1){
                     Settings.setpositionp2-=ranb; }}}
       if(inBounds(eventb,0 ,0, 1100,700 ) && next==1){
              if(Settings.player1==1){
                     if(Settings.playerturn==1){
                           Settings.setendp1(1);}
                     game.setScreen(newGameScreenC(game));}
              if(Settings.player2==1){
                     if(Settings.playerturn==1){
                           Settings.setendp2(1);}
                     game.setScreen(newGameScreenSv(game));}}}}}
publicint random(int number){
       intrannum=random.nextInt(number);
       returnrannum;
}
```

Figure 3.58 *Eventmoveb: update* method

Eventmoveb screen used *update* method to handle player input. When the player touches the screen, the system will call *random* method to generate a number for moving point of this player.

```
Publicvoidpresent(floatdeltaTime) {
	Graphics g = game.getGraphics();
	g.drawPixmap(Assets.moveb, 0, 0);
	if(set1==0){
		ran=random(6)+1;
		if (ran ==1){
			g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 25, 0,
			22, 30); }
		if (ran ==2) {
			g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 40, 0,
			22, 30);}
		if (ran ==3){
			g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 60, 0,
			22, 30);}
		if (ran ==4) {
			g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 80, 0,
			22, 30); }
		if (ran ==5) {
			g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 100,
			0, 22, 30);}
		if (ran ==6) {
			g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 120,
			0, 22, 30);}
	}
	if (ranb ==1){
		g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 25, 0, 22,
		30);
		next=1;  }
	if (ranb ==2)  {
		g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 40, 0, 22,
		30);
		next=1;  }
	if (ranb ==3) {
		g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 60, 0, 22,
		30);
		next=1; }
	if (ranb ==4)  {
		g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 80, 0, 22,
		30);
		next=1;  }
	if (ranb ==5) {
		g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 100, 0, 22,
		30);
		next=1;  }
	if (ranb ==6)  {
		g.drawPixmap(Assets.number, g.getWidth()/2+35, 717, 120, 0, 22,
		30);
		next=1;
	}
```

Figure 3.59 *Eventmoveb: present* method

A *present* method in the *Eventmoveb* screen is used for drawing the event screen and its result *Assets.moveb* is used for displaying the event screen. *Assets.number* is used for displaying the number in the event result.

3.6.9.2 Energy Charger mini-game (Mini-game1)

```
Publicclass Minigame01 extends Screen{
Publicvoidupdate(floatdeltaTime) {
      Graphics g = game.getGraphics();
      List<TouchEvent>touchEvents = game.getInput().getTouchEvents();
      game.getInput().getKeyEvents();
      intlen = touchEvents.size();
      for(inti = 0; i<len; i++) {
            TouchEvent event1 = touchEvents.get(i);
      if(event1.type == TouchEvent.TOUCH_DOWN||event1.type ==
      TouchEvent.TOUCH_UP) {
      if(inBounds(event1,g.getWidth()/2-100 ,650, 275,125 )&&stop==0 ) {
            Assets.click.play(1);
            set =0;
            if(event1.type == TouchEvent.TOUCH_UP){
                  count++;
                  set=1;
            }}
            return; }
      if(Currenttime>= 60&&stop==0){
            if(Settings.player1==1){
                  Settings.setdatagame01p1(count); }
            if(Settings.player2==1){
                  Settings.setdatagame01p2(count);}
                  stop=1;}
      if (inBounds(event1,0 ,0, 1280,800 )&&stop==1) {
            if(Settings.player1==1){
                  if(Settings.playerturn ==1) {
                        Settings.setstartplayer1(0);  }
                  game.setScreen(newGameScreenC(game));
                  Settings.setendminigame01p1(1);  }
            if(Settings.player2==1){
                  Settings.setreceivedatap1(0);
                  if(Settings.playerturn ==1){
                        Settings.setstartplayer2(0); }
                  game.setScreen(newGameScreenSv(game));
                  Settings.setendminigame01p2(1);  }
            return;}}
            y+=2;}
```

Figure 3.60 *Minigame02*: *update* method

The *Minigame02* screen uses *update* to handle the player input. When the player start touching a tab button, the system will count the number of times the player touch and keep the data in *datagame02*variables.*nanoTime*is a method for time counter in *Minigame01*. When the *nanoTime* is more than 60, the *Minigame01*will stop, and change screen back to the *GameScreen*.

```
Publicvoidpresent(floatdeltaTime) {
	Currenttime=  System.nanoTime();
	Currenttime -= Starttime;
	Currenttime /=100000000;
	Graphics g = game.getGraphics();
	g.drawPixmap(Assets.stage1,0 , 0);
	if(set == 1)  {
		g.drawPixmap(Assets.button,g.getWidth()/2-100 , 700);  }
	if(set ==0) {
		g.drawPixmap(Assets.button_push,g.getWidth()/2-100 , 706);  }
		g.drawPixmap(Assets.arrow,g.getWidth()/2 -30 , 580+y);
	if (y>30) {
		y=0;}
	g.drawPixmap(Assets.indicater,1050 , 600);
	if (Currenttime>60){
		g.drawPixmap(Assets.number,1125, 700, 5, 0, 22, 30);
	}
	if (Currenttime>50&&Currenttime<=60){
		g.drawPixmap(Assets.number,1125, 700, 25, 0, 22, 30);
	}
	if (Currenttime>40&&Currenttime<=50){
		g.drawPixmap(Assets.number,1125, 700, 40, 0, 22, 30);
	}
	if (Currenttime>30&&Currenttime<=40){
		g.drawPixmap(Assets.number,1125, 700, 60, 0, 22, 30);
	}
	if (Currenttime>20&&Currenttime<=30){
		g.drawPixmap(Assets.number,1125, 700, 80, 0, 22, 30);
	}
	if (Currenttime>10&&Currenttime<=20){
		g.drawPixmap(Assets.number,1125, 700, 100, 0, 22, 30);
	}
	if (Currenttime>=0&&Currenttime<=10){
		g.drawPixmap(Assets.number, 1125, 700, 120, 0, 22, 30);
	}
}
```

Figure 3.61 *Minigame01:present* method

The *present* method in the *Minigame01*screen is used for representing the screen, buttons and time of the second mini-game. *Assets.stage1*is used for displaying background of the *Minigame01*screen.*Assets.button* and *Assets.button_push* are for displaying action buttons in the *Minigame01*. *Assets.number* will display the time counter of the *Minigame01*.

3.6.9.3 Evasive game (Mini-game2)

```
Publicclass Minigame02 extendsScreen{
Publicvoidupdate(floatdeltaTime) {
	Graphics g = game.getGraphics();
	List<TouchEvent>touchEvents = game.getInput().getTouchEvents();
	game.getInput().getKeyEvents();
	intlen = touchEvents.size();
	for(inti = 0; i<len; i++) {
		TouchEvent event2 = touchEvents.get(i);
	if(event2.type == TouchEvent.TOUCH_DOWN||event2.type ==
	TouchEvent.TOUCH_UP) {
	if(inBounds(event2,75 ,600, 150,150) && next1==0 &&stop==0) {
		Assets.click.play(1);
		set1 =0;
		next2=0;
		if(event2.type == TouchEvent.TOUCH_UP) {
			set2=1;
			next1=1;
			count++;}
		return; }
	if(inBounds(event2,975 ,600, 150,150 )&& next2 == 0 &&stop==0 ) {
		Assets.click.play(1);
		set2 =0;
		next1=0;
		if(event2.type == TouchEvent.TOUCH_UP) {
			set1=1;
			next2=1;
			count++;}
		return; }
		realcount=count/2;
	if(Currenttime>= 60&&stop==0){
		if(Settings.player1==1){
			Settings.setdatagame02p1(realcount);  }
		if(Settings.player2==1){
			Settings.setdatagame02p2(realcount);  }
			stop=1; }
		if (inBounds(event2,0 ,0, 1280,800 )&&stop==1){
		if(Settings.player1==1) {
		if(Settings.playerturn ==1){
			Settings.setstartplayer1(0);}
			game.setScreen(newGameScreenC(game));
			Settings.setendminigame02p1(1);
		}
		if(Settings.player2==1){
		if(Settings.playerturn ==1){
			Settings.setstartplayer2(0);
		}
		Settings.setreceivedatap1(0);
		game.setScreen(newGameScreenSv(game));
		Settings.setendminigame02p2(1);}
	return;}}
	 }}
```

Figure 3.62 *Minigame02* :*update* method

```
publicvoid present(floatdeltaTime) {
	// TODO Auto-generated method stub
	Currenttime =System.nanoTime();
	Currenttime -= Starttime;
	Currenttime /=100000000;
	Graphics g = game.getGraphics();
	if(fin==0){
		g.drawPixmap(Assets.ionstorm,0 , 0);

		if(set1 == 1) {
			g.drawPixmap(Assets.lrelease,100 , 700);
		}
		if(set1 ==0) {
			g.drawPixmap(Assets.lpress,100 , 700);
		}

		if(set2 == 1) {
			g.drawPixmap(Assets.rrelease,1000 , 700);
		}
		if(set2 ==0)  {
			g.drawPixmap(Assets.rpress,1000 , 700);
		}
	}
}
```

Figure 3.63 *Minigame02*: *present* method

The *update* method in *Minigame02* is used for handling the player input. The player has to touch each button one at a time until the time is up. The system will count 1 point per 1 correct touch. The system will store data in *datagame02p1* (player1) *datagame02p1*(player2) variables. A *nanoTime* method is the time counter in the*Minigame02.* When the *nanoTime* is more than 60 *Minigame02,* the system will stop the mini-game and change the screen back to the *GameScreen.* The *present* method is used for representing the screens and button of this mini-game. *Assets.ionstorm* is used for displaying the background of the *Minigame02. Assets.lrelease, Assets.lpress, Assets.rrelease and rpress*are used for displaying button actions in the Minigame02.

3.7 Message model

3.7.1 Event

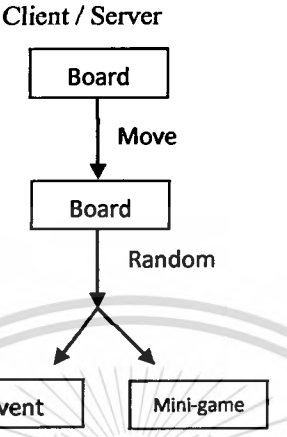


Figure 3.64 Walking phase

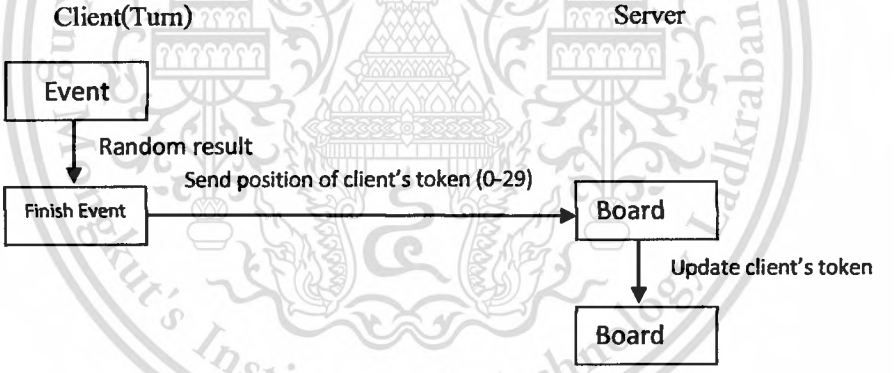


Figure 3.65 Event message model

After the player has moved and the program randomly chooses the event as the next action. The current player must resolve the event first before sending his update position to his opponent. In an example in **Figure 3.65**, assuming that this is the client turn which after the client moves, the event is chosen. The program first executes the event so that the player will get his new positions. Then, this new position number (between 0-29) will be sent to the server so that the server will update the client's position on his side.

3.7.2 Mini game

3.7.2.1 Client Turn

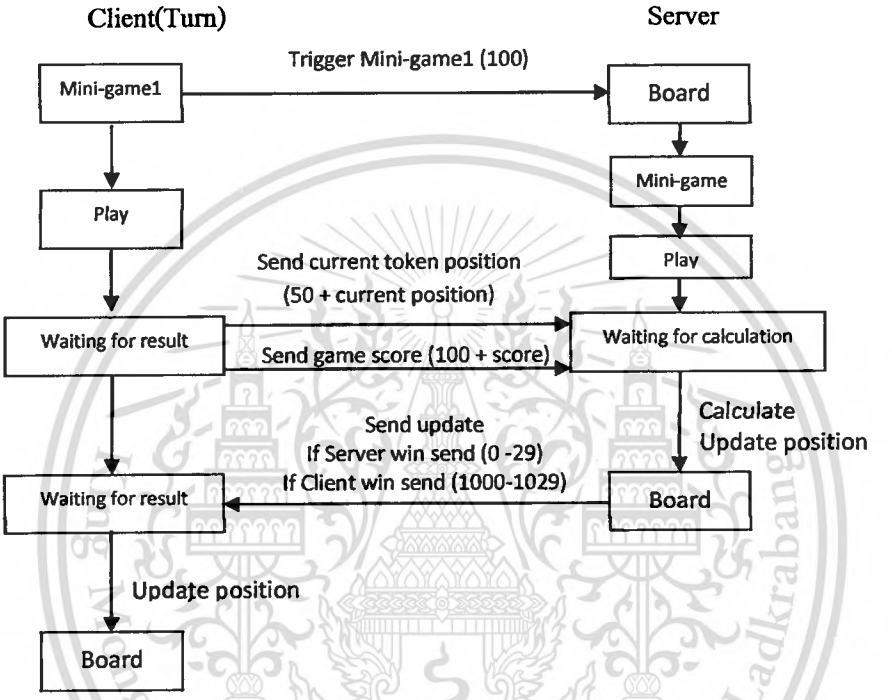


Figure 3.66 Mini-game 1 message model (Client Turn)

For the mini-game1, the messages processing between the two players are different. For example, assuming that this is the client turn, when the client gets the mini-game, the mini-game value (100)will be sent to the server triggering it to run the mini-game1 process. Then, both players will play the mini-game. After the client finishes the mini-game, the program on his side will send his current token's position added up by 50 (50 + token's position(0-29)) together with his score added by 100 (100+score) to the server. For the server side, when the received value is between 50 to 79, it will subtract that data with 50 for the new client's token position. When the server gets the value between 100 to 199,it will subtract with 100 and keeps this data in *ScoreFromClient* variable.

The server then will compare his player's score with the client's score (*ScoreFromClient*). If the server's score is higher, the server will update his position on his device first and send his new position (similar set of values to sending the event result) to the client. Otherwise, the server will update the client position on his device and send the client's new position added with 1000 (1000+new client's position) to the client device.

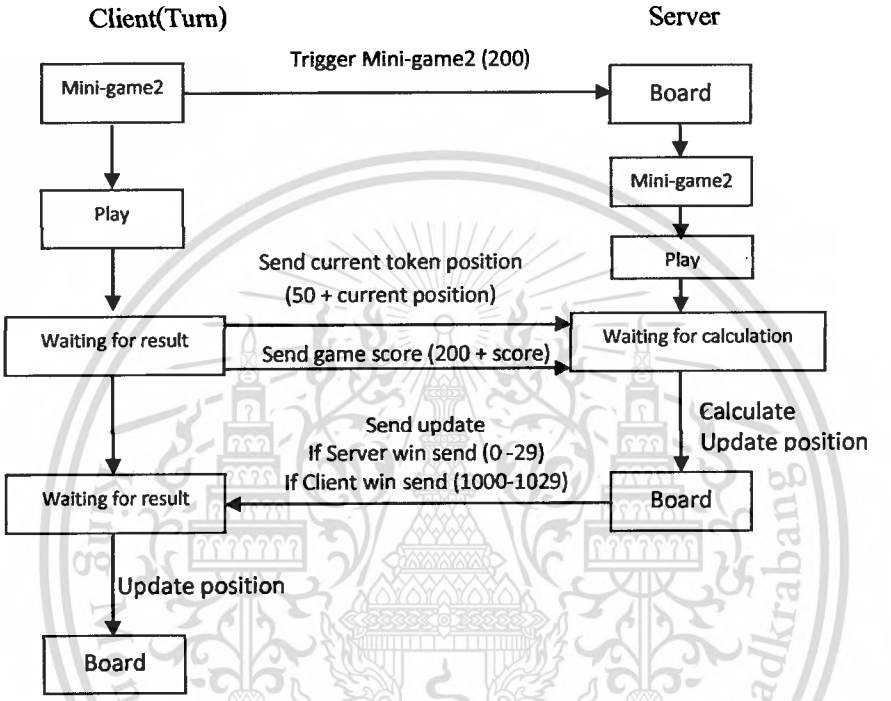


Figure 3.67 Mini-game 2 message model (Client Turn)

For the mini-game 2, the message model is similar to the mini-game 1. The difference is the triggering number for the mini-game 2 is 200 and the score values are between 200 to 299(Ex.234) instead of 100 to 199.

3.7.2.2 Server Turn

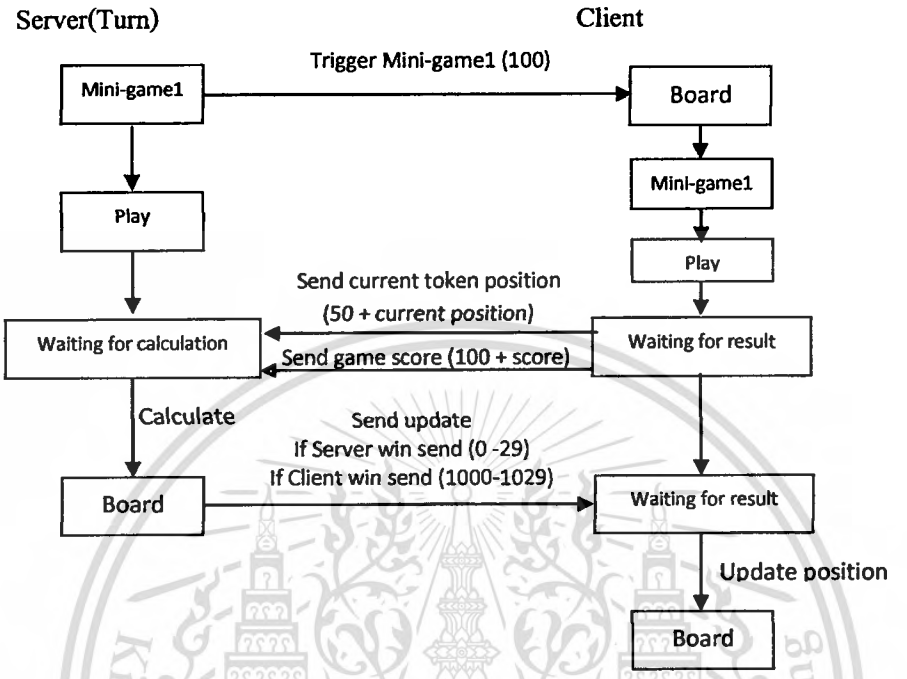


Figure 3.68 Mini-game 1 message model (Server Turn)

In case of the server side turn, the server will send the mini-game triggering value to tell the client. After both players play the mini-game, the flow will be the same as in the mini-game1. The server will be the one who calculates the result and the client side must wait for the result from the server.

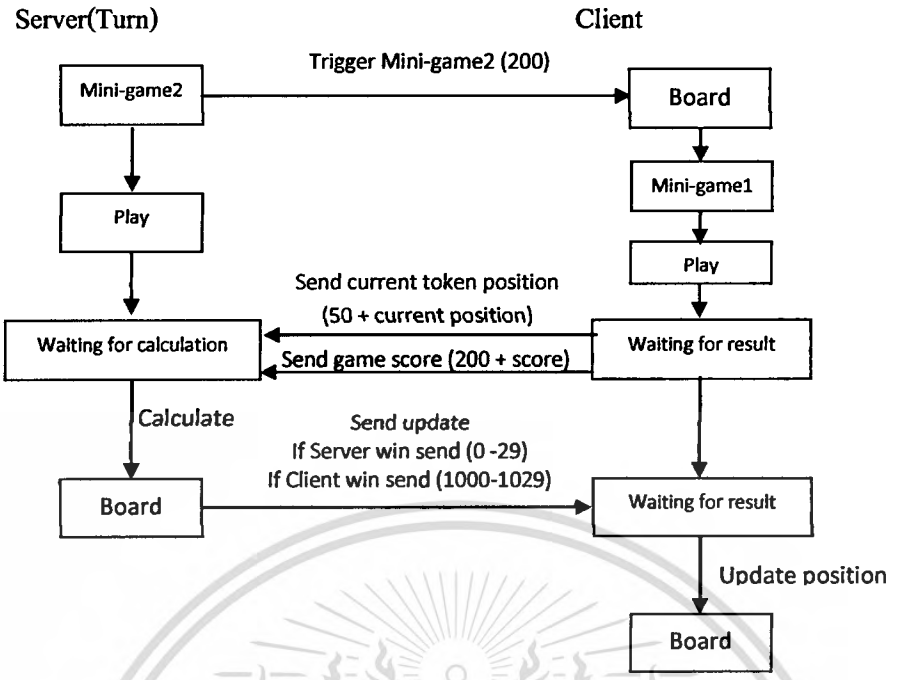


Figure 3.69 Mini-game 2 message model (Server Turn)

In mini-game 2 the trigger message is use 200 and the score format is use 200 + score instead.

Chapter 4

Implementation

This chapter explains how to implement our game on both the android and iOS devices.

4.1 Deploying the game to the iOS tablet device or iPad on AppStore

For the iOS device, we need to send our application to the apple tech support team to analyze and generate the certificate key for the application first before it can be downloaded and deployed on the AppStore. Our application is under the university developer program which can be deployed for testing purpose via Xcode with the registered device only. It is not allowed to be published on the AppStore.

4.2 Deploying the game to the android tablet device

For android, the application will be in the *.apk* format which can be downloaded or put in the device via the data link between the PC and device. Then, click at the directory that the downloaded or copied file is stored.

4.3 Game overview

Before starting the game, players must turn on their Wi-Fi on their devices. The **Nyorion** space race game is a turn-based game which 2 players need to race to the third planet of the **Nyorion** System. The one who reaches this planet first will win.

4.4 Game play

The main menu of the game will be very simple.

For the iOS device, see the screen in Figure 4.1 which after tapping the **Tap To Start** button the view changing code as described in Subsection 3.5.4 will be called to change the current screen to the setup screen.

For the android device, *LoadingScreen* in Section 3.6.5 is called to load picture to display the main menu screen. After tabbing the **Tab To Start** in Figure 4.2, the program will call *SetScreen* method in Subsection 3.6.1.1 to change to the *setup screen*.

Figure 4.1 Game Title for iOS

Figure 4.2 Game Title for android

Then, the Setup screen will ask each player to choose the role for his device. There are 2 buttons namely **Host button** and **Join button. Host button** is for starting the game as the server (host) which the system will call the code in Subsection 3.5.1(iOS) or thread *starthost* in Subsection 3.6.2.1(android) to create the socket to listen for the connection and when there is a connection the server's view will change to the game board screen. **Join button** is for the player when he plays the game as the client which the system will run the code in Subsection 3.5.2(iOS) or thread *startjoin* in Subsection 3.6.2.2(android) to connect to the server.

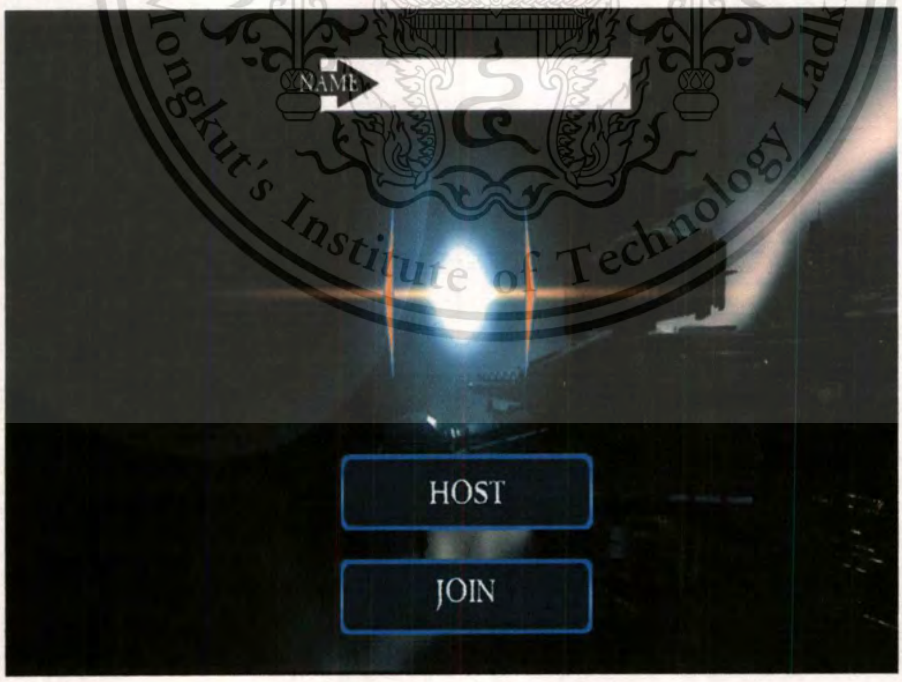


Figure 4.3 Setup screen

Next, the game will proceed to the Game board screen as in Figure 4.4 for iOS and Figure 4.5 for android. The player tokens will be red and blue for the server and the client roles respectively. Placing the token on the space uses codes in Subsection 3.6.6.3 for android and codes in Subsection 3.5.6 and 3.5.7 For iOS devices.

Figure 4.4 Game board screen for iOS

Figure 4.5 Game board screen for android

There are some different steps in playing on each platform. For the iOS device, the player has to click **Next button** in the board screen (Figure 4.4) to proceed. See Subsection 3.5.4 for this screen's iOS codes which will use the change view function. For the android [redacted] device, he can click anywhere on the screen (Figure 4.5).See the *gamescreen* class in Subsection 3.6.6.1for this screen's android codes. The game will proceed to the walking screen as shown in Figure 4.6.In the walking screen, the player must tab on for his random number of move(1-6). The system generates this number randomly using*arc4random* function for iOS or method *random* for android (See Section 3.6.6.5).

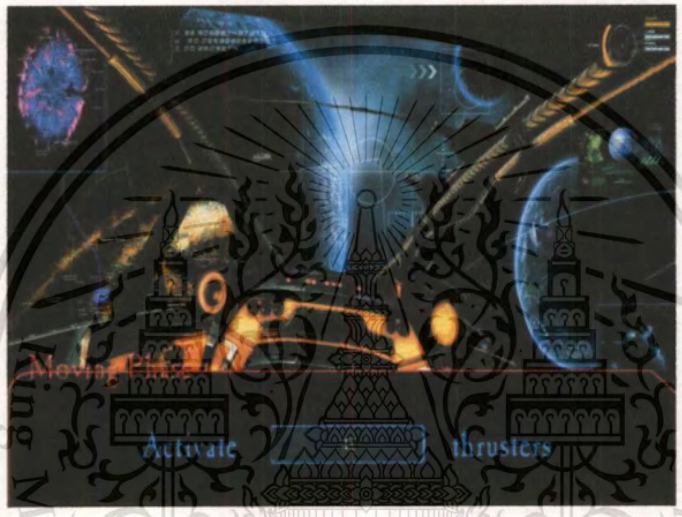


Figure 4.6 Walking screen

Then, the result as in Figure 4.7(iOS) or Figure 4.8 (android) will be displayed.

Figure 4.7 Walking result screen for iOS

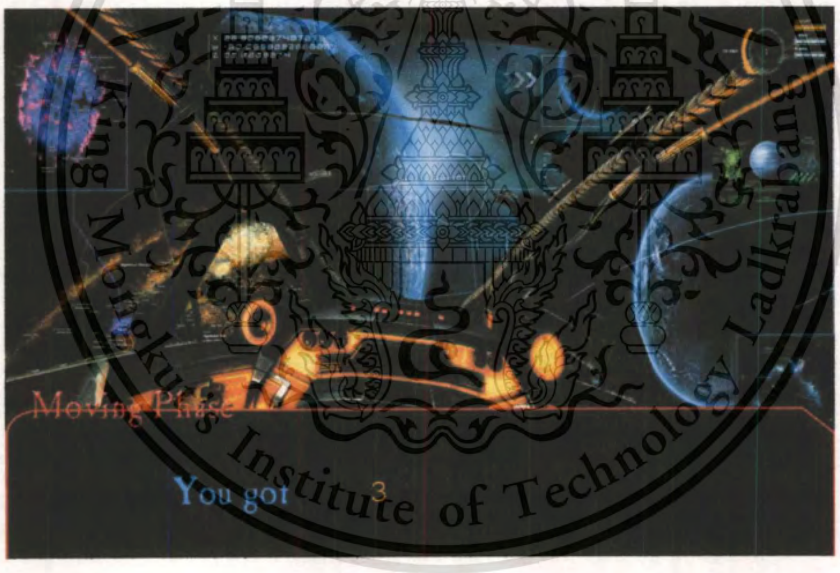


Figure 4.8 Walking screen for android

After that the system will move the player's token according to the point he gets. See codes in Subsection 3.5.7 for iOS and the *update* method in Subsection 3.6.6.1 for android which will result in either the event or the mini-game for you to play. To determine whether the player will get the event or has to play the mini-game the android uses the *event* method which furthers randomly choose which event the program will execute and go to that particular event page(see Subsection 3.6.6.5).

4.4.1 Events

There are 3 events. The first event is **Warp Relay** (see Figure 4.9 and 4.10). This event will move the player token forward. The number of move is also determined randomly by the program.

Figure 4.9 Warp Relay for iOS

See codes in Subsection 3.6.9.1 for the Warp Relay screen for iOS. See codes in Subsection 3.6.9.1 for the Warp Relay screen for android.

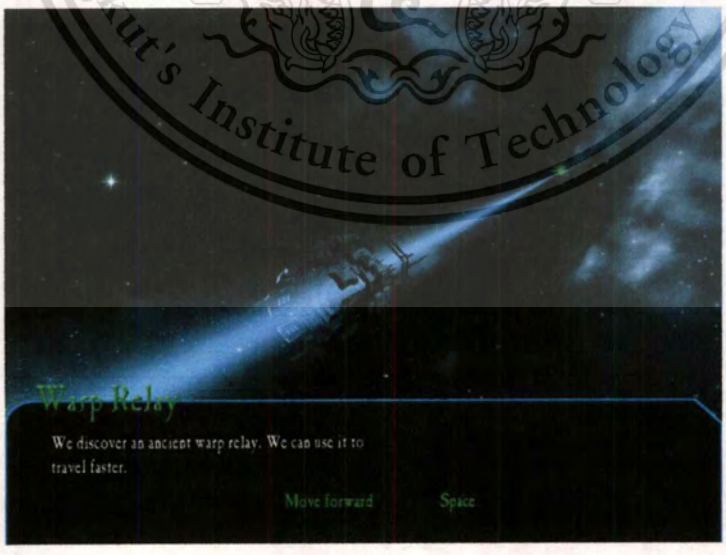


Figure 4.10 Warp Relay for android

The second event is **Parallel Universe!**(see Figure 4.11 and Figure 4.12).This event will move your token backward. The number of move is determined randomly by the program. Codes for Parallel Universe Event are implemented similarly to Warp Relay (just the moving the direction is opposite).

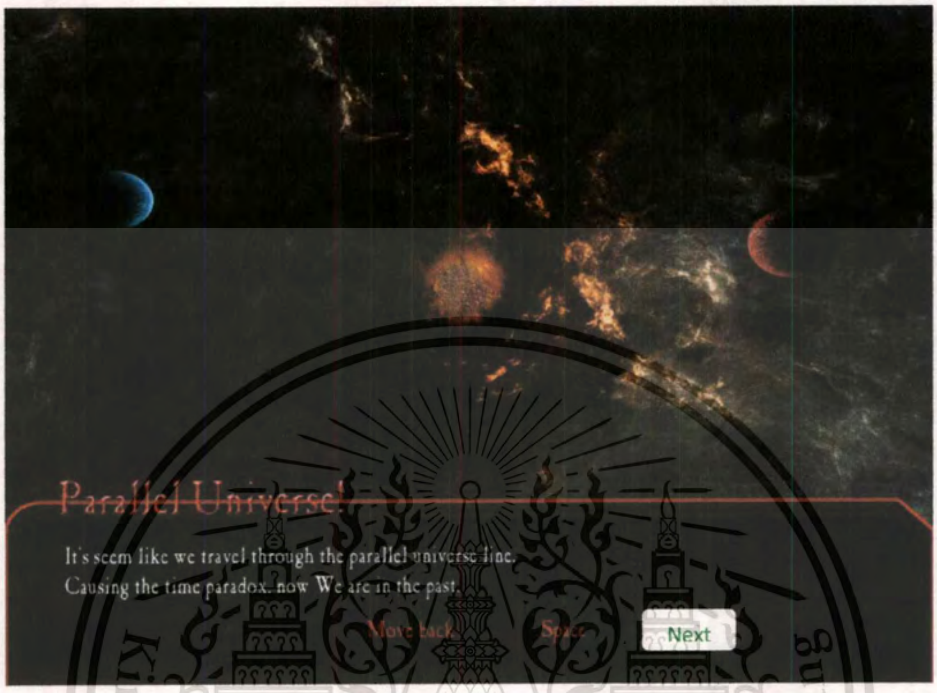


Figure 4.11 Parallel Universe for iOS

Figure 4.12 Parallel Universe for Android

The third event is **Warp Storm!**(see Figure 4.13-4.18).This event consists of two random actions. The first random action is for the direction of the player token (backward or forward). The second random action determines the number of moves for the player's token. Figure 4.13(iOS) and Figure 4.14(android) show the first random action screens which are for the direction of the player token (backward or forward). If the first random action is for moving forward, the good Warp Storm screen for iOS (Figure 4.15) or for android (Figure 4.16) will appear. Otherwise, the bad Warp Storm screen for iOS (Figure 4.17) or for android (Figure 4.18) will appear instead. Codes for Warp Storm Event are implemented similarly to Warp Relay and Parallel Universe combined together plus the additional task in randomly selects the direction.

Figure 4.13 Warp Storm for iOS

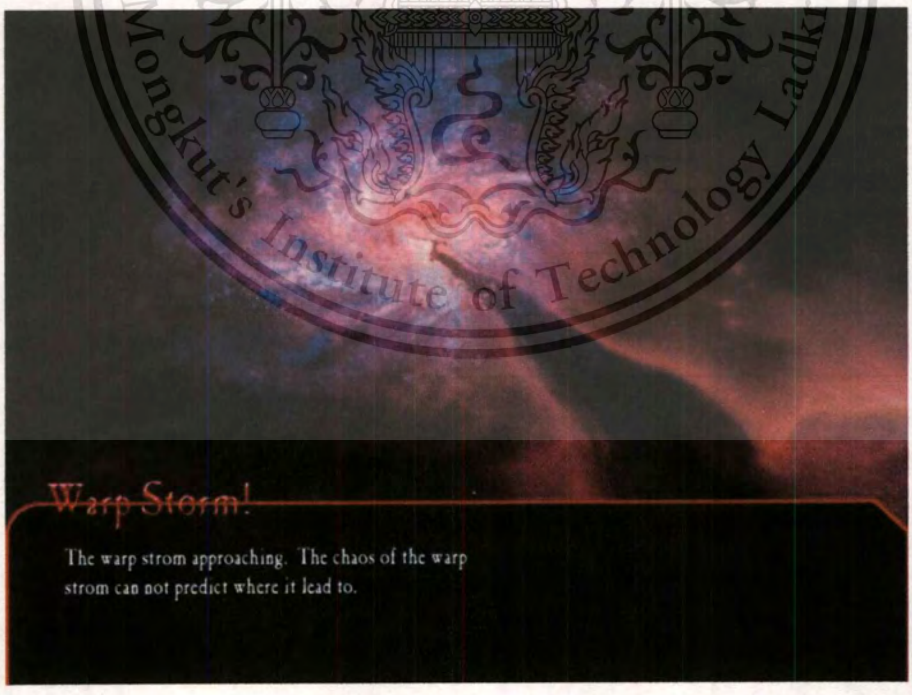


Figure 4.14 Warp Storm for Android

Figure 4.15 Good Warp Storm for iOS

Figure 4.16 Good Warp Storm for android

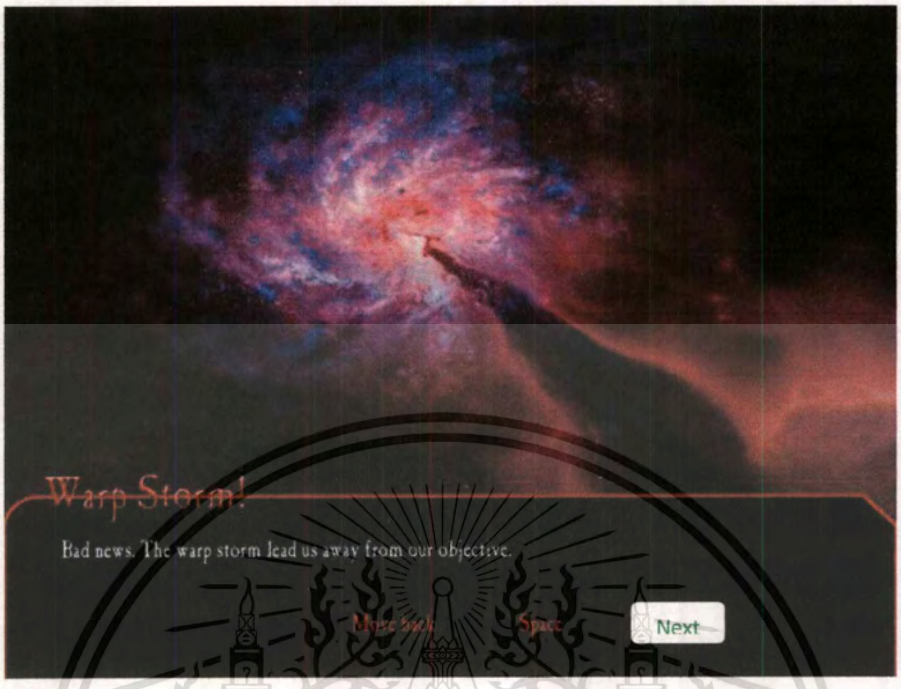


Figure 4.17 Bad Warp Storm for iOS

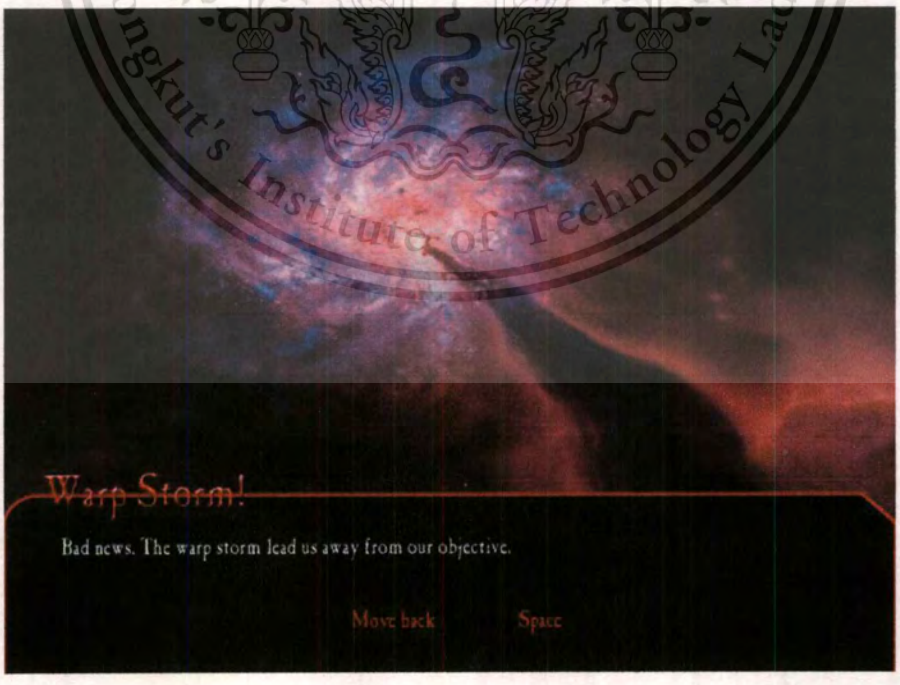


Figure 4.18 Bad Warp Storm for android

4.4.2 Mini-games

The first mini-game called the **Evasive game** as shown in Figure 4.19**.** The objective of this game is to tap the button on the left and right sequentially as much as possible before the time runs out. The game will handle the score and sending data to the other player. See Subsection 3.5.3.1 for this mini-game code in iOS and subsection 3.6.9.3 for the codes in android.

Figure 4.19 Evasive game mini-game for iOS

Figure 4.20 Evasive game mini-game for android

The second mini-game called the **Energy Charger** game in Figure 4.20for iOS and Figure 4.21 for android. In this game, both players need to press the middle button as fast as possible before the time runs out. See Subsection 3.5.3.1 for this mini-game code in iOS and subsection 3.6.9.2 for the codes in android.

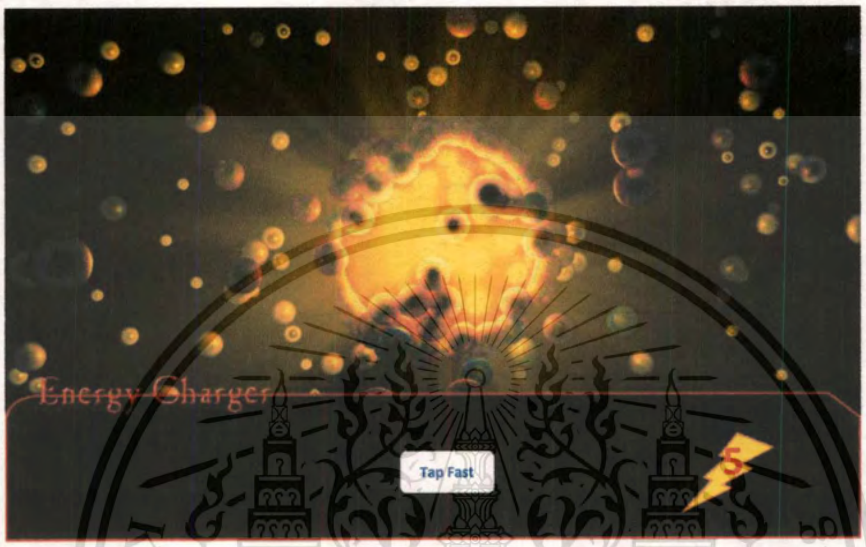


Figure 4.21 Energy Charging mini-game for iOS

Figure 4.22 Energy Charging mini-game for android

After your token arrives at the third planet or the center of the board, the screen as in **Figure 4.23** will appear. This means you are the winner of the game. If your opponent arrives at this destination first, the screen as in **Figure 4.24** will appear. This mean you lose.

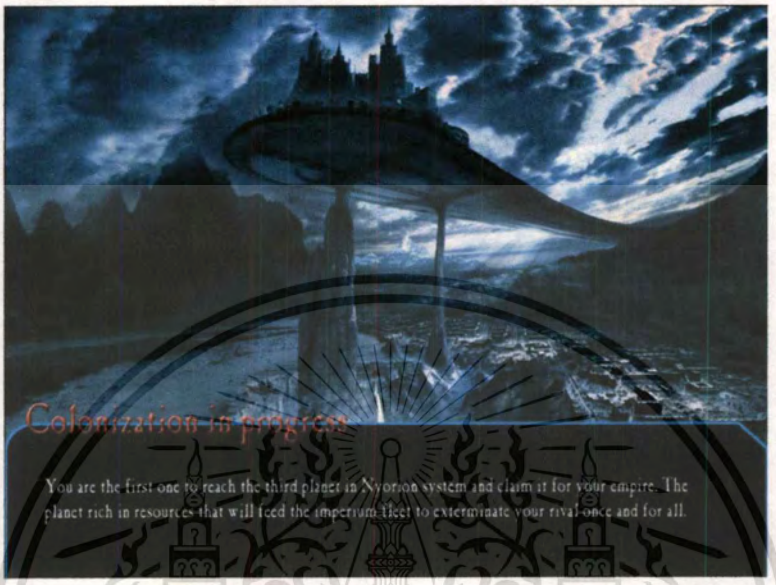


Figure 4.23 Good ending

Figure 4.24 Bad ending

Chapter 5

Conclusion and Problem

5.1 Conclusion

In this special project, we developed a Wi-Fi cross platform game called "Nyorion". The game is a 2 player turn based board game which the player who arrives at the destination first will win. In each turn, the player's number of moves is determined randomly by the system. The system then moves the player's token according to that move point and randomly chooses between the event or the mini-game for the player. The event is an action which the system moves the player token's backward or forward. The mini-game is an action which both players have to compete and the one who wins the game, his token will be moved forward which the number of moves depends on how different their scores are after the mini-game is finished. At this stage, there are 3 events and 2 mini-games. Players have to continue playing until any one of them reaches the destination.

The main objective of the project is to study how to develop a none web-based game which can be played between any iPads and Android tablets through Wi-Fi connection. To develop this game, we divided our task into two phases: the design phase and the implementation phase. In the design phase, we designed the story board of our game, the screen interfaces, all system diagrams and messages format together. In the implementation phase, one of us developed the game on android while another developed on iOS. Our android game was developed using android version 3.0 and works best with the screen with resolution 1280x800. For the iOS version, we developed using Xcode version 4.3 and works best with iPad. During developing this game, there were several problems arisen due to the differences between languages (java for android and objective C for iOS) and the developers' lack of experience in developing object oriented program. We found that even though the design is the same, the implementation is done independently. Only the concept can be shared.

5.2 Problems

5.2.1 Problems on Android

- Android development did not run smoothly due to the very slow emulator. Since, we had to test the program on two machines, we tested with the same platform first. For android, we used one PC and one android tablet (for each player). For the PC, we ran our game on the emulation. The problem is that when the game was run on the emulator, it was much slower than running on the tablet (we felt that it was ten times slower). Since we ran and test the programs all the times, it took us so long to finish each function. We note that we also tested using one MacBook Pro and one iPad1 for the iOS version. There was no problem with the Xcode emulator.
- Android emulator does not support server function (cannot be run as a server). Since at first we did not know, it took us quite a while to figure this out.
- In the Android development, ***force close*** sometimes appeared with no cause. Indeed, it showed errors on the log but these errors did not tell the real cause so that we had to debug the code ourselves line by line.
- Since the socket programming was quite new to us. Also, android implemented socket using thread only. Thread programming in java was also new to us. We found that to implement codes to control the socket thread across the different screens, it was very complex. For example, sometimes the threads were still running even though the corresponding thread(s) on the opposite site was(were) closed. To solve this problem, we had to totally uninstall and reinstall the game again so that the game can be run.
- Sometimes, the game had ***force close*** problem when it was run on the emulator. However, there was no problem at all when the same version was run on the tablet. We guess that it was because there was not enough space in the emulator.

5.2.2 Problem on iOS

- Since objective C was new to us, it took time to understand and develop the code.

5.3 Suggestions and future work

- Implementing the cross-platform mobile game application separately on each platform such as our game is quite time consuming because it is similar to implement two applications even though the design is the same. This is because each platform uses different programming language to develop the application. So we suggest to use the engines such as Unity[16], Unreal engine[17] that provides feature that can port the game into several mobile platforms. For example, Unity provides the export function which can convert the unity file into .apk (for android) and .ipa (for iOS).
- More game events and mini-games should be added. However, since we used different values of the number to indicate different values for the mini-games. These numbers tied tightly with the type of the mini-game such as number 100-199 for the mini-game1 and 200-299 for the mini-game2. Adding more mini-game means the program has to add more codes to check for the new mini-game numbers and the new codes must be recompiled.
- The save game function can be implemented to save the state of the game so the player can save the game to play later.
- The score, money, or the battle system can also be implemented to improve the variety to the game. There should be a variety of tokens for the player to choose.
- Our game can be played over the Internet if the device uses the valid global IP address and is configured properly in case the NAT(network address translation is used).
- For more experienced developers, we believe that the game can be developed much better. For example, in some game screens, the appearance on each platform is slightly different or playing on the iOS device, the player has to press **Next button** twice so that his device (acting as the client) can establish the connection to the server but the android player does not have to press any button.
- For methods used by several classes, we suggest to declare them as global methods.

Reference

[1][Online] http://en.wikipedia.org/wiki/Video_game

[2][Online] http://en.wikipedia.org/wiki/Video_game_genres

[3][Online] http://en.wikipedia.org/wiki/Smartphone

[4][Online] http://en.wikipedia.org/wiki/Tablet_computer

[5][Online] http://developer.android.com/guide/basics/what-is-android.html

[6][Online] http://en.wikipedia.org/wiki/Ios

[7][Online] http://en.wikipedia.org/wiki/Computer_network_programming

[8][Online] http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[9][Online] http://en.wikipedia.org/wiki/User_Datagram_Protocol

[10][Online] http://en.wikipedia.org/wiki/Cross-platform

[11][Online] http://en.wikipedia.org/wiki/Android_SDK#Android_SDK

[12][Online] http://en.wikipedia.org/wiki/Xcode

[13][Online] http://en.wikipedia.org/wiki/Eclipse_%28software%29

[14][Online] http://developer.apple.com/library/ios/navigation/

[15] [Online] http://homepages.ius.edu/rwisman/C490/html/tcp.htm

[16] [Online] http://unity3d.com/

[17] [Online] http://www.unrealengine.com/

Appendices

Appendix A

Game Manual

Game overview

Before starting the game, please make sure the Wi-Fi connection is on and the device is connected. The Nyorion space race game is a turn-based game which 2 players need to race to the third planet of the Nyorion System. The one who reaches this planet first will win.

FigureA.1 Game Title for iOS

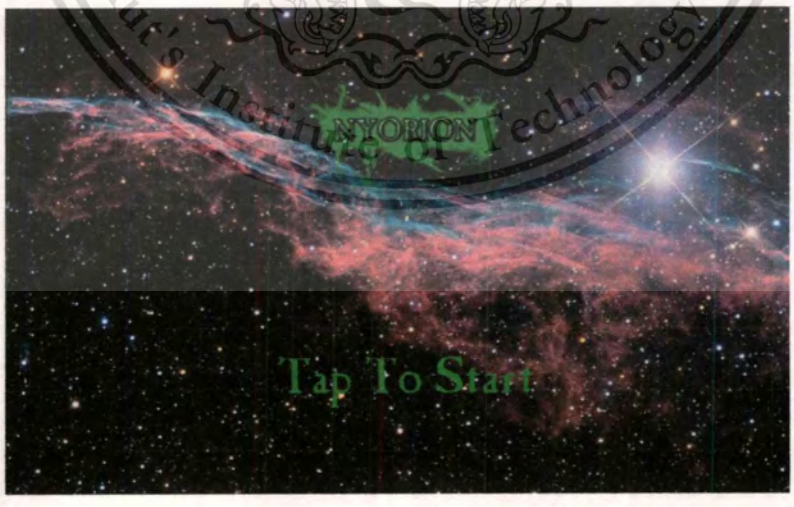


Figure A.2 Game Title for android

Game play

See Figure A.1 for the main menu of the iOS device and Figure A.2 for the main menu of the android device. These menus will appear when you start playing the game.

1. Tap any key to start the game. Note that for iOS menu, you have to tab inside the button. For android menu, you can tab anywhere on the screen. Then, the setup screen as in Figure A.3 will appear.

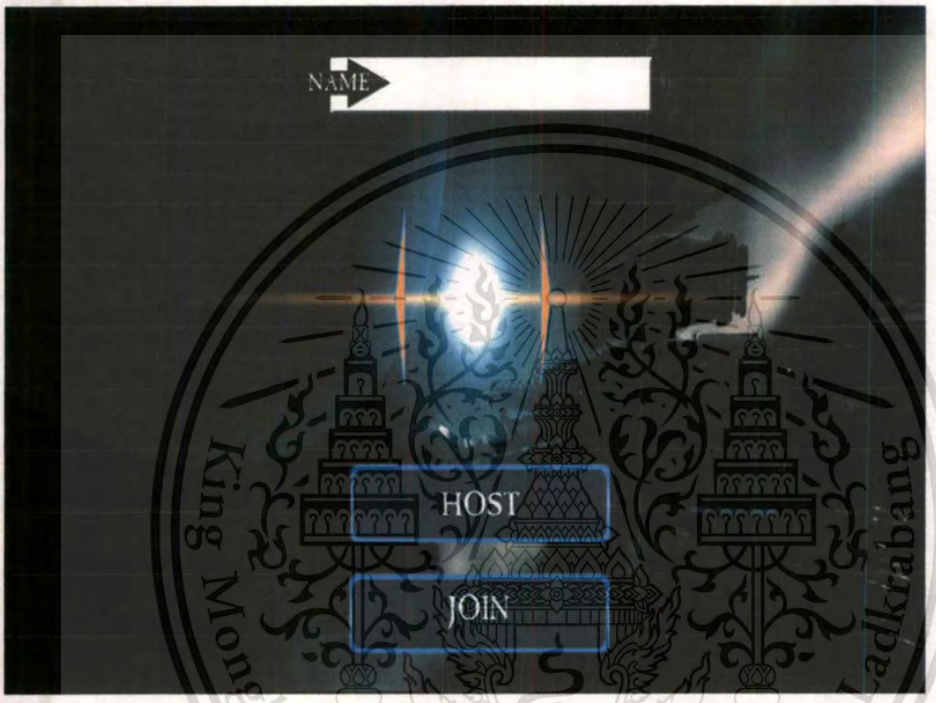


Figure A.3 Setup Screen

2. The setup screen (Figure A.3) will ask you to enter your name and choose the role for your device. If you want to start the game as the server, touch **Host button**. If you friend has already chosen to be the server, touch **Joinbutton** for joining the game as the client.

3.Next, the screen as in Figure A.4 (iOS) or Figure A.5 (android) will appear. If your device take the server role, your token will be red and will start playing from the top left. If your device takes the client role, your token will be blue and will start playing from the bottom right.

Figure A.4 Game board screen for iOS

Figure A.5 Game board screen for android

4. There are some differences to play this step on each platform. For the iOS device, you have to click **Next button** in the board screen (Figure A.4) to proceed; however, for the android device, you can click anywhere on the screen (FigureA.5).

5. After clicking, the walking screen as in Figure A.6 will appear. You need to tap [redacted] to roll for your moving score.

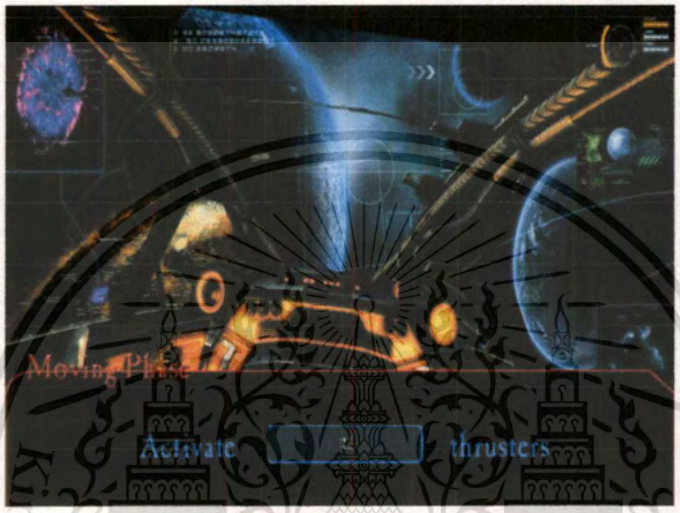


Figure A.6 Walk screen

6. The score screen as in Figure A.7 for iOS or Figure A.8 for android will appear. The system will automatically move your token according to the point you get and then will further randomly choose between the event or mini-game for your action. The game will perform different actions for the event and the mini-game.

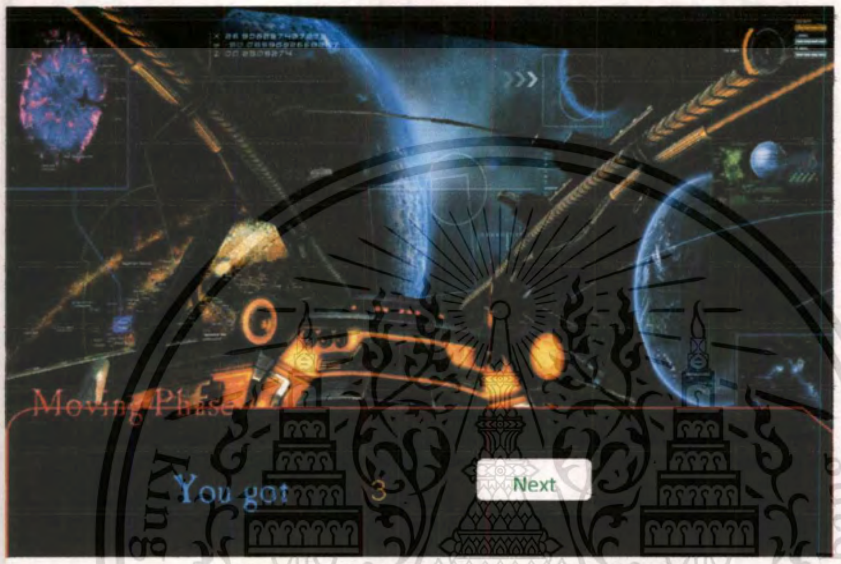


Figure A.7 Walking result screen for iOS

Figure A.8 Walking result screen for android

7. If the **event** is chosen. There are 3 events: **Warp Relay, Parallel Universe!** and **Warp Storm!**. The system will randomly perform one event.

7.1 The first event is **Warp Relay** (see Figure A.9 and Figure A.10). This event will move your token forward. The number of moves is determined randomly by the program.

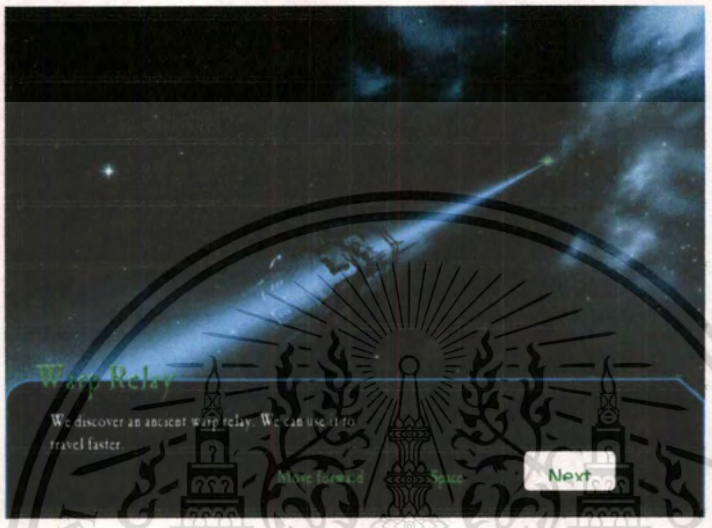


Figure A.9 Warp Relay for iOS

Figure A.10 Warp Relay for android

7.2 The second event is **Parallel Universe!**(see Figure A.11 and Figure A.12).This event will move your token backward. The number of moves is also determined randomly by the program.

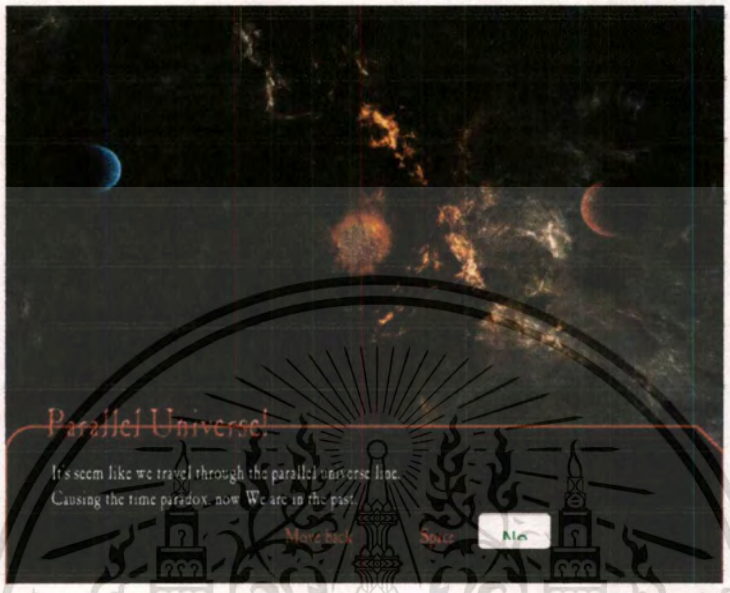


Figure A.11 Parallel Universe for iOS

Figure A.12 Parallel Universe for android

7.3 The third event is **Warp Storm!**(see Figure A.13 and Figure A.14).This event consists of two random actions. The first random action is for the direction of your token (backward or forward). The second random action determines the number of moves for your token. If the first random action returns forward, the good Warp Storm screen as in Figure A.15 or Figure A.16 will appear. If the first random action returns backward, the bad Warp Storm screen as in Figure A.17 or Figure A.18 will appear.

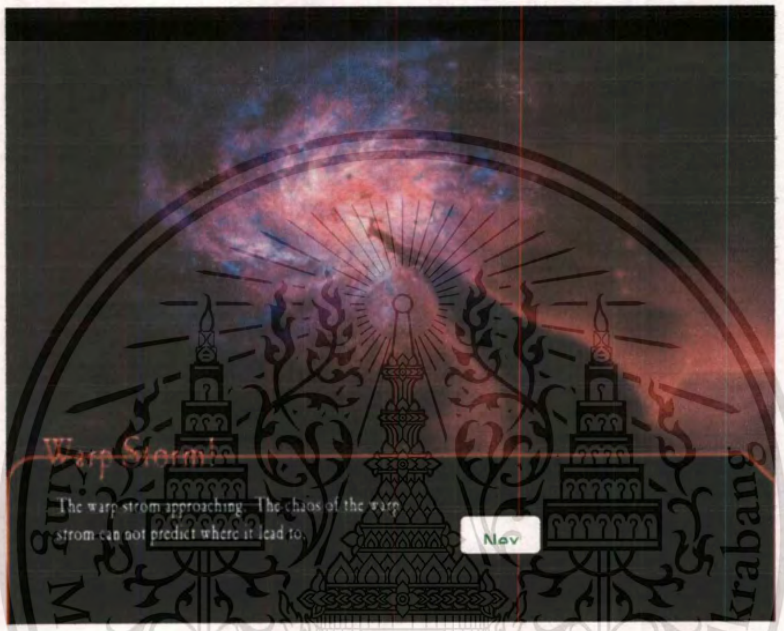


Figure A.13 Warp Storm for iOs

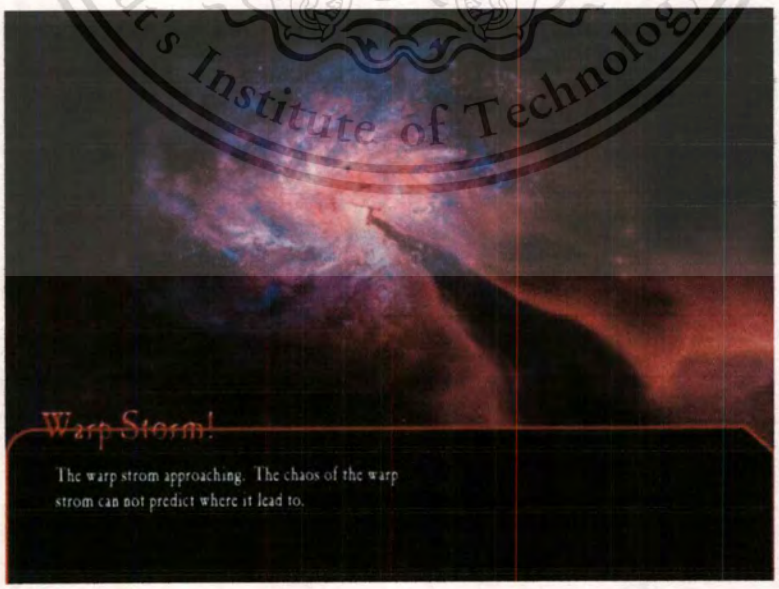


Figure A.14 Warp Storm for android

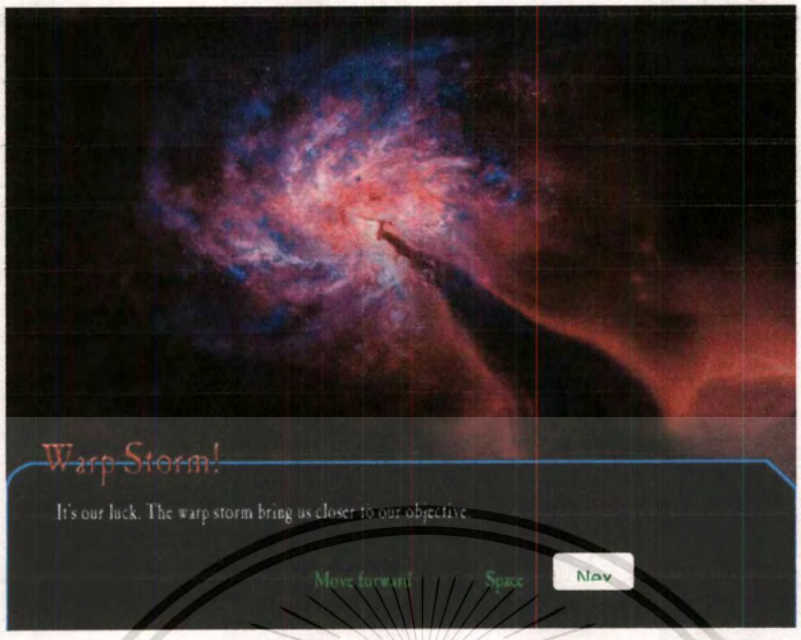


Figure A.15 Good Warp Storm for iOS

Figure A.16 Good Warp Storm for Android

Figure A.17 Bad Warp Storm for iOS

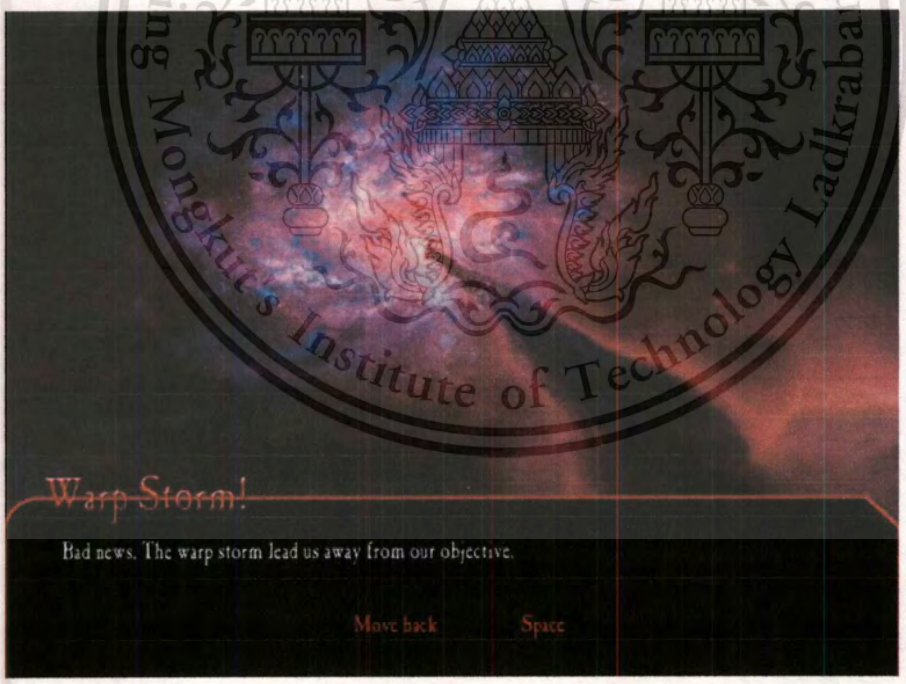


Figure A.18 Bad Warp Storm for android

8.If the **mini-game** is chosen. In our game there are two mini-game.

8.1 The first mini-game is called **Evasive game**. In this game you have to press left and right buttons one at a time as fast as possible in 5 seconds. The game screen as in Figure A.19 and Figure A.20 are shown for iOS and android respectively.

Figure A.19 Evasive game mini-game for iOS

Figure A.20 Evasive game mini-game for android

8.2 The second mini-game called **Energy Charger**. In this game, you have to tap the button as fast as possible in the limited time of 5 seconds. The game screen as in Figure A.21 and Figure A.22 are shown for iOS and android respectively.

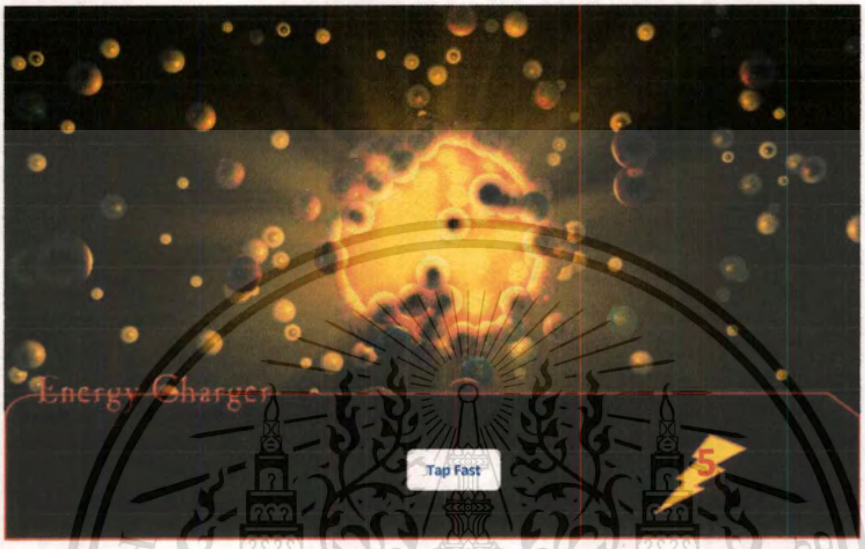


Figure A.21 Energy Charging mini-game for iOS

Figure A.22 Energy Charging mini-game for android

9. After your token arrives at the third planet or the center of the board, the screen as in Figure A.23 will appear. This means that you are the winner of the game .If your opponent arrives at this destination first, the screen as in Figure A.24 will appear. This mean you lose.

Figure A.23 Good ending

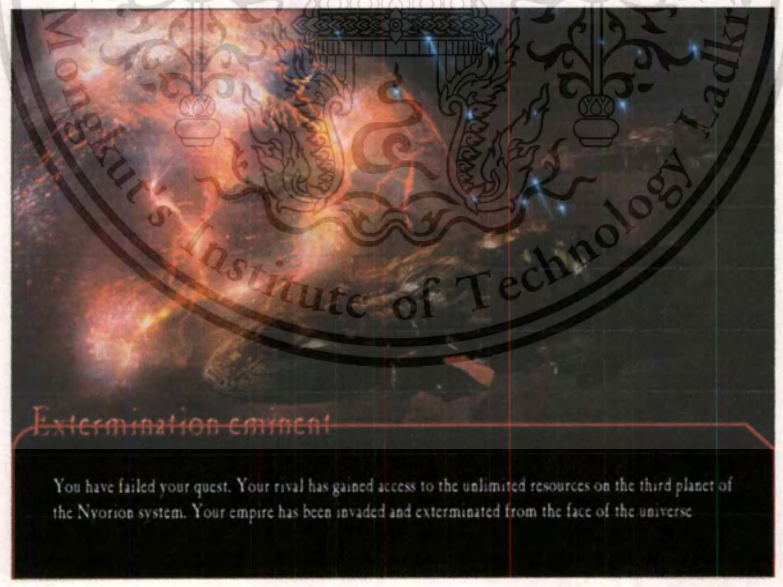


Figure A.24 Bad ending

Appendix B

Xcode Installation

Install Xcode

1. First go to https://developer.apple.com/xcode/ in your apple computer. Then click view download as shown in Figure B.1

Figure B.1 Xcode File

2. Sign in with your Apple ID (*the same one that you use for iTunes and app purchases*) (see Figure B.2).

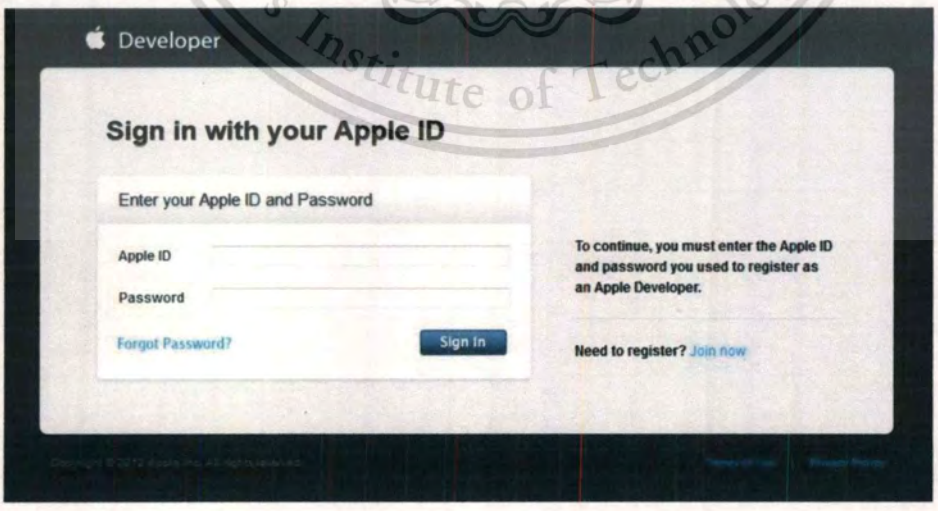


Figure B.2 Sign in screen

3. Find Xcode 4.3.2 for your Mac OS version and click on the .dmg link to download it as shown in Figure B.3.

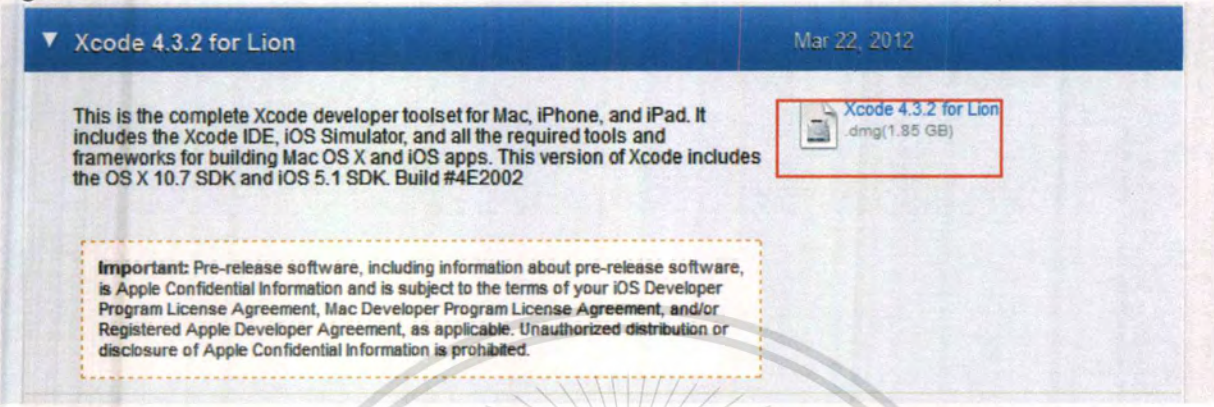


Figure B.3 .dmg file

4. Once the .dmg has finished downloading, it should automatically mount the disk image and open a window that looks like one in Figure B.4 in your Finder. Double-click on the "Xcode" package installer.

Figure B.4 DMG image

5.Once the installer launches, make sure all the checkboxes are checked, as shown in Figure B.5. Then, click "Continue" and go through the rest of the installation.

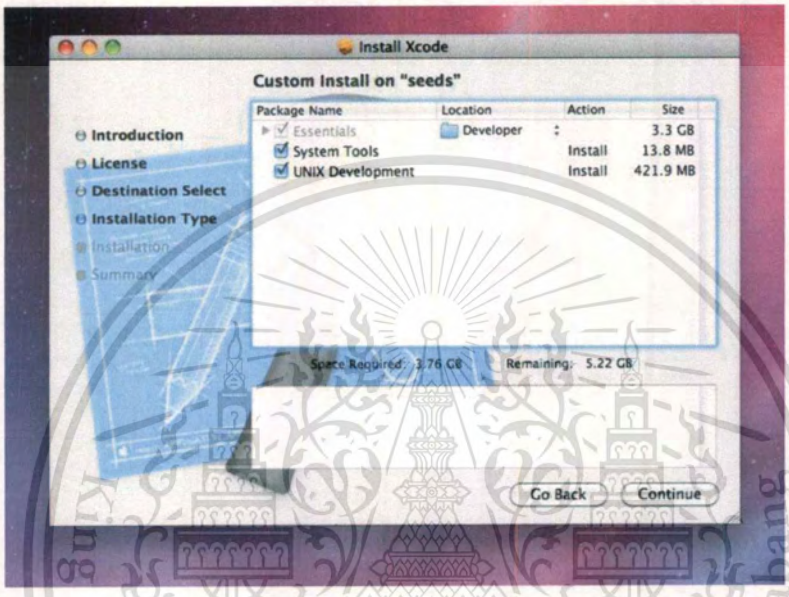


Figure B.5 Install Xcode

Appendix C

Deploy application on iPad

To test the application on your iPad device. It can be done according to the following steps

1. Register as Apple Developer. Only registered accounts can deploy to actual devices.
2. Open up Xcode 4's Organizer and select "Provisioning Profiles" in the left side menu (see Figure C.1).

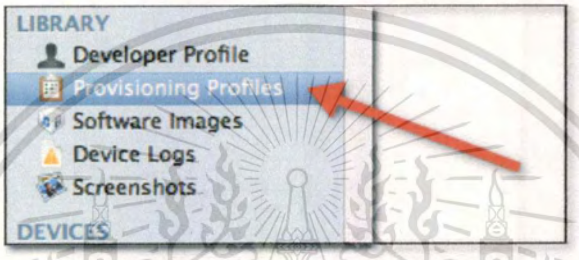


Figure C.1 Provisioning Profile

3. On the bottom of the right side, select the checkbox that labeled "Automatic Device Provisioning" (see Figure C.2).

Figure C.2 Provisioning Profile select

4. Plug your iPad device into your Mac computer. Select device in the left side menu, and click the "Use for Development" button as shown in Figure C.3

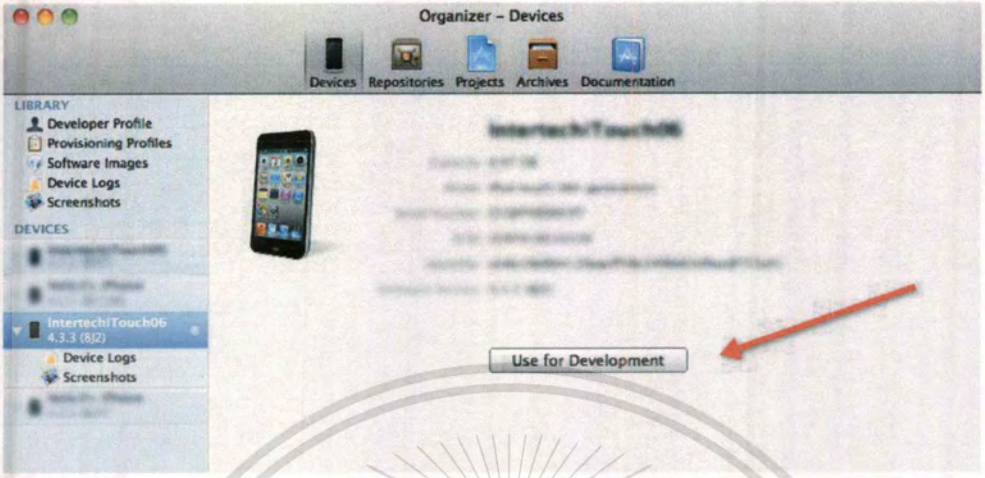


Figure C.3 Organizer screen

5. Then, it will prompt you to log in to your Apple Developer account as shown in Figure C.4

Figure C.4 Login screen

6. First the Xcode will tell that it does not recognize the version of iOS. Allow it to collect the necessary information when prompted as in Figure C.5

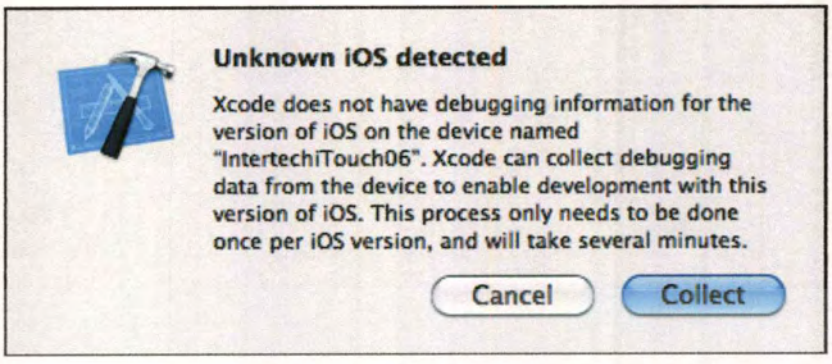


Figure C.5 Identifier screen

7. After identification of your iPad device has been complete. Your device should be listed in the top left list of Scheme in Xcode as shown in Figure C.6

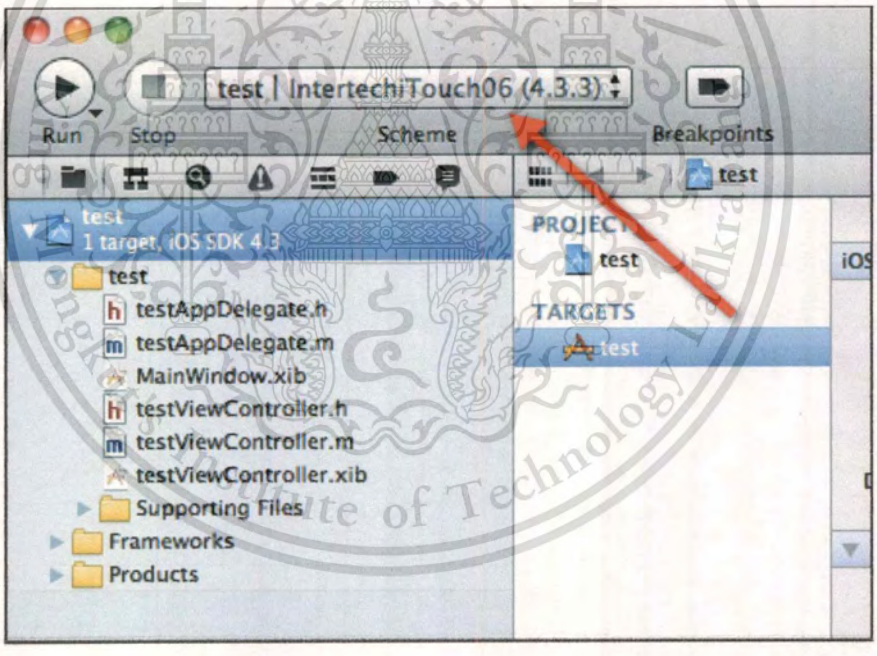


Figure C.6 Xcode workspace screen

Appendix D.

Eclipse and Android Plugins Installation

D.1 How to install Android SDK and "Eclipse"

1. Download Eclipse from http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/indigosr2 (see Figure D.1).

Figure D.1 Eclipse's downloading page

2. Download JDK from http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1637583.html (See Figure D.2).

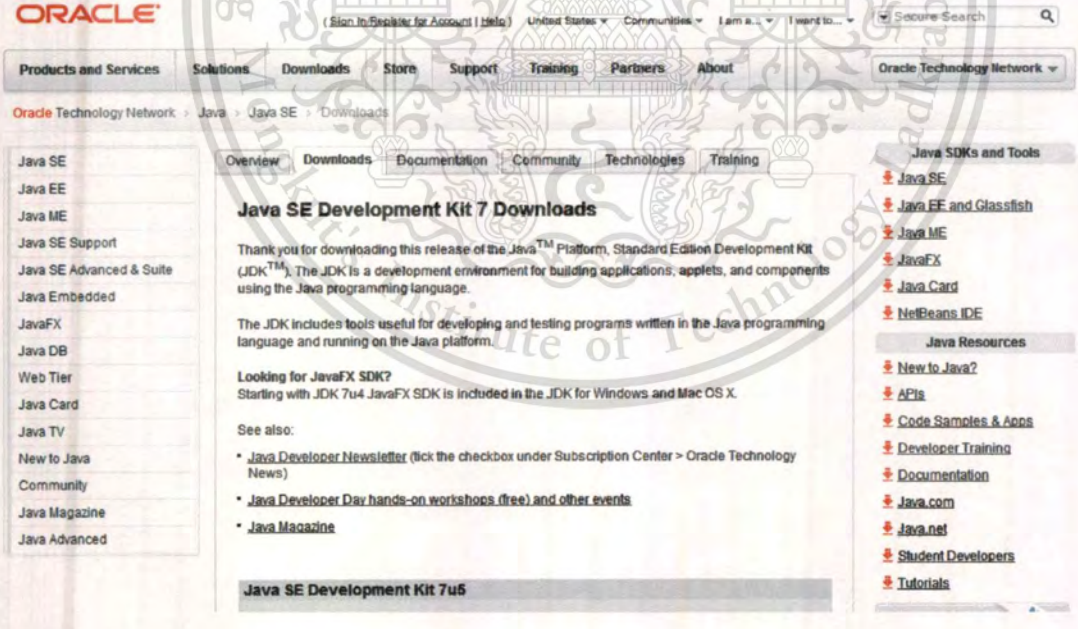


Figure D.2 JDK's downloading page

3. Download Android SDK from http://developer.android.com/sdk/index.html (see Figure D.3).

Download the Android SDK

Welcome Developers! If you are new to the Android SDK, please read the steps below, for an overview of how to set up the SDK.

If you're already using the Android SDK, you should update to the latest tools or platform using the *Android SDK and AVD Manager*, rather than downloading a new SDK starter package. See Adding SDK Components.

Platform	Package	Size	MD5 Checksum
Windows	android-sdk_r18-windows.zip	37448775 bytes	bfbfdf8b2d0fdecc2a621544d706fa98
	installer_r18-windows.exe (Recommended)	37456234 bytes	48b1fe7b431afe6b9c8a992bf75dd898
Mac OS X (intel)	android-sdk_r18-macosx.zip	33903758 bytes	8328e8a5531c9d6f6f1a0261cb97af36
Linux (i386)	android-sdk_r18-linux.tgz	29731463 bytes	6cd716d0e04624b865ffed3c25b3485c

Figure D.3 Android SDK's downloading page

4. Install ADT plug in to Eclipse

4.1 Open program Eclipse then select **Help** from its menu bar (see Figure D.4).

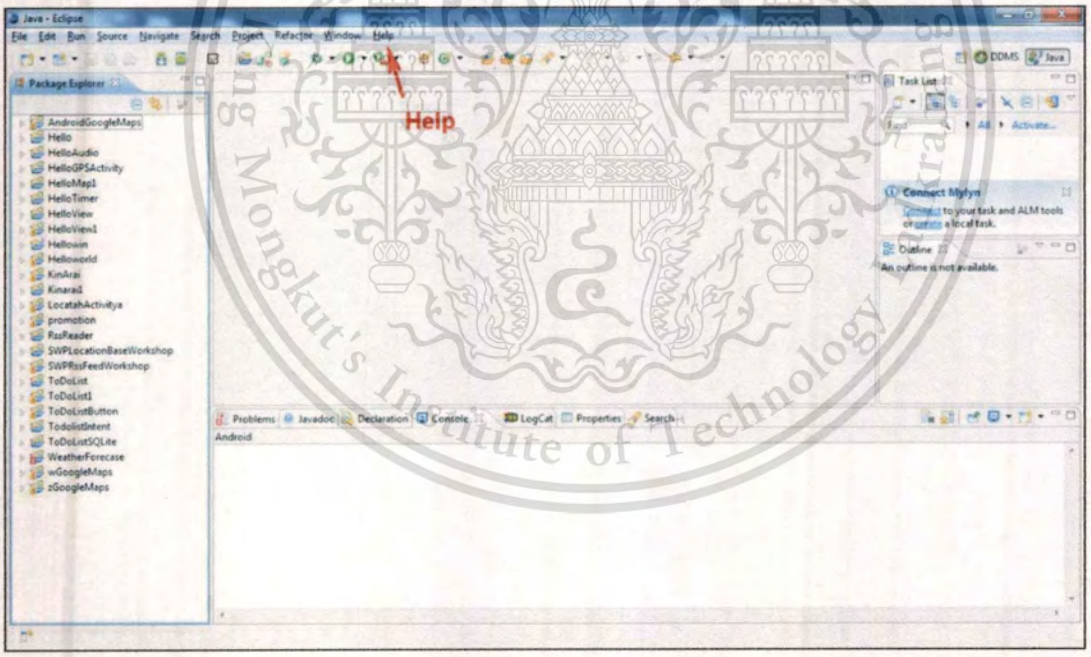


Figure D.4 Eclipse editor : Help button

4.2 Select **"Install New Software"** (see Figure D.5).

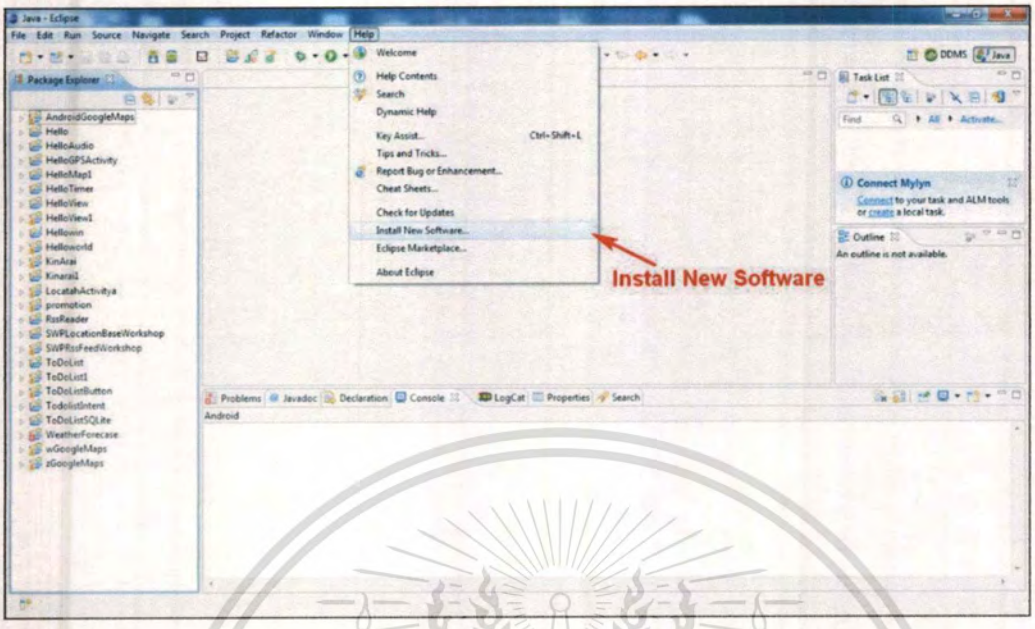


Figure D.5 Eclipse: Installation of new software

4.3 In Figure D.6, click **Add** button, the next window as shown in Figure D.7 will appear.

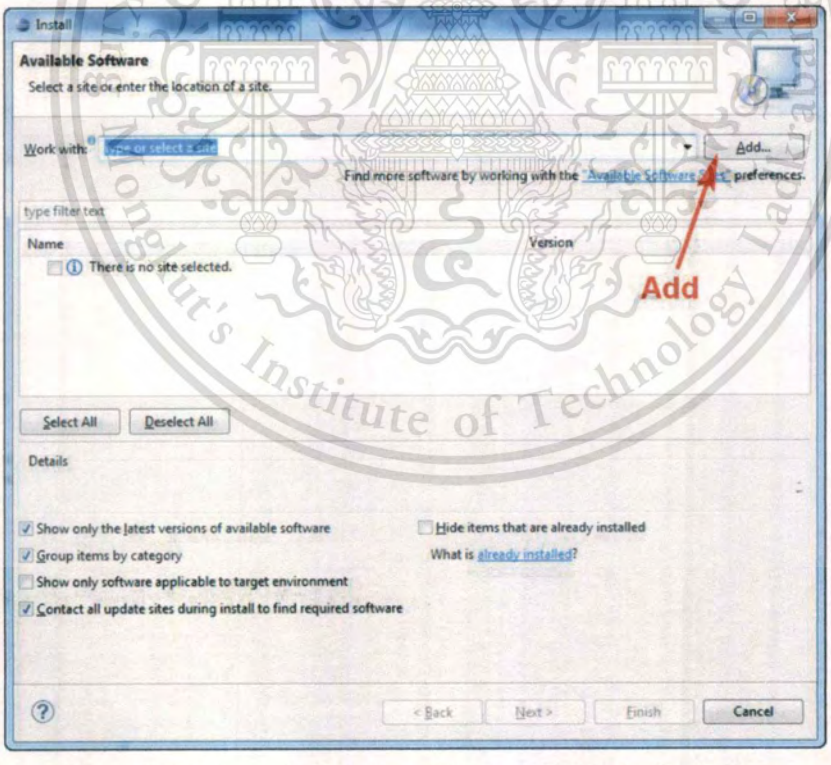


Figure D.6 Eclipse: Installation of Android SDK plugin window

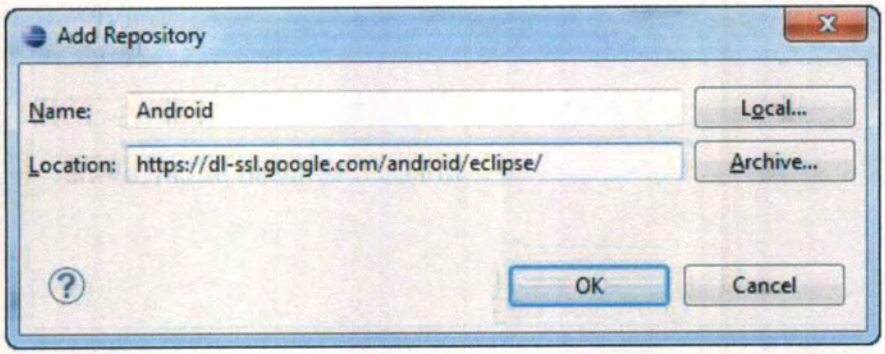


Figure D.7 Eclipse: Adding Android repository

4.4 In Figure D.7, fill in Android for name and https://dl-ssl.google.com/android/eclipse/for its location in the android repository.

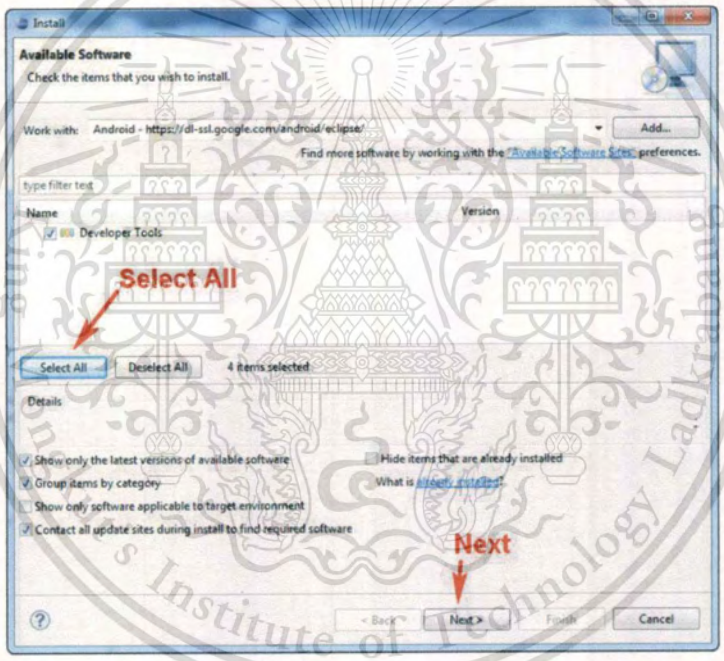


Figure D.8 Eclipse: SDK plugin selection

4.5 In Figure D.8, click at **Select All** then click at the **Next** button, the downloading process will begin.

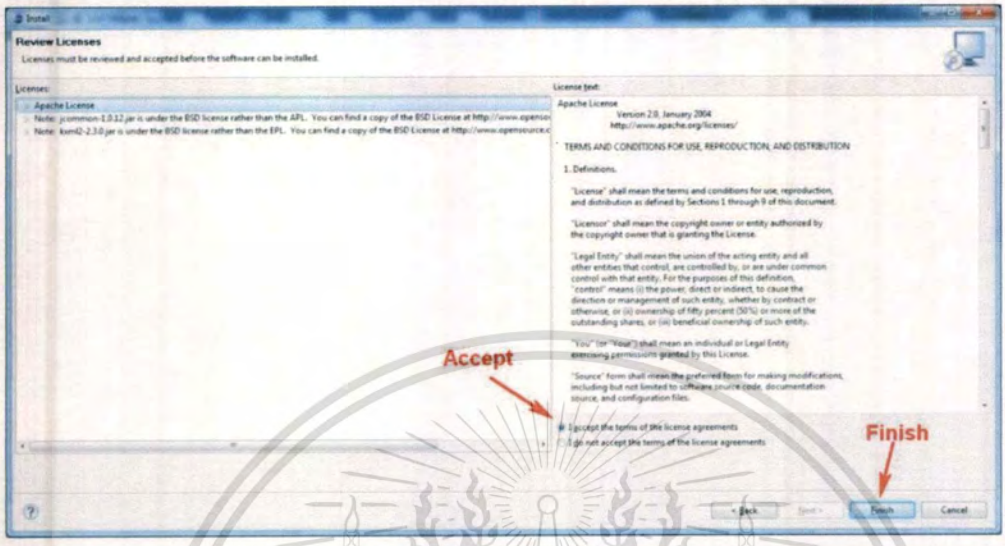


Figure D.9 Eclipse: SDK plugin's review license window

4.6 In Figure D.9, when downloading process is finished, the system will ask for the acceptance of plugins' installation. Click at **I Accept**. Then, click at **Finish**.

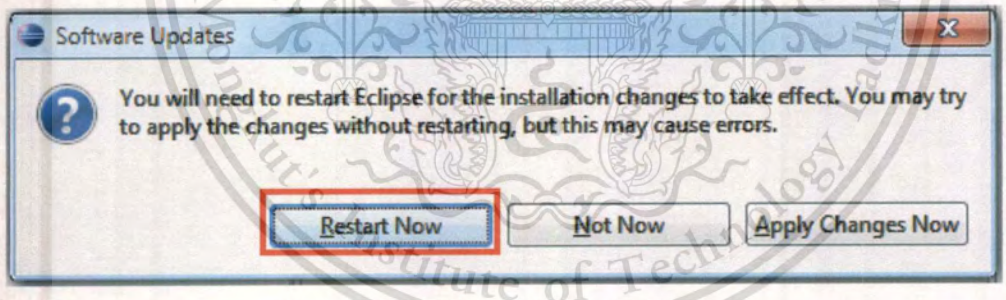


Figure D.10 Eclipse: SDK restart promp

4.7 When the system finishes installing the plugins, press **Restart Now** as shown in Figure D.10. The system will restart the program.

4.8 Open the Eclipse program and download the SDK version of android by clicking at **Android SDK Manager's icon** as shown in Figure D.11.

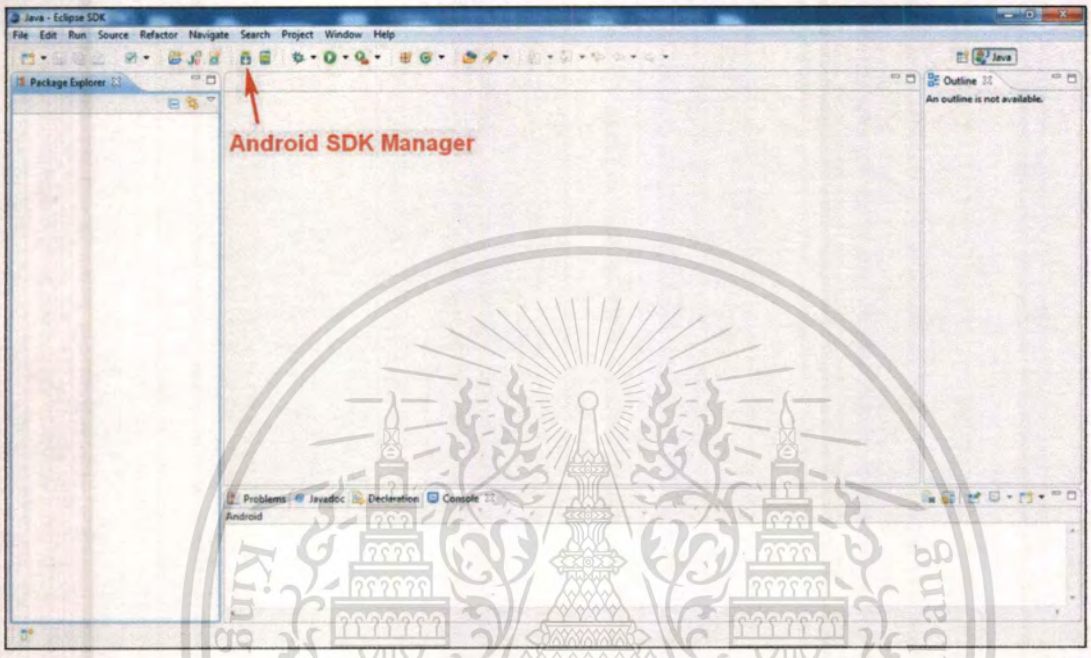


Figure D.11 Eclipse: Android SDK Manager's icon

4.9 Android SDK manager allows its developers to choose the SDK version that they want to install, which more than one version can be installed at the same time. The choice of installation version depends on the developers' applications and devices. In our program, Android SDK version 2.3.3 was chosen which there were totally 24 packages as shown in Figure D.12. Note that the developer can omit some packages by expanding at the folder icon in front of the version number and deselect those packages then, click at **Install 24 packages** button.

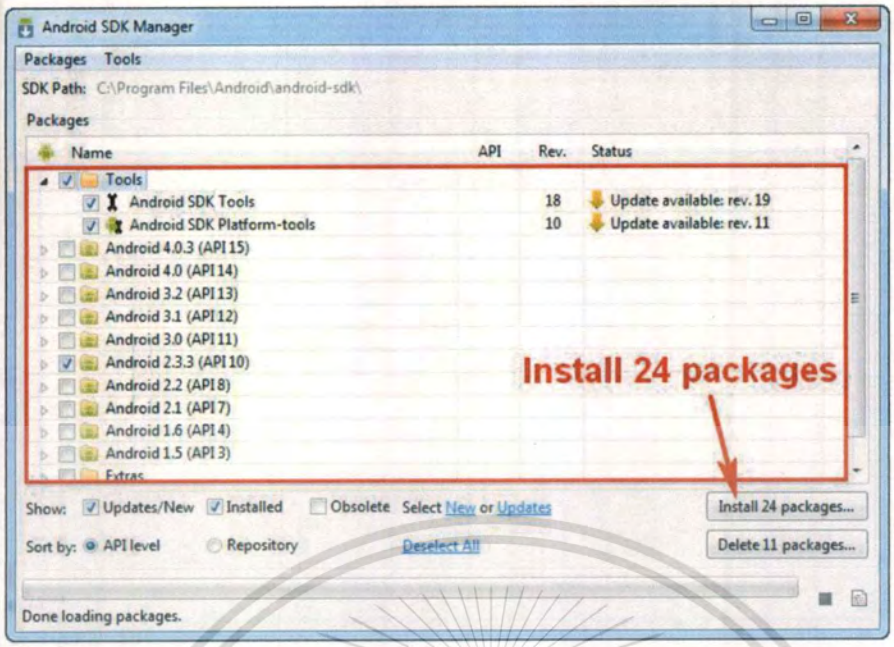


Figure D.12 Android SDK manager: version selection

4.10 In Figure D.13, select **Accept All** and click at **Install** button.

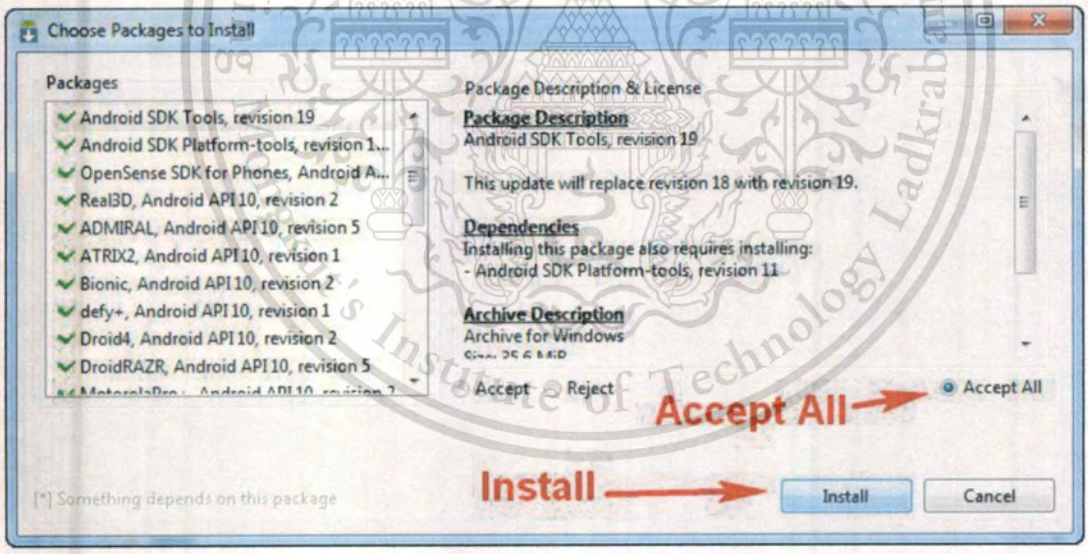


Figure D.13 Android SDK manager: Package description and license acceptance

4.11 Figure D.14 shows the installation of android SDK which when the installation is finished, the program will ask to restart ADB (Android Debug Bridge). Click **Yes** to restart as shown in Figure D.15.

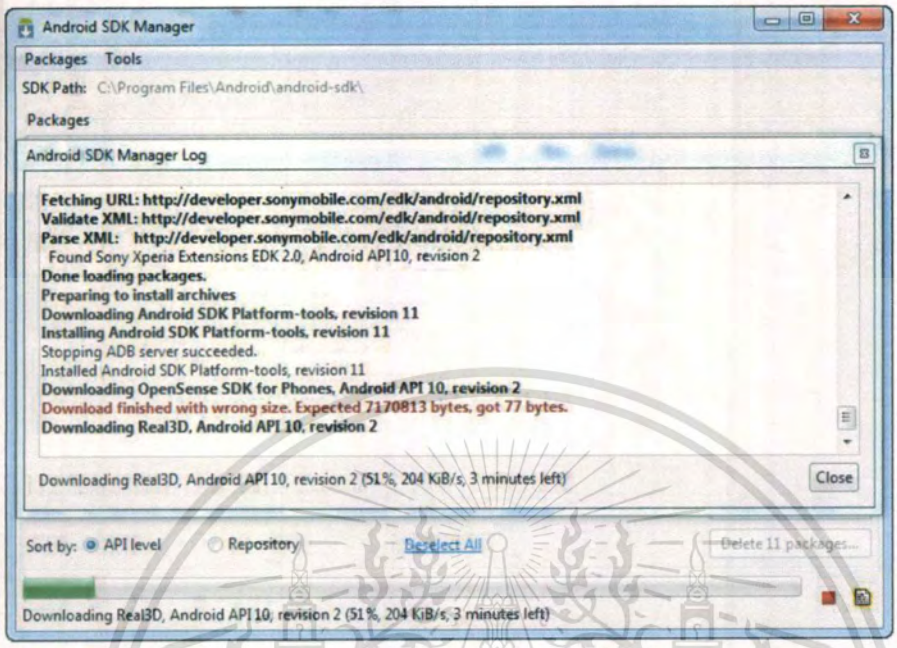


Figure D.14 Android SDK Manager: Installation of android SDK

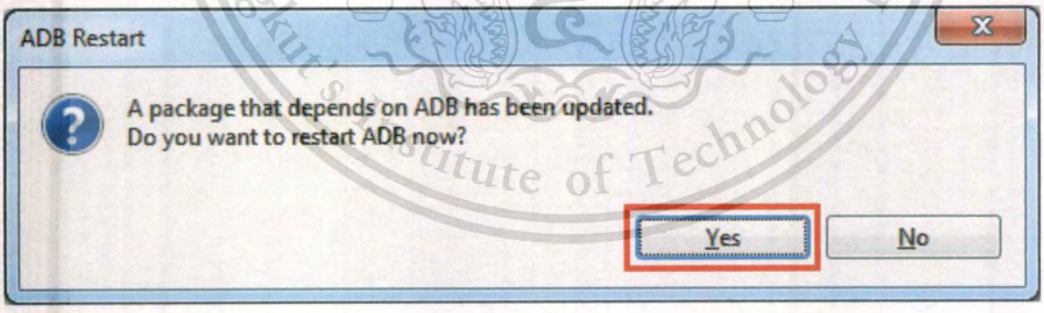


Figure D.15 Android SDK Manager: Restart ADB

4.12 Figure D.16 shows the Android SDK manager log after the installation is successful. Click **at Close** button to prepare for next step.

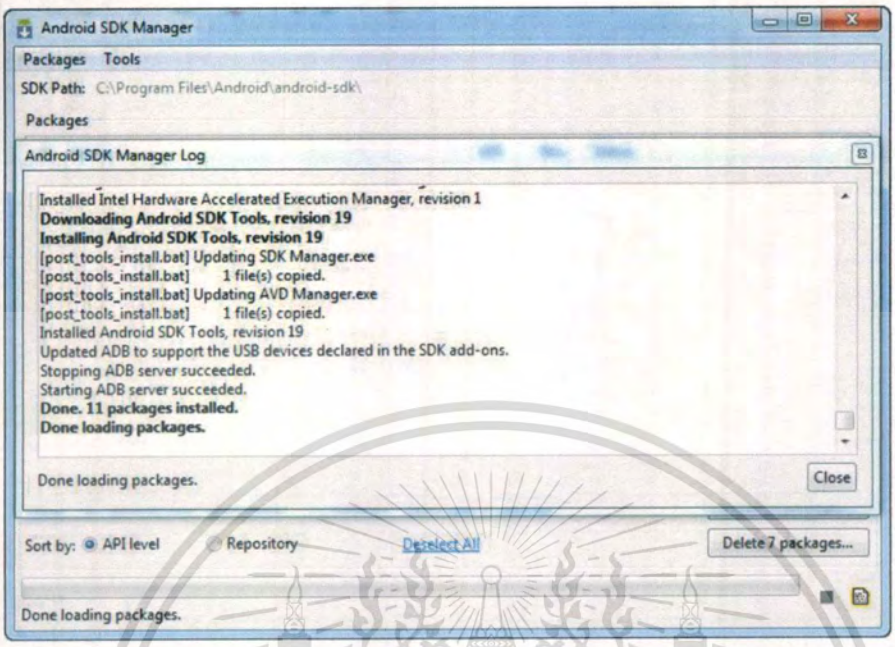


Figure D.16 Android SDK Manager: Android SDK Manager Log

5. Start the android emulator by pressing **File → New → Other** as shown in Figure D.17.

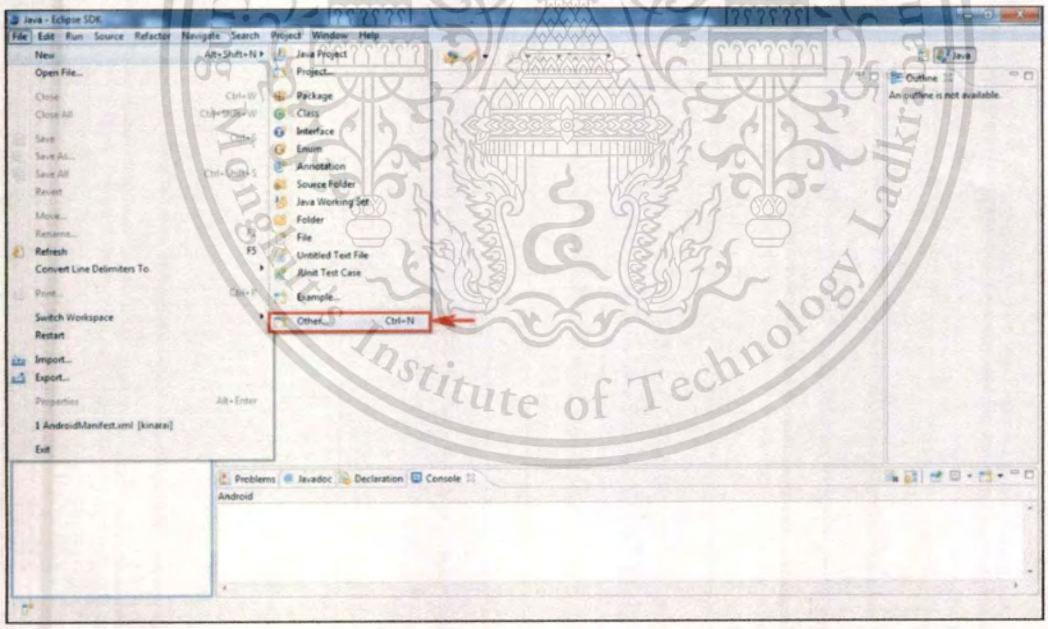


Figure D.17 Eclipse: Starting the android emulator

5.1 To create an android project, click at **Android** → **Android Project** → **Next button** as shown in Figure D.18.

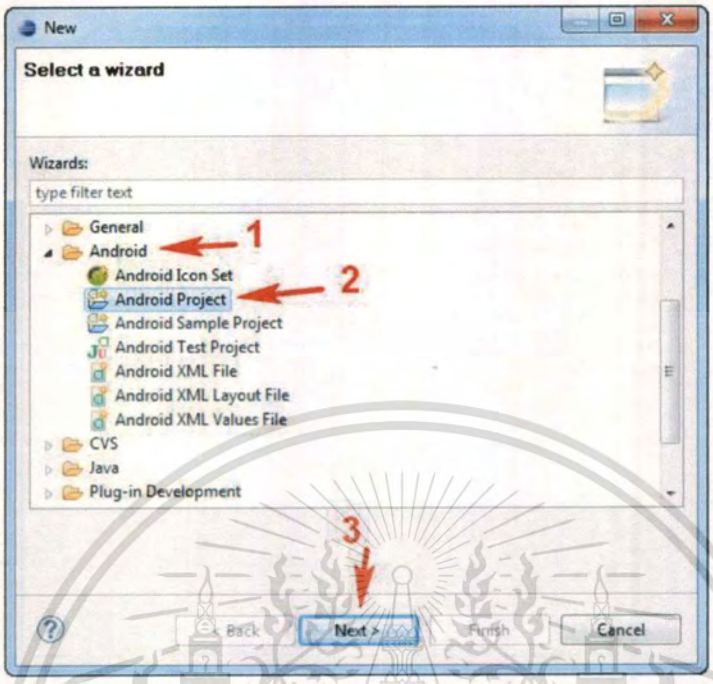


Figure D.18 Eclipse: Creating an android project

5.2 As seen in Figure D.19, type your project name and click at **Next** button.

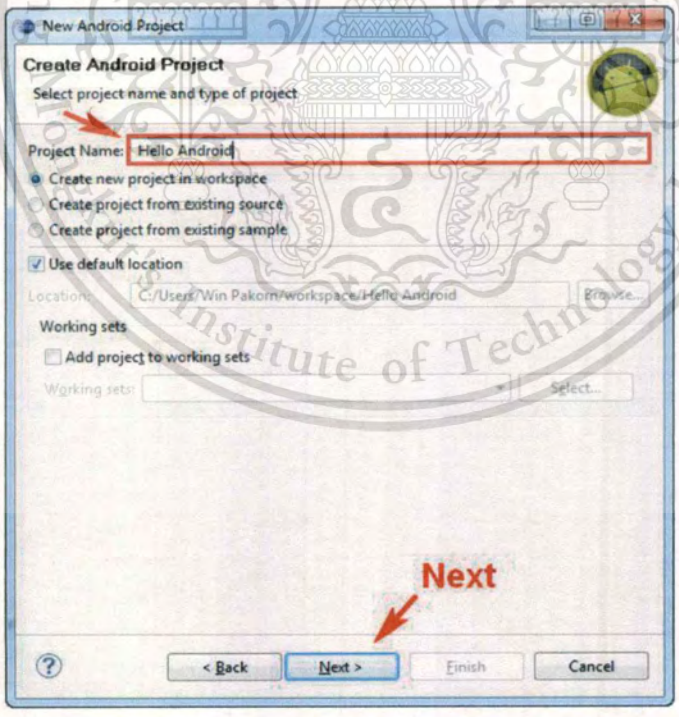


Figure D.19 Eclipse: New Android Project

5.3 Then, checking at your target version and click **Next** button as shown in Figure D.20.

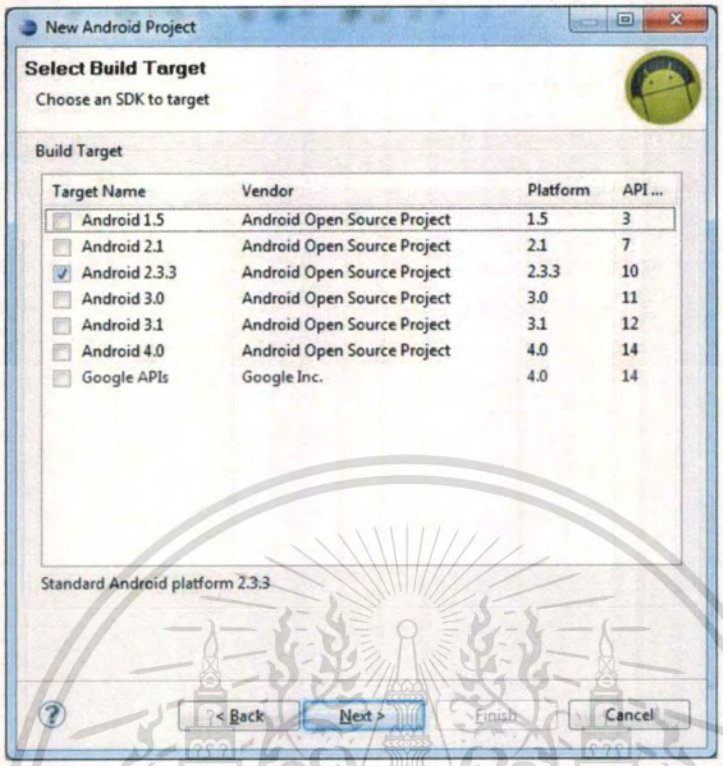


Figure D.20 Selection of android version

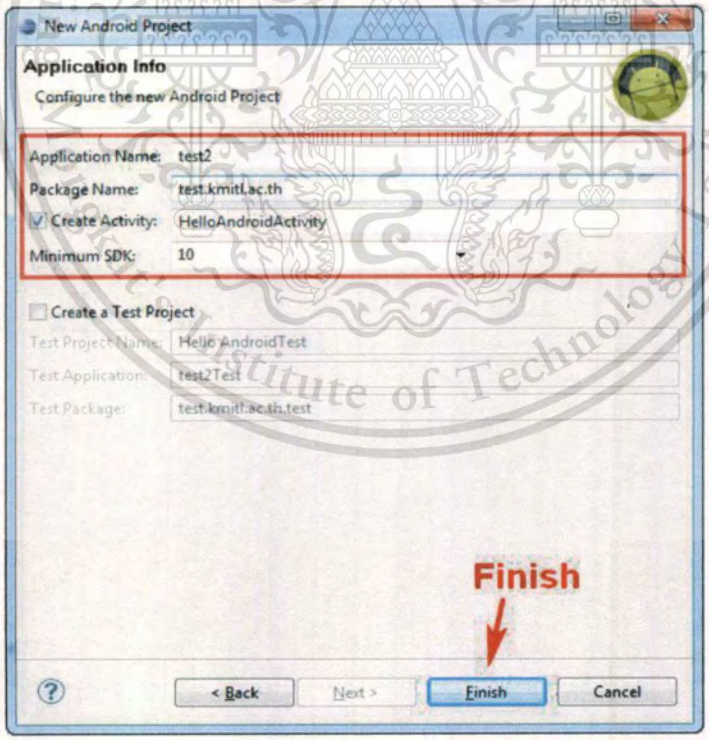


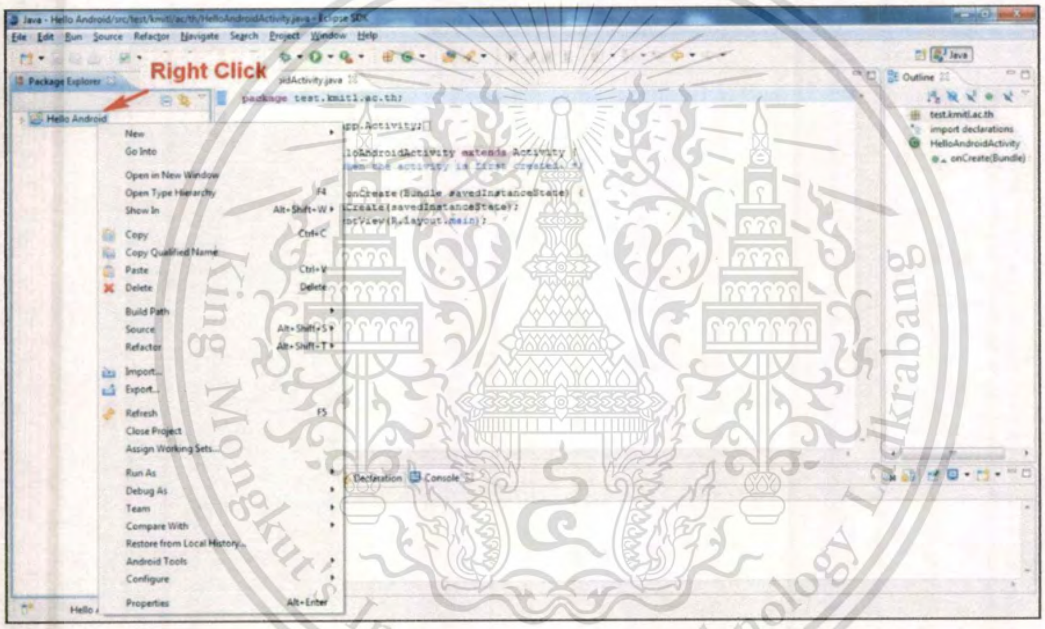
Figure D.21 Android project application information setting

5.4 An application information is configured in Figure D.21. You, as a developer, are required to fill the **application name** *(test2)* and the **package name** *(test.kmitl.ac.th)*. **Create Activity** box is checked and the activity name is derived from the project name (*HelloAndroidActivity*), which is the project name followed by the word "Activity". A value next to **Minimum SDK** indicates the lowest Android SDK version that the project can run. When the configuration is done, click **Finish** button and the window as shown in Figure D.22 will appear. Now the program test2 can be compiled and seen in the emulator.

D.2 Steps to Compile and Run Android Project

Steps to compile and run the android projects are listed as follows:

1. **Right click** at the android project **Hello Android** as shown in Figure D.22.



FigureD.22 Running a program on the emulator: Step 1

2. Select **Run As** → **Run Configurations** as shown in Figure D.23.

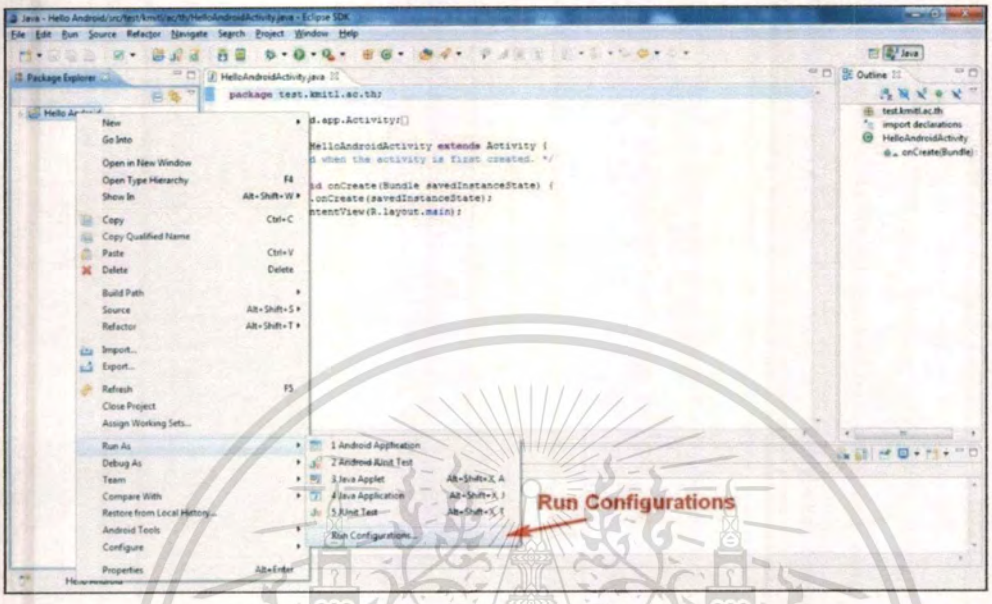


Figure D.23 Running a program on the emulator: Step 2

3. In Figure D.24, **double click** at **Android Application.** The **New Configuration,** and the Android tab will appear as shown in Figure D.25.

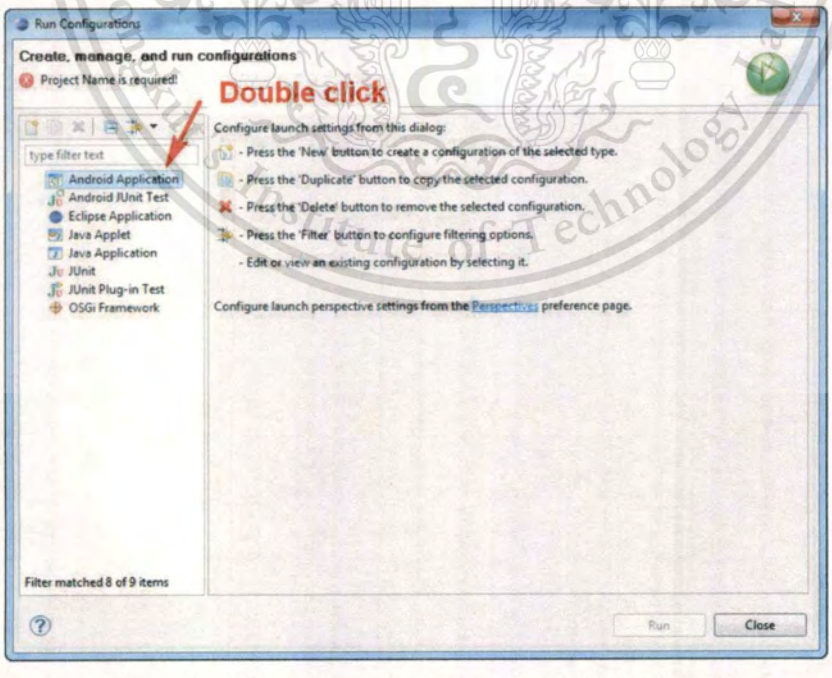


Figure D.24 Running a program on the emulator: Step 3

4. Next, enter the name of the project in the box next to **Name:** *(Test)* . Then, click at **Browse** button to select your project *(HelloAndroid)*.

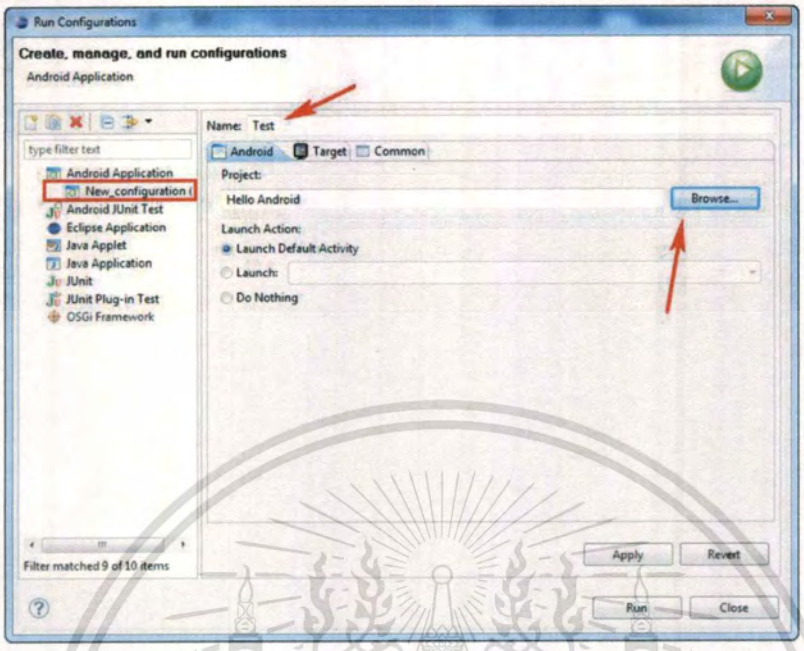


Figure D.25 Running a program on the emulator: Step 4

5. Click at **Target tab** as shown in Figure D.26. Then, tick box in front of your chosen Android Virtual Device *under the* **AVD Name.** *Next, click at* **Apply** *button and* **Run** *button.*

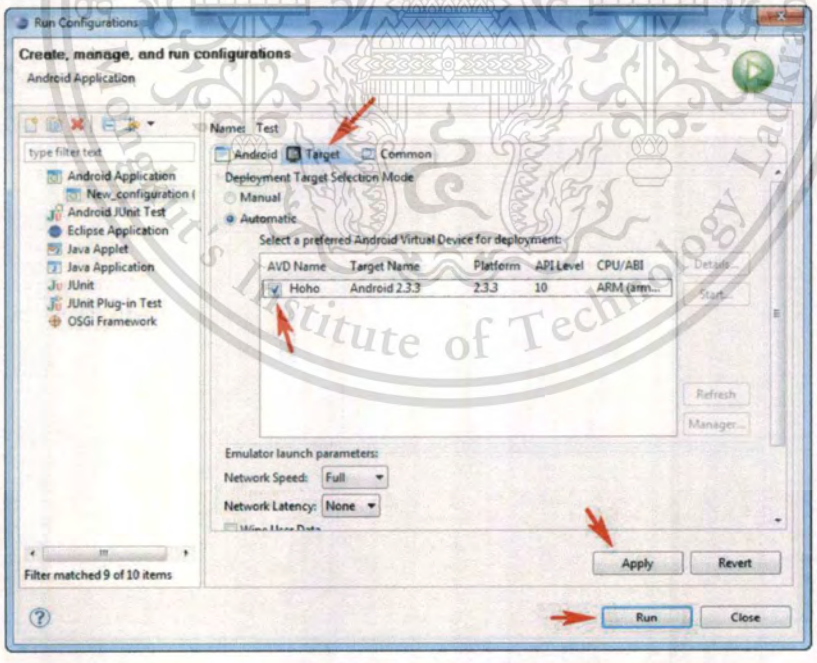


Figure D.26 Running a program on the emulator: Step 5

6. After clicking **Run** button in Figure D.26, the emulator will show and load the project as shown in Figure D.27.

Figure D.27 Running a program on the emulator: Step 6

7. Figure D.28 shows the result of the project that are built.

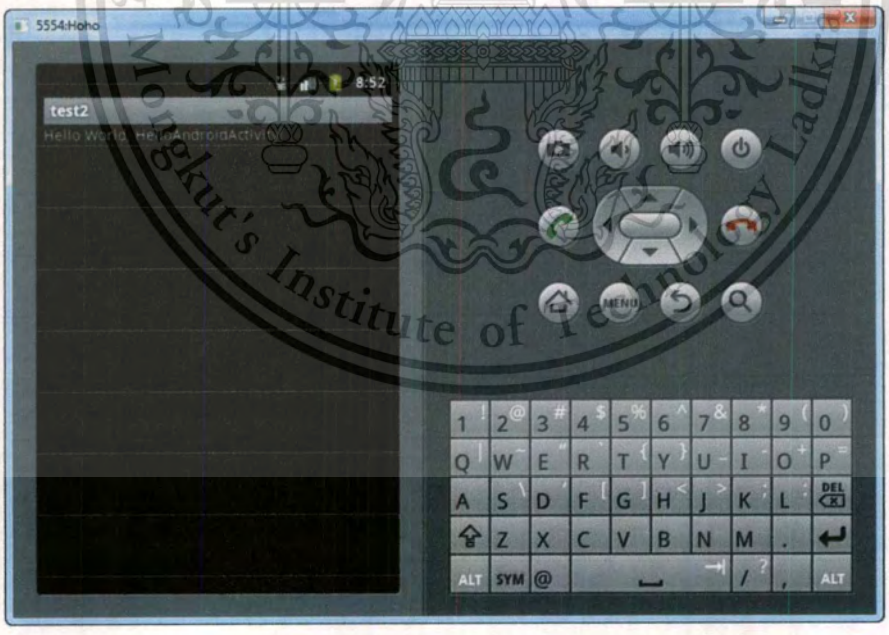


Figure D.28 Running a program on the emulator: Step 7

Appendix E

How to export to the APK files

Export to the APK file.

When the users want to install our Nyorion application on their android mobiles, they need .apk file. The instructions to export the application from Eclipse SDK to the APK File are listed below.

1. Right click at the android project that you want to export under the tap **Project Explorer**.

2. Go to **Android Tools** and click at **Export Signed Application Package** (as shown in Figure E.1). The screen as in Figure E.2 will appear.

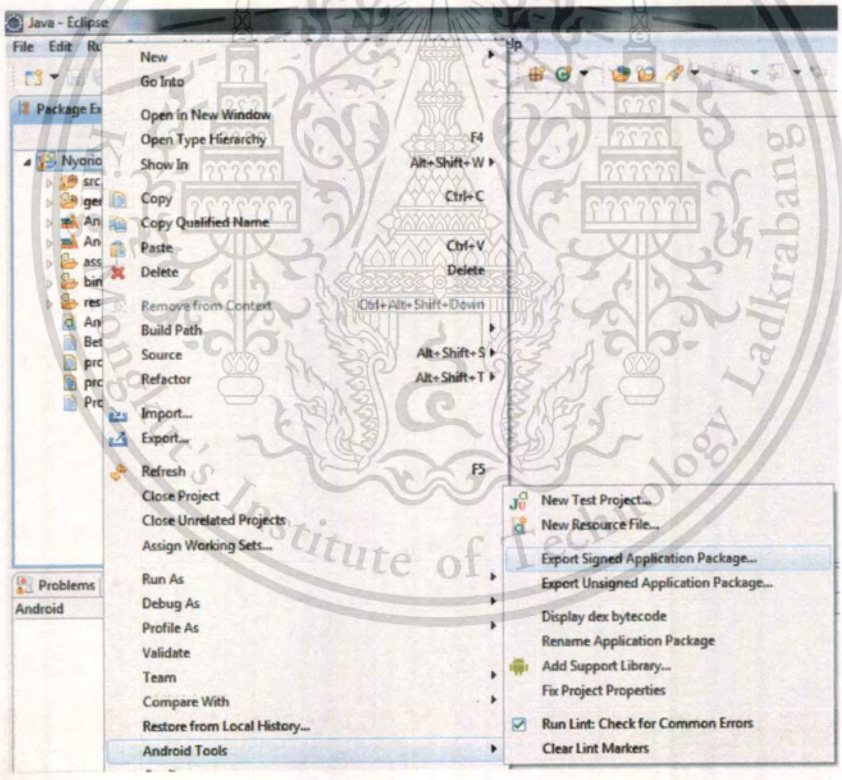


Figure E.1 Exporting android project to an APK file

3. **Browse** to the project name that you want to export and click. Then, click **Next.**

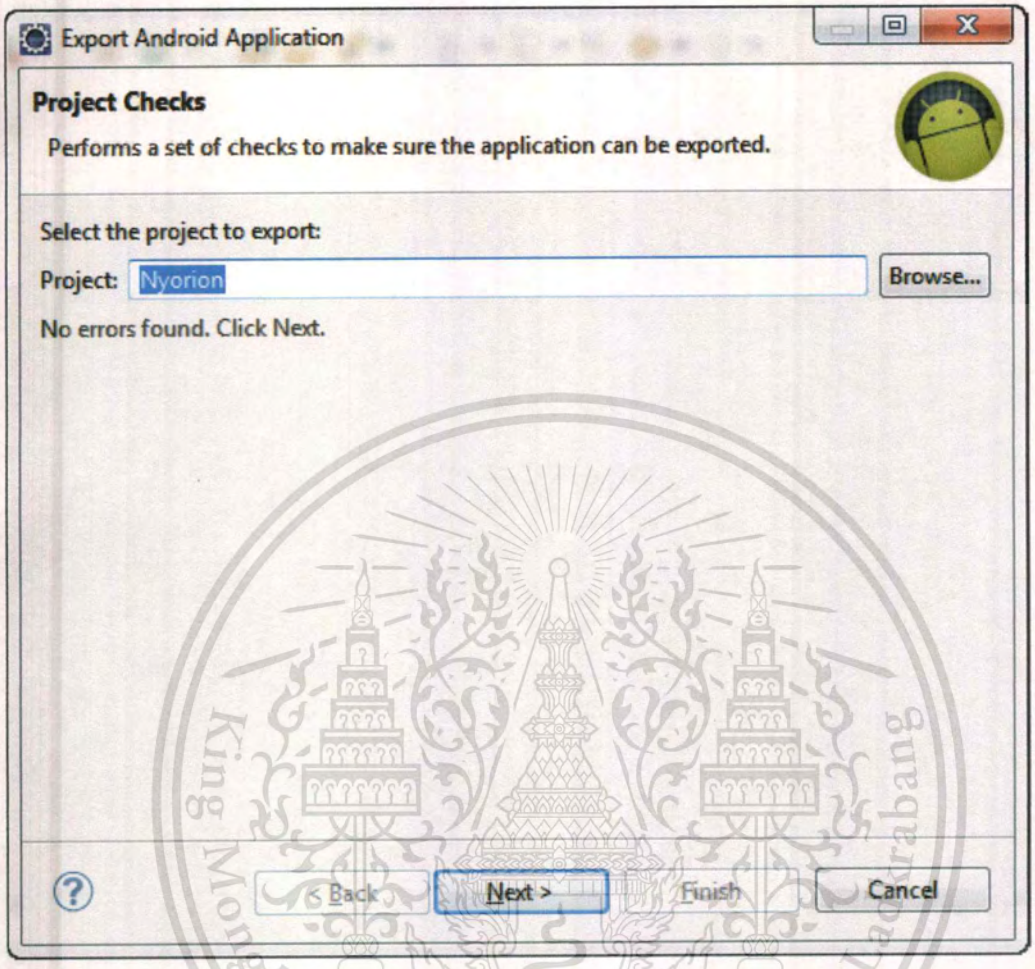


Figure E.2 Selecting the project

4. It will ask for the keystore. If the user already has set up the keystore, select **Use existing keystore** otherwise select **Create new keystore.** Then, type the password in both password and confirm textboxes. For **Use existing keystore**, enter the keystore file name or click **Browse** to browse for the existing keystore file as shown in Figure E.3.For **Create new keystore**, click Browse for the location that you want to save the keystore or otherwise use the default location which it is C/Users/Jack/Meesith as shown in Figure E.4. Then, click **Next.**

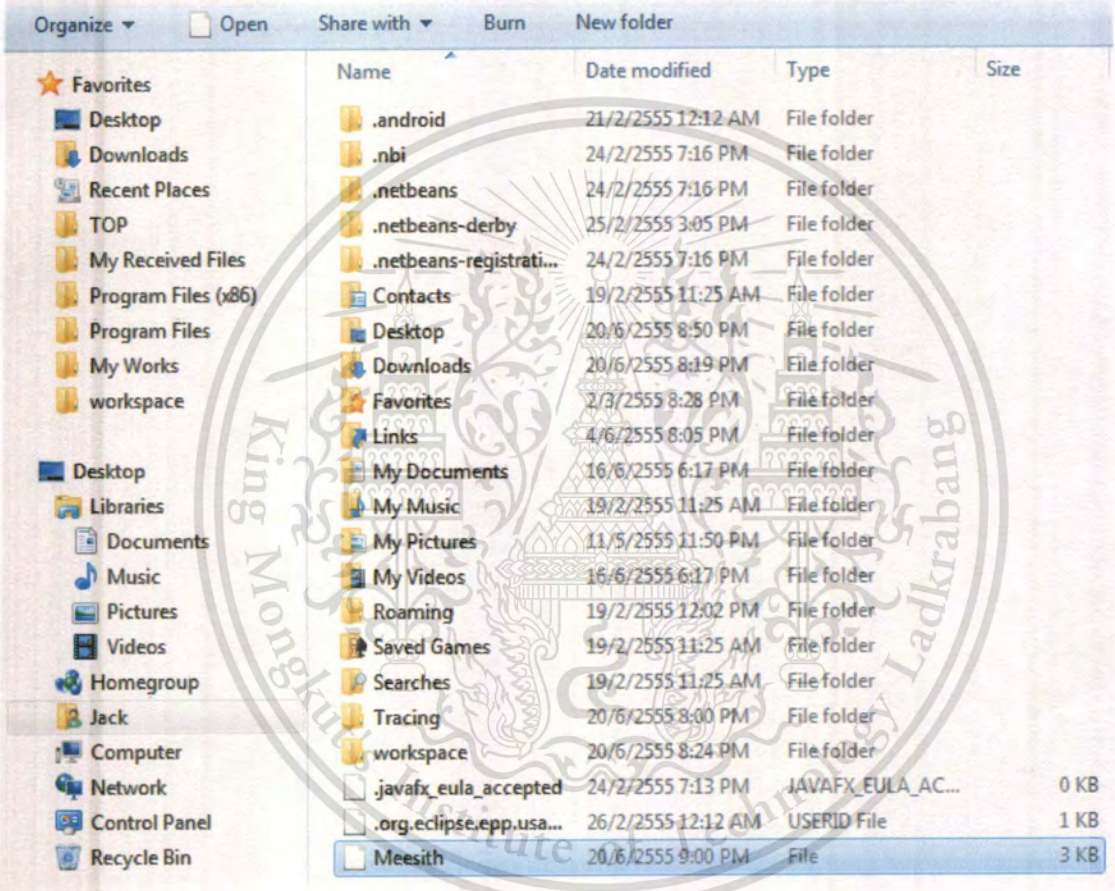


Figure E.3 Keystore selection (1)

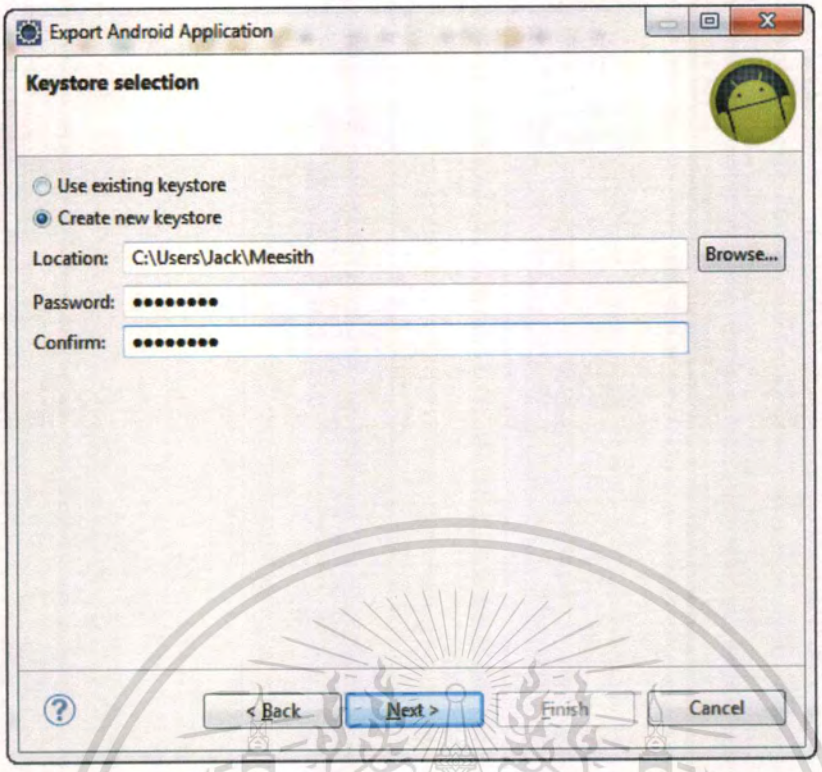


Figure E.4 Keystore selection (2)

5. It will ask for the key alias which is the alias of the developer who develops this application. Since we already have it so we create the new one as an example (see Figure E.5). Then, click **Next.**

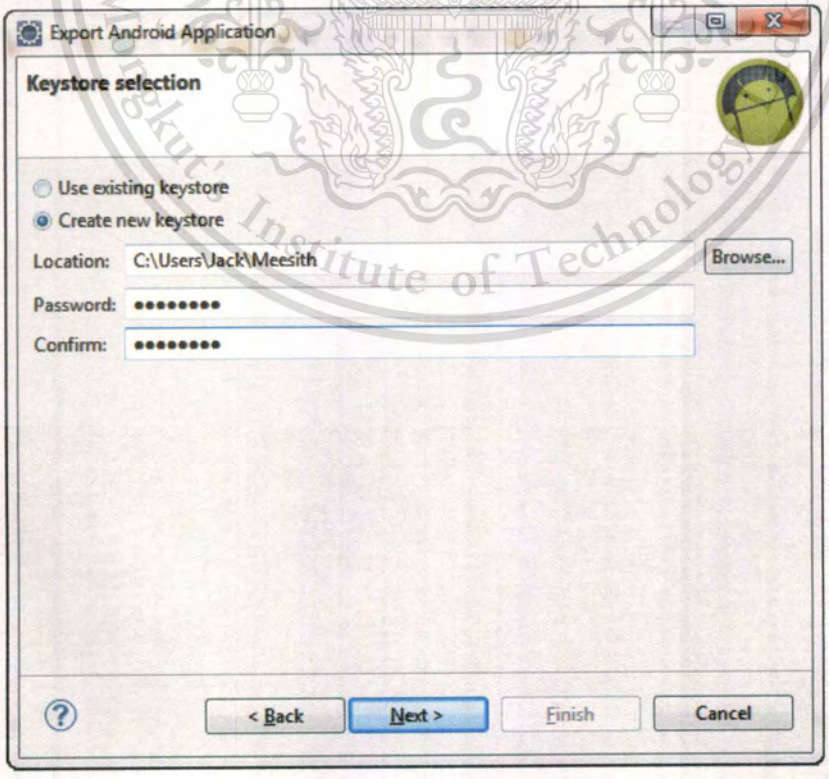


Figure E.5 Asking for key alias

6. The screen as in Figure E.6 which asks for the developer information will appear. Click **Next** after you fill all required information.

Figure E.6 Creating the key alias

7. Enter or browse for the destination path to export the file to. Then, click **Finish** (see Figure E.7).

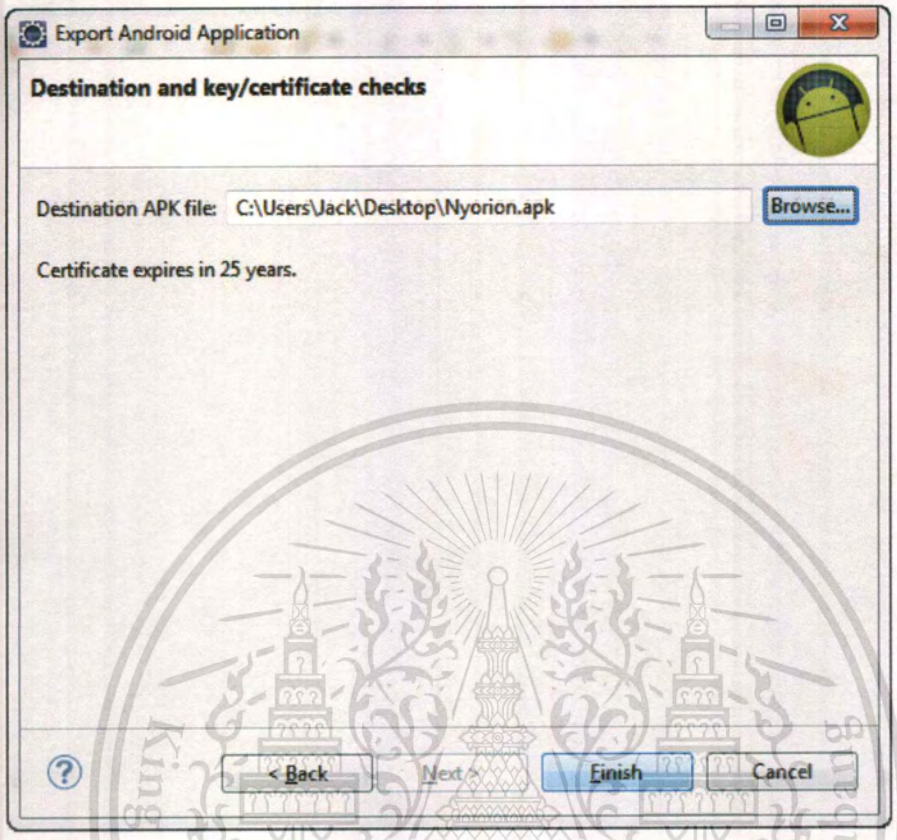


Figure E.7 Selecting the destination path