Supporting

Distributed Multiplayer RoboTable Games

A thesis

submitted in partial fulfilment

of the requirements for the Degree of

Master of Software and Information Technology

at

Lincoln University

by

Leshi Chen

Lincoln University

2012

Abstract of a thesis submitted in partial fulfilment of the requirements for the Degree of Master of Software and Information Technology.

Supporting Distributed Multiplayer RoboTable Games

by

Leshi Chen

Lincoln University is cooperating with Tufts University, USA, on the development of a RoboTable to facilitate interaction between groups learning about robotics and engineering problem solving. A RoboTable is a mixed reality tabletop learning environment and provides a distributed learning platform with groups of children at remotely located tables interacting and competing on robotic projects.

This project investigates the development and support of distributed multiplayer games using the RoboTable environment. Currently, there are no general-purpose tools to support RoboTable game development or distributed game play. Hence a gap exists to develop a robust solution that will allow distributed multiplayer games to be created and played using RoboTable.

To address these issues, we have implemented a set of toolkits to support distributed multiplayer RoboTable game development. The toolkits comprise a Network Toolkit, a Robot Tracking Toolkit, a Game Management Toolkit and a Communication Toolkit. In addition, we have developed a skeleton project to help the game developers.

To evaluate the toolkits, we have used a number of approaches. The first approach was a case study of the game development process using the toolkits. The second approach was to establish baseline performance benchmarks for the system. The third approach was to carry out experiments to evaluate their real-world performance and the scalability of the toolkits using the game created in the case study. The results from the experiments have shown that the toolkits perform well within a distributed computer environment. The results from the case study have revealed that the development of a new RoboTable game is straightforward.

Keywords: RoboTable Game, Multiplayer RoboTable Game, HCI, Multiplayer Game, Robotics, Toolkits, Game Engine, Distributed Network, Educational Game.

Acknowledgements

Firstly, I would like to acknowledge my supervisor team, Stuart Charters, Keith Unsworth and Alan McKinnon, for their support and direction. They all made significant effort to provide comprehensive comments and suggestions for this research. I appreciate their hard work.

Secondly, I would like to express my thanks to Mark Anderson, for his help with the computer hardware.

Thirdly, I would like to express my particular thanks to my wife Yuanmiao (Karen) Wang and my new baby Zilong (Jamie) Chen. Without their support and their kindly letting me to spend a lot of time on my studies, I could not have completed this project. Also, I would like to thank to my family, and my family-in-law. Without their financial support, I would not have had the chance to study for a Masters Degree.

Finally, I would like to thank all who have supported me for this project.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Lincoln University is cooperating with Tufts University, USA, on the development of a RoboTable to facilitate interaction between groups learning about robotics and engineering problem solving. Currently the RoboTable comprises a semi-transparent top that can display images projected from a projector under the table, a Wiimote controller that it is used for robot tracking and a PC that controls the display, tracking and game play.

The RoboTable is a mixed reality tabletop learning environment that can provide a powerful engineering learning experience to school children (Mason, 2005). Research has confirmed that having groups of school children compete with each other through playing games can increase their motivation in study (Facer, 2003; Oblinger, 2006; Prensky, 2001). Therefore, RoboTable can provide a useful distributed learning platform with groups of children at remotely located tables interacting and competing on robotics challenges.

Although Mason (2005) developed a demonstration game for the RoboTable, there are no general-purpose tools to support RoboTable game development or distributed game play. Hence a gap exists to develop a robust solution that will allow distributed multiplayer games to be created and played using RoboTable.

1.1 Goal and Scope

The goal of this research is to investigate how to support the implementation of distributed multiplayer games using the RoboTable environment. The objective is to support RoboTable game developers, so they do not have to implement multiplayer functionality from scratch for each new game. This thesis describes the development of a set of toolkits to support the development of distributed multiplayer RoboTable games as the solution to achieve this goal.

1.2 Structure of the Thesis

Chapter 2 provides background about the use of games for learning, the development of the RoboTable, network communication technologies, multiplayer game architectures and the problem definition.

Chapter 3 reviews the literature on existing game architectures, network toolkits, multiplayer game state management, and design patterns.

Chapter 4 discusses user requirements and presents the proposed solution to match the user requirements. It also discusses the idea of using toolkits to simplify the RoboTable game development process and introduces the concept of combining different toolkits to provide generic interfaces to RoboTable game developers.

Chapter 5 discusses the implementation of the toolkits, and describes in detail the process of the development of the toolkits.

Chapter 6 evaluates the toolkits that have been developed. It describes the evaluation methods and discusses the results.

Chapter 7 concludes the research and discusses possible future work.

Appendix A provides an interview question list.

Appendix B presents a game developer guide.

Appendix C presents a developer manual.

Appendix D outlines debugging and error logging tool support.

Chapter 2
Background

The following sections introduce background information on games for learning, RoboTable development, network communication technology and game architecture. Subsequently, the problem definition of this project will be discussed.

2.1 Games for Learning

How do students in the 21st century learn? Computers have already been part of normal life for everyone especially students and adolescents (Oblinger, 2004). Oblinger (2004) described the students who were born in or after 1982 as NetGen'ers (Net Generation people). NetGen'ers are more likely to enjoy group activities, believe being smart is cool and are attracted to new technologies. The learning preferences of NetGen'ers tend towards active learning rather than passive learning. NetGen'ers often spend most of their free time surfing the Internet, using instant messaging communications, playing video games, reading the latest news and studying new technologies on the web. "Today's students are digitally literate. Whether 18 or 48, virtually all learners are accustomed to operating in a digital environment for communication, information gathering and analysis" (Oblinger, 2004).

Playing computer games is already a significant part of the culture of young people. Oblinger and Prensky (Oblinger, 2004; Prensky, 2001) found that net generation students were likely to spend 10,000 hours playing video games before they reached 21 years of age. "Games are now part of modern culture" (Oblinger, 2006). Thomas Malone cited by Facer (2003) argues that the essential features that contribute to the motivation to play games were "challenge", "fantasy" and "curiosity".

Prensky (2001) encapsulates the key elements of games as:

- A combination of rules that a player can follow.
- A number of goals or objectives that players can achieve.
- Different outcomes that a player can discover.
- Challenging situations that a player can enjoy.
- Interaction and fascinating stories that a player can play.

Games can play a crucial role in learning and computer games have already changed the study abilities of students and expectations about school learning (Facer, 2003; Prensky, 2001). With regard to how people learn using games, Oblinger (2006) summarizes the key attributes as:

- **Social**. Games are often social environments, and some of them involve large networked communities.
- **Research**. A new player must learn new skills in order to master the game, and thus improve the researching skills.
- **Problem solving**. Knowing more information and techniques means greater success.
- **Transfer of Learning**. Games can require players to transfer learning from other environments to a unique game situation.

Recent research has confirmed that games are effective ways of learning (Isbister, Flanagan, & Hash, 2010; Klopfer, Osterweil, & Salen, 2009; Park & Park, 2010; Stansbury, 2009). "Games, it seems, 'have something', they seem to have a way of engaging and interesting young people. The desire to harness this motivational power to encourage young people to want to learn is the main driver behind and interest in computer games for learning" (Facer, 2003). Prensky (2001) claims that those students who have regular and intensive game play, can develop the ability to process information remarkably quickly as well as more readily estimate what information is and is not of significance to them. However, Prensky also pointed out that not all students prefer games as a way to learn, and the relationships students have with games vary widely. Girls may not like the types of games that boys do. To address these issues, Papastergiou (2009) conducted a study that compares the Digital Game–Based Learning (DGBL) approach in high school computer science with the non-gaming approach and found that DGBL was more effective and motivational. Papastergiou also found that DGBL can be equally effective and motivational for boys and girls. He concluded that DGBL can promote curriculum knowledge and student motivation in core academic subjects of High School computer science.

There are a number of education centres paying attention to game development for learning, especially in engineering education. The Centre for Engineering Educational Outreach (CEEO) at Tufts University in Boston (http://www.ceeo.tufts.edu) is developing expertise in this area.

The CEEO believes engineering involves many skills that are of practical value to all citizens in handling challenges of daily life because a basic education in engineering can help students to learn by:

- **Thinking** of a solution creatively and independently
- **Designing and testing** a solution by using his/her own hands
- **Learning from failure** of a solution to ultimately succeed
- **Communicating** with others effectively
- **Demonstrating that multiple solutions** can be used to solve a problem

Thus, the CEEO provides workshops, conferences, graduate, and undergraduate programmes for students and educators to improve teaching in science, technology and mathematics. Using engineering as the learning medium encourages students to use their pre-learned knowledge to solve problems. The students can design a solution, play with it, study the immediate results and then redesign the solution.

RoboTable, originally developed by CEEO and Lincoln University, Canterbury, New Zealand, is an educational game environment designed to encourage interest in engineering education through the use of LEGO™ robotics as a game tool.

2.2 RoboTable

In this section, we provide an overview of the development of the RoboTable project. Furthermore we describe the components that are used to set up a RoboTable, and finally review the current RoboTable tracking technologies.

2.2.1 Overview

RoboTable is a mixed reality tabletop learning environment that can provide a hands-on engineering learning experience. It supports interactions between groups of people through LEGO™-based robotics in order to enhance engineering learning (Mason, 2005). The original prototype behind the RoboTable is HI-SPACE[1]. HI-SPACE (Cowell, May, & Cramer, 2004) applies knowledge from many areas of research from Psychology to Computer Science, and it combines physical and digital information to enhance the ability to solve complex problems. HI-SPACE forms a 3D interaction volume by using a sensor array and a 2D display surface. The 3D interaction volume is the interaction space for the user. Figure 2-1 shows a prototype of HI-SPACE.

[1] Human Interface Technology Laboratory, (http://www.hitl.washington.edu)

http://www.hitl.washington.edu/projects/hispace/hispace_files/image003.jpg

Figure 2-1 HI-SPACE (HITLab, 2003)

Early research conducted by Pattie (2004) from Lincoln University and Mason (2005) from Tufts University established the basis of the current study of the RoboTable. Pattie developed an optical tracking algorithm using the ARToolkit[2]. This is discussed further in section 2.2.3. Mason discusses the development of RoboTable and the general integration of RoboTable software in his thesis (Mason, 2005). Further research conducted by Festing enhanced the optical tracking using infrared techniques (Festing, 2005). Based on the ideas of Festing, Graham Smart[3] from Lincoln University implemented a Wiimote infrared tracking system. Ben Gemmill cited by Mason (2005) developed a custom Internet server to create a physically controlled, online, persistent 3D world and a custom Internet server, which could be used to connect remote RoboTables. However, the custom Internet server was mainly designed for collaborating on tasks such as a whiteboard, and it was a non-specific design for the RoboTable, in that it contained a number of features that were not necessary for RoboTable environments. Zhao also designed a framework for distributed multiplayer RoboTable games (Zhao, 2008), which is discussed further in section 3.1.

The following sections describe the tool requirements for RoboTable, the robotics we have used for evaluation and the tracking technologies developed by researchers.

[2] ARToolkit, A software library for building Augment Reality (AR) applications, (http://www.hitl.washington.edu/artoolkit/)
[3] Graham Smart, a graduate of Lincoln University, developed the current Wiimote RoboTable Tracking system, 2010-2011

2.2.2 Hardware

The physical set-up of RoboTable at Lincoln University is pictured in Figure 2-2.

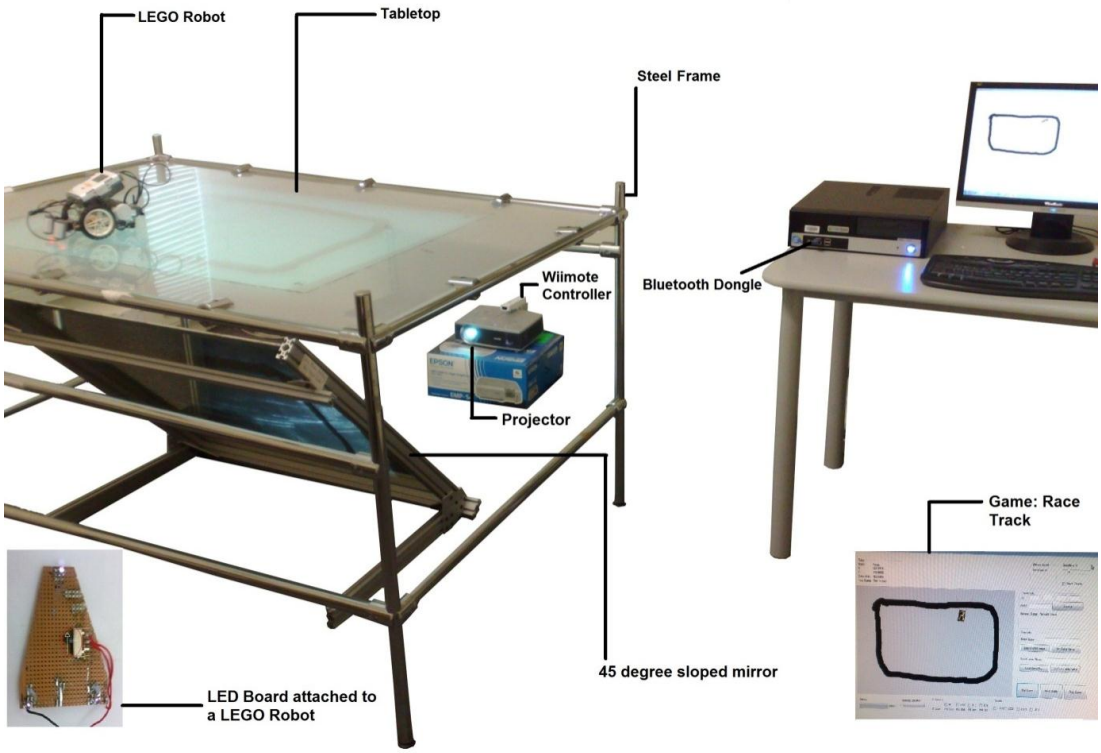


Figure 2-2 RoboTable at Lincoln University

The physical components of a RoboTable are a tabletop, a steel frame, a projector, computers, an IR pen and a Wiimote[4]. The table top is made from half inch perspex acting as a rear projection screen. In order to produce rear projection screen effects, a translucent adhesive vinyl coating is applied directly to the perspex (Mason, 2005). The projector is placed in a position under the table where a 45 degree sloping mirror reflects the projected image onto the underside of the tabletop. A Wiimote detects the position of IR diodes attached to the underside of the robot. A computer is used to track the position of a robot using the information from the Wiimote. An IR pen is used for calibrating the coordinates of the tabletop.

Figure 2-2 shows a LEGO™ robot on a RoboTable. LEGO™ Group was founded in 1932 by Ole Kirk Kristiansen to motivate children and young people to develop knowledge through fun, learning and high–quality creative play activities. The first version of LEGO™ Mindstorms was released in 1998 as the Robotics Invention System (RIS), and the current version is LEGO™ Mindstorms NXT 2.0, which is used in the Lincoln University research lab. LEGO™ Mindstorms combines programmable controllers with electric motors, sensors, LEGO™ blocks, and LEGO™ pieces such as gears, axles, beams, and wheel parts. One software system that can be used to program LEGO™ Mindstorms robotics is

[4]Wiimote, a controller of a Nintendo Wii console, 10/01/2012, (http://www.wiisworld.com/wii-controller.html)

ROBOLAB which is based on LabVIEW[5], was developed at Tufts University (Zhao, 2008). The development of ROBOLAB ceased in 2009 and the last version of ROBOLAB is 2.9.4c.

Figure 2-3 shows a sample LEGO™ Mindstorms robot, which contains an NXT 2.0 controller, colour and ultrasonic sensors, servo motors and various LEGO™ blocks.

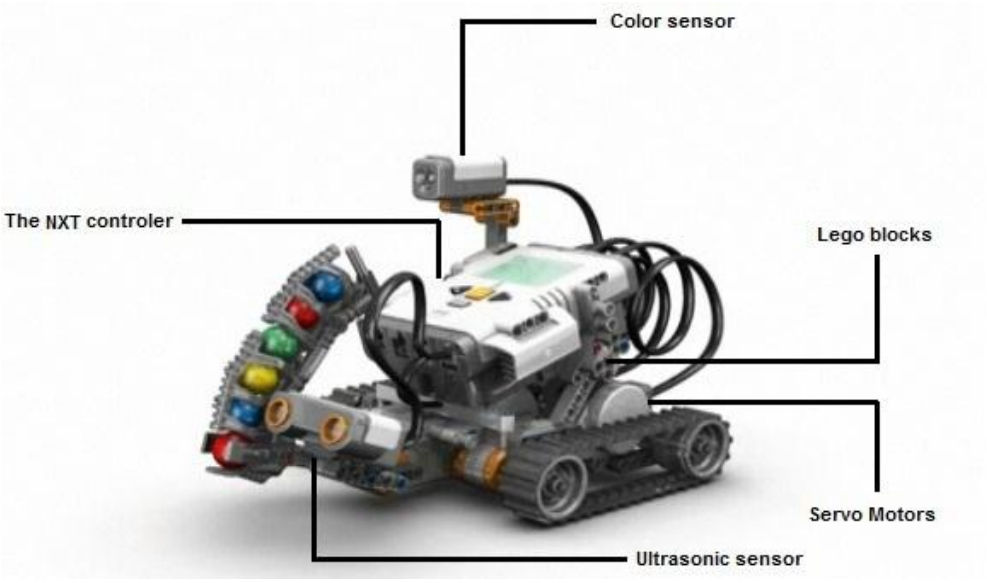


Figure 2-3 LEGO™ Mindstorms Robotics

2.2.3 Tracking Technologies

Determining the position and orientation of an object as it moves in the real world and displaying it on a computer screen is referred to as tracking (Pattie, 2004). Currently there are two main techniques to track the position of a robot for the RoboTable in 2-dimensional space. They are optical tracking (Festing, 2005; Mason, 2005; Pattie, 2004) and Wiimote IR tracking system implemented by Graham Smart. ARToolKit (2003) is a software library for building Augmented Reality (AR) applications, and it can be used to track square optical markers. The markers can be adhered to an object that can be used as a reference location to display a 3D virtual object, which is generated by computer graphics (Pattie, 2004). Pattie developed a tracking system by using the ARToolkit. However, Festing (2005) noted that the tracking system may fail in the event of bright-spots because the markers reflect light, they do not emit light. A bright-spot is a small, non-uniform, bright area, which comes from the reflection of lights through a semi-transparent tabletop. To overcome this weakness, Festing developed infrared techniques by using infrared (IR) filters in the near-infrared part of the electromagnetic spectrum. Festing (2005) argues that the infrared filters can eliminate the negative factors that affect the lighting conditions, and can produce a higher contrast image for marker detection enhancement. Graham Smart further developed the IR approach by implementing

[5] LabVIEW, a program used to automate testing and data gathering, (http://www.ee.buffalo.edu/faculty/paololiu/edtech/roaldi/tutorials/labview.htm)

a tracking system using a Wiimote and three Infrared light emitting diodes (LEDs). These small LEDs (Front LED, Left Back LED and Right Back LED) (Figure 2-4) are used as infrared light sources, and are attached to the base of a LEGO™ Mindstorms robot.

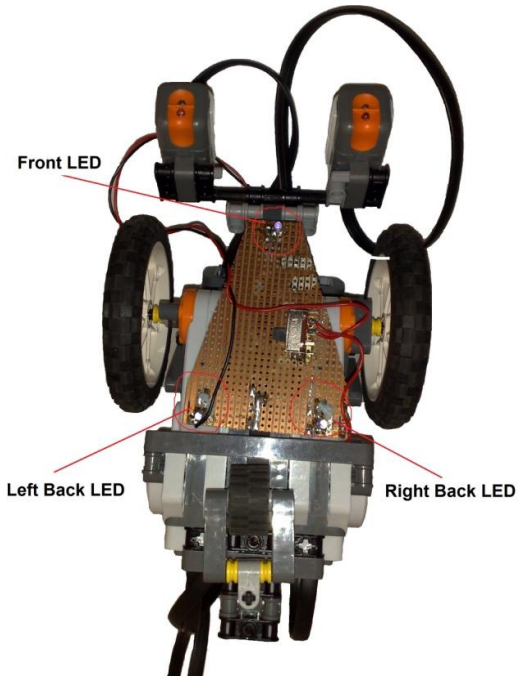


Figure 2-4 LEGO™ Robot with Three LEDs Attached

A Wiimote is used as an infrared detector, and it can detect the positions of the three LEDs and produce the corresponding raw position data. The Wiimote transfers the raw data to the computer via a Bluetooth receiver. There is no additional image processing required for the detection because the Wiimote can detect the raw position of the three infrared LEDs directly. Variations in lighting conditions have no effect on the tracking system because the three LEDs give off IR light directly. The Wiimote IR tracking system has been shown to work better than the earlier tracking systems and it has been adopted for the RoboTable at Lincoln University.

2.3 Network Communication

The following sections introduce some technologies related to network communication.

2.3.1 Network Architectures

Network Architecture is referred to as the structure of the network, which comprises the hardware, software, connectivity, communication protocols, as well as the mode of transmission. There are many types of network architectures in use, however, we are focusing on how a RoboTable can communicate with others, and thus only Client-To-Server and Client-To-Client architectures are introduced in this section.

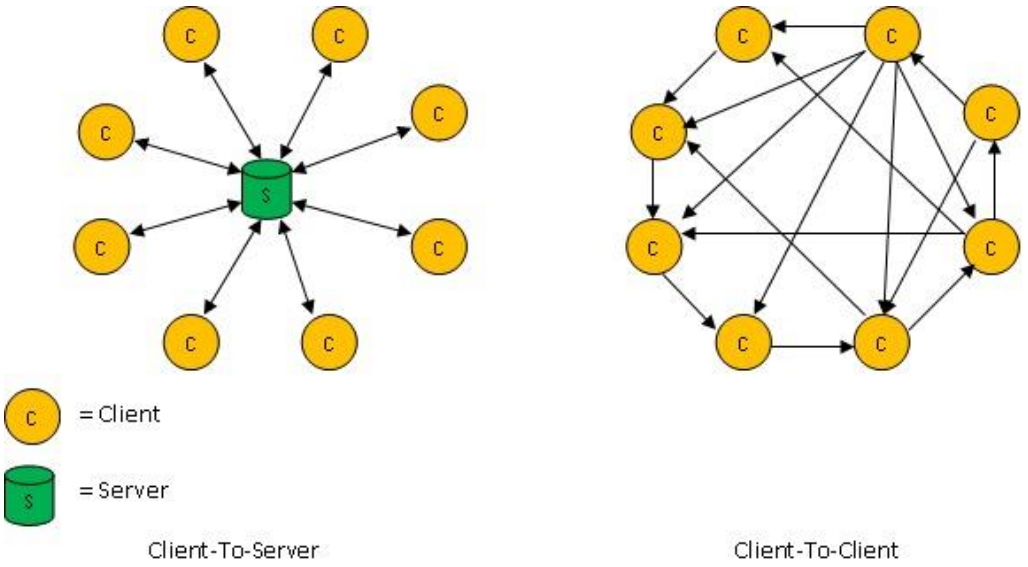


Figure 2-5 Network Architecture: Client-To-Server and Client-To-Client

A Client-To-Server architecture employs a server as a central agent and every client is required to register their network identification (IP address) with the server. In contrast, a Client-To-Client architecture does not require a central server, and clients register their network identification to other clients directly (Singh & Schulzrinne, 2005). The difference between Client-To-Server architecture and Client-To-Client architecture is shown in Figure 2-5. The two architectures work differently and have their pros and cons. Having a central server to manage every domain is convenient, and clients can reach each other with ease. However, if the central server fails to respond, it causes a single point of failure and all clients will lose the connection. The maintenance and configuration of a central server is the principal cost of a Client-To-Server architecture (Singh & Schulzrinne, 2005). On the other hand, the Client-To-Client architecture does not have a high maintenance cost because there is no central server involved. However, these advantages come with trade-offs of increasing resource lookup delay and reliability issues (Singh & Schulzrinne, 2005). The resource lookup cost for a Client-To-Client based system, which is *O*(logN), is much higher than a Client-To-Server based system which is *O*(1) (Singh & Schulzrinne, 2005).

2.3.2 Messaging protocols

Two popular network protocols for communication are TCP[6] (Transmission Control Protocol) and UDP[7] (User Datagram Protocol). TCP is a reliable connection network protocol, and it guarantees that data packets will be delivered in the correct order and without error. However, TCP uses more bandwidth than other protocols as it has a larger overhead to ensure error-free packet delivery. UDP has a smaller overhead which reduces the start-up latency in distributed applications. The start-up latency is the time it takes for a Network protocol to start to work before sending a message. One

[6] Transmission Control Protocol, 2012, (http://www.ietf.org/rfc/rfc793.txt)
[7] User Datagram Protocol, 2012, (http://tools.ietf.org/html/rfc768)

main disadvantage of UDP is it has no flow control, and developers have to implement this in their program (Tanenbaum, 2003). Other disadvantages of UDP are that firewalls often block UDP, and packets may be duplicated or lost.

XMPP[8] (Extensible Messaging and Presence Protocol) is a protocol using an XML stream specified by the IETF[9] (Internet Engineering Task Force) for carrying instant messaging over a network (Nei, 2006; Saint-Andre, 2004). XMPP is built on top of the TCP protocol. The advantages of XMPP are decentralization, open standards, security and flexibility (Saint-Andre, 2004). Decentralization means anyone can run their own XMPP server without the need for a central server (Saint-Andre, 2004). Disadvantages are large data overhead, limited scope for scalability and absence of binary data (Nei, 2006). Unmodified binary data cannot be delivered because XMPP is encoded as an XML document. To enable XMPP to deal with binary data, binary encoding such as Ascii85[10] or BasE91[11] encoding is required.

SIP[12] (Session Initiation Protocol) is a signalling protocol defined by IETF and it is widely used for controlling communication sessions over IP (Chen & Li, 2007). SIP is independent of the underlying Transport Layer, which means it can run on top of TCP, UDP or SCTP[13]. SIP is a text-based protocol incorporating many elements of SMTP[14] and does not support one-many routing. SIP reuses existing protocols such as HTTP. The disadvantage of SIP is that it is not designed to support large data sets because it is mainly used to deliver key signalling elements (Chen & Li, 2007).

AMQP[15] (Advanced Message Queuing Protocol) is an open standard for passing business messages between applications or organizations. AMQP can connect applications in different organizations on different platforms simultaneously or asynchronously. AMQP was officially launched in October 2011 (Burton, 2011), and its development has been led by a number of the large banks of the world such as JP Morgan, Goldman Sachs and industry backers including Microsoft and Cisco System. Vinoski (2006) introduced this protocol in 2006 while it was in development. The goal of AMQP is to make message-oriented middleware pervasive (Das, 2009). The disadvantage of AMQP is that it is developed to support a brokered model only and hence only defined communications for Client-To-Broker, i.e. lack of support to other communications (Gerardo & Angelo, 2007).

[8] Advantages and Strengths of XMPP, January, 2012, (http://www.iamatechie.com/advantages-and-strengths-of-xmpp/).
[9] The Internet Engineering Task Force (IETF), 2012 (http://www.ietf.org/)
[10] Ascii85, 2012 (http://ascii85.rubyforge.org/)
[11] BasE91, 2012 (http://base91.sourceforge.net/)
[12] SIP, 2012, (http://www.voip-info.org/wiki/view/SIPl)
[13] SCTP, Stream Control Transmission Protocol, 2012, (http://www.sctp.org/)
[14] SMTP, Simple Mail Transfer Protocol, 2012, (http://cr.yp.to/smtp.html)
[15] AMQP, January, 2012, (http://amqp.org/about/what)

2.4 Multiplayer Game Architectures

Multiplayer game architectures are categorized based on their network architecture. Currently there are two main multiplayer game architectures: Client-To-Server and Peer-To-Peer. The biggest difference between the Client-To-Server and Peer-To-Peer game architectures is where game state is managed (Bharambe et al., 2008; Zhao, 2008). A game state is referred to as information (such as the location of objects and the state of variables) about the game at a certain point in time (Zhao, 2008). In a Client-To-Server game architecture, the game state is managed at the central server. In contrast, with Peer-To-Peer architecture each peer has to manage the game state which is therefore duplicated. Advantages of the Client-To-Server game architecture are the management of game state is straightforward, and the cost of game state synchronization is low because all game clients will receive the same game state information. Disadvantages of the Client-To-Server game architecture are the cost of running a central server, a single point of failure is unavoidable, and there is a possible bottleneck of internet traffic for game state exchange. In the Peer-To-Peer game architecture, no central server is required which means it omits the risk of bottleneck of internet traffic and a single point of failure, as well as sparing the higher maintenance cost. However, the biggest disadvantage of the Peer-To-Peer architecture is inconsistent updating of game state information that may cause a game to become unsynchronised across clients (Singh & Schulzrinne, 2005).

2.5 Problem Definition

The main objective of a RoboTable game is to educate school children to assemble and program a basic LEGO™ robot in order to learn engineering knowledge. Education is the main objective of playing RoboTable Games. However, there are some problems with current RoboTable game development.

Two RoboTable games, namely Race Track, which is based on Line Follower[16], and Land Grabbing (Zhao, 2008) are used at Lincoln University. The Race Track was used as a demonstration of the Wiimote IR tracking system when Graham Smart had been developing the tracking code in 2011. The Land Grabbing is the only multiplayer RoboTable game (two tables) that has been developed using LabVIEW. However, both games are custom designed, which means they cannot be incorporated into a new multiplayer game easily.

The Land Grabbing game has been integrated with the optical tracking system that we have discussed in section 2.2.3. The optical tracking system has been shown not to be a very robust tracking method.

[16] Line Follower,08/07/2008, http://legoengineering.com/index.php?option=com_community&Itemid=59&c=item&id=204

Although Mason (2005) developed a demonstration game for the RoboTable, there are no general-purpose tools to support RoboTable game development or distributed game play. Hence, there is a need to have a set of tools integrating the Wiimote IR tracking system and providing efficient and effective network communications to support the development of distributed RoboTable games in a way that makes it easy for a new game to be developed and implemented.

2.6 Summary

Games can play a crucial role in school learning and RoboTable games can play a similar role in engineering learning. Without the support of robust tools, developing multilayer RoboTable games would be very difficult because the game developers have to deal with Robot tracking processing, network communication and multiplayer game architectures as well their particular game. In addition, RoboTable game developers would have to rewrite the entire system from “a clean slate” when developing a new game.

In order to facilitate easier multiplayer RoboTable game development, we have studied the requirements of RoboTable game development and the related research literature. The next chapter discusses the related research literature.

Chapter 3
Literature Review

In this chapter topics that are reviewed contribute towards the comprehension or enhancement of game architecture, network toolkit development, multiplayer game state management, and design patterns.

3.1 Game Architecture

Background information related to network communication and multiplayer game architectures has been discussed in the previous chapter. Client-To-Server and Peer-To-Peer architectures are widely employed in the development of multiplayer games. Hence, the distributed multiplayer RoboTable games will use one of these architectures. The following sections discuss an existing framework for distributed multiplayer RoboTable games that uses Client-To-Server architecture (Zhao, 2008) and the technologies that provide Peer-To-Peer architecture for games (Nagel, Evjen, Glynn, Watson, & Skinner, 2008).

3.1.1 Client-To- Server Architecture

Zhao (2008) presents a framework for distributed multiplayer RoboTable games using two network structures: Client-To-Server structure for network communication and Peer-To-Peer structure for game state management. In this architecture, shown in Figure 3-1, a communication server stores all the player profiles and forwards their game states to all tables. The server also forwards all communication between peers, so all peers can communicate with other peers. The game state is managed at each peer based on the information passed through the communication server.

The project toolkit (Figure 3-1) provides RoboTable-based functions to transfer information between RoboTables using TCP or UDP (Zhao, 2008). The existing communication toolkit integrates the TCP or UDP protocol and provides functions for network communication.

The architecture combines the advantages of Client-To-Server and Peer-To-Peer architectures, but it also contains the disadvantages of both. Zhao claims the communication server is extremely simple, as it only provides user profile management and forwarding game states. However, there is still a maintenance cost and the risk of inconsistent game states still exists. Zhao does not provide any mechanisms to synchronize the game states.

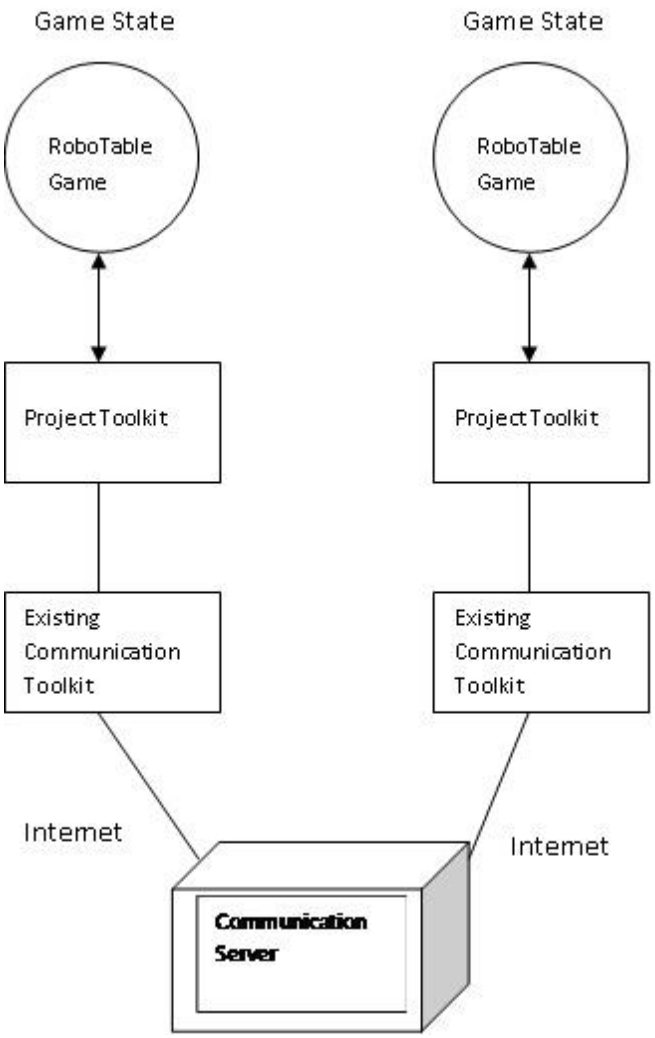


Figure 3-1 Zhao's architecture for multiplayer RoboTable Games (Zhao, 2008)

3.1.2 Peer-To-Peer Architecture

Peer-To-Peer architecture means that every pair of peers has a connection. The number of connections between any pair of peers is relatively small because peers are required to exchange information in a small group only, and removing a peer will not affect other connections (Nagel, Evjen, Glynn, Watson, & Skinner, 2008).

The process of Peer-To-Peer communication involves discovery, connection and communication. For example, the Microsoft Windows Peer-To-Peer Networking platform (Nagel et al., 2008) is an implementation of Microsoft Peer-To-Peer technology, and it includes two technologies that programmers can use to create .NET Peer-To-Peer application. The two technologies are:

- The Peer Name Resolution Protocol (PNRP), which is used to issue and resolve peer addresses.

- The People Near Me Service (PNMS) is an execution of the discovery phase that enables a user to locate peers who have signed in to the Windows People Near Me service in their local area (Nagel et al., 2008).

PNRP enables a peer to register an endpoint that is automatically distributed among peers in a cloud. The endpoint is known by a peer name and is encapsulated in a PNRP ID, which can be used to resolve to the actual peer name (IP address) so that other peers can communicate directly with it (Nagel et al., 2008).

3.2 Existing Networking Toolkits

This section studies groupware network toolkits in order to understand what factors result in a robust network toolkit.

3.2.1 GroupKit

GroupKit (Roseman, 1993) is a real-time groupware toolkit based on the INTERVIEWS toolkit, which is a C++ graphical interface toolkit supporting the development of a graphical user interface from a set of interactive objects (Linton, Calder, & Vlissides, 1988). GroupKit is designed based on the requirements of groupware developers and end users. Roseman (1993) outlines a number of steps that can be used when designing a toolkit using a user-centred approach. These steps are outlined below.

- **Specify Toolkit Domain** in which the toolkit will be used
- **Identify Developers** who are going to use the toolkit
- **Identify Use of Toolkit** such as using the building blocks provided by the toolkit to create domain-specific objects or using the domain-specific objects directly provided by the toolkit
- **Consider Target Application** in order to convert the common needs from the target application and possible future applications into the functionality and features of the toolkit
- **Design for Proper Use** to encourage developers to develop appropriate programs by knowing how applications should be developed using the toolkit
- **Apply Design Affordances** to advise developers how an object of the toolkit can be used
- **Apply Iterate Design** to repeat the process of design, test and redesigning the toolkit until it matches with the original design goal

These design principles are the basis for the GroupKit implementation. The derived design rationale of GroupKit is defined in terms of the following areas (Roseman, 1993).

- **Minimize developers' efforts** to create their application without reducing the quality of their product

- **Encourage developers** to employ suitable features using design affordances, where an affordance is defined as the properties of objects that imply particular uses to developers (Roseman, 1993)

- **Provide extensibility of the toolkit** using inheritance and polymorphism in object-oriented programming

- **Provide flexibility of toolkit components** to allow developers to build a wide variety of different systems with the trade-off of ease of use

The design principles and the subsequent design rationale of GroupKit provide the base guidelines for the proposed approach. We need to consider the trade-off between flexibility and ease of use.

3.2.2 Groupware Toolkit (GT)

GT (Alwis, 2009) is a toolkit for developing real-time distributed groupware systems on the .NET platform providing object-oriented, message-based network connections between distributed computers. Alwis (2009) argues that GT can deal with a number of the general aspects of network communication while still providing control over communication channels. There are seven ideas that separate GT from other groupware toolkits. They are latency-management techniques, application-level network control, generic shared dictionary, different types of message content from basic data types, multiple messaging paradigms, debugging and testing support, and extensibility (Alwis, Cutwin, & Greenberg, 2009).

GT employs modular design to separate the main aspects of network communication into three parts: communicator, connection and transport (Alwis, 2009). The communicator comprises two classes representing Client and Server. The Client and Server classes provide a number of programmer-facing APIs for programmers to interact with, to send and receive messages. The communicator sends and receives messages with other distributed computers through connections. A connection is a logical group of different network transports that programmers can select to connect remote endpoints (Alwis et al., 2009). An endpoint is a device which can be networked. The connection uses a packet scheduler to manage the marshalling of messages and conversion into a packet.

Alwis (2009) presents a high-level design of the GT framework. This gives an example of how to use modular design and design patterns that provide extensibility and flexibility to developers.

3.3 Game State Management

In order to provide a robust solution to support distributed multiplayer RoboTable game development, we need to consider how to manage the game states effectively. One approach is to allow a state to transition to other states depending on the conditions or actions that are required to be satisfied. For example, when a game is just started, the game is initialising all data such as loading data into the computer memory and we can define this as the “beginning state”. The game will transition to the next state such as the “ready state” only when all data have been loaded into the computer memory, otherwise, the game will remain in the “beginning state”. A technology to manage these state transitions is crucial for this project because the possible RoboTable games will have various states that it must transition between such as beginning state, initialising state, game starting state, game looping state and game finishing state. For that reason, we review some game state management techniques.

Traditional game state management can be defined as a finite state machine (FSM) (Brickman & Joshi, 2010) because these state machines encapsulate different game states and use events to trigger the transitions between states. With a finite state machine, programmers have to use conditional constructs such as if/else and switch statements to transition the game execution to the appropriate state (LaMothe, 2003). The problem with finite state machines is that they can be inflexible, difficult to maintain and hard to extend with limited states. The MIMESIS architecture is a solution that attempts to overcome the problem.

The MIMESIS architecture (Young et al., 2004) integrates plan-based action generation with a conventional game engine. The game engine initiates a planned request by sending a message to a MIMESIS component called the story world planner, which contains a number of states. The planned request identifies the requirements of the game engine, and the system creates a story world plan, which is a data structure that defines an action sequence to fulfil the requirement from the game engine (Young et al., 2004). The created story world plan is passed to an execution manager. The execution manager executes the plan by sending commands to the game engine and receiving feedback from the game engine for monitoring.

Schuller (2005) explains how to apply game state management using the State Pattern and provides comprehensive sample codes for developing game states with DirectX devices. Schuller (2005) starts by creating an interface for a general state and then implements some concrete state classes. The individual states can be used to plug into a state manager as required, which provides flexibility for game state management. Figure 3-2 is a sample of the game state management of Schuller, which shows that a state can be plugged freely into a state manager in a game loop.

http://img233.exs.cx/img233/6664/loop7zk.png

Figure 3-2 State Pattern management (Schuller, 2005)

3.4 Design Patterns

Schmidt (1995) explains how to use design patterns for initialising network services. A design pattern is a common repeatable solution to a recurring problem and it is a description or template for how to solve a problem that can be used in many different situations in software design (sourcemaking.com, 2012). Currently there are some limitations with existing network services because they are tightly coupled. There are two types of design patterns that can remove these limitations and decrease coupling, namely, Strategic design patterns and Tactical design patterns (Schmidt, Harrison, & Al-Shaer, 1995). Strategic design patterns have a general impact on the software architecture, which are typically oriented to solutions in a specified domain. Reactor and Acceptor patterns are examples of strategic design patterns (Schmidt et al., 1995). An event-driven server distributing network messages to its clients is an example of the Reactor pattern. An Acceptor pattern is used for passively initializing Network services. Tactical design patterns have a relatively localized impact on the software architecture, which is domain-independent. Wrapper, Adapter, Bridge, Factory Method, and Strategy patterns are categorized as Tactical design patterns (Schmidt et al., 1995). An AC power adapter that can make a US standard AC plug work with a European wall outlet is a sample of the Adapter pattern. Schmidt (1995) concludes that design patterns can alleviate coupling and unnecessary complexity in communication software. Schmidt (1995) also notes that both strategic and tactical patterns are necessary to build flexible and extensible solutions.

3.5 Summary and Discussion

There are two different game architectures which could be used to support distributed RoboTable games these are Client-Server and Peer-To-Peer. The game architecture of Zhao is a hybrid requiring a server to manage the user profiles and using a Peer-To-Peer game architecture implemented using Microsoft tools. The Peer-To-Peer component uses two technologies PNRP and PNMS. The PNRP and PNMS protocols are not available outside of Windows XP which impacts on their usefulness, however the concepts they employ to connect peers can be adapted

The design rationale of GroupKit provides clear steps for developing a robust network toolkit. Extensibility and flexibility are the two principal characters we need to consider when designing a solution for this project. Using the interface and abstract classes can provide polymorphism in object-oriented programming. This allows the game developers to extend the general functions for their own purpose.

The GT provides a design for a network toolkit and this design could be useful for developing a similar network toolkit to support the development of distributed multiplayer RoboTable games. GT is a groupware toolkit that contains a lot of features which are not necessary for the RoboTable game environment.

The possible RoboTable games will have different states need to be managed. Therefore, robust game state management to support the RoboTable games is an important consideration. The MIMESIS uses a story world planner that contains a number of states and uses story world planes to transition states. The story world plans define action sequences for the manager to execute and transition between states. Because a state of a RoboTable game might have some actions sequences such as reading input from users, setting up different components and waiting for commands, we can adapt ideas from MIMESIS to create a game state manager to manage all states and transition between states when some actions have been executed. The State Pattern discussed by Schuller (2005) provides a remarkably robust structure for implementing a state manager for the RoboTable games because the pattern provides extensibility and flexibility to the design of the game state management. We can create as many states as we want and transition between them easily.

It is appropriate to employ design patterns as basis for the development of distributed multiplayer RoboTable games. This will allow the game developers to extend the general functions and not require them to modify the existing code. We will therefore make extensive use of abstract classes and interfaces.

Chapter 4
Toolkit-Design

The goal of the project is to investigate how to support the implementation of distributed multiplayer games for a RoboTable environment; i.e. to support RoboTable game developers, so they do not have to implement multiplayer functionality from scratch for each new game.

We have studied background in chapter 2 about the use of games for learning, the development of the RoboTable, network communication technologies and multiplayer game architectures. We have also reviewed the literature in chapter 3 on existing game architectures, network toolkits, multiplayer game state management, and design patterns. The project problem has been defined in section 2.5.

In order to understand the requirements of distributed multiplayer RoboTable game development and to comprehend what functionalities a distributed multiplayer RoboTable game requires, the following section discusses an investigation for supporting distributed multiplayer RoboTable game development.

4.1 Investigation for Supporting Distributed Multiplayer RoboTable Game Development

The investigation involves interviews with potential RoboTable developers and an illustration of a distributed multiplayer RoboTable game.

4.1.1 Interviews

In order to understand the requirements of RoboTable game development, interviews were conducted with Alan McKinnon and Keith Unsworth who have been involved in the RoboTable project since it started in 2003. They were able to provide insights into the requirements of the system. The interviews were mainly focusing on the following areas (the question list can be found in Appendix A).

- Usability of a RoboTable game
- Interface for RoboTable game development
- Main data of RoboTable games
- Techniques for RoboTable game development

Table 4-1 highlights the main requirements for the RoboTable game developers that came out of the interviews.

Table 4-1 Developers' Requirements

Technical requirements	
Distributed networked process	
	Provide robust network communication infrastructure
	Integrate text and audio communication
	Avoid having a custom server
Game management	
	Provide basic, but as close to real-time as possible, RoboTable tracking facilities
	Manage game setup and game play
Graphics model	
	Display local robot movement on remote tables

Following the interviews, it was concluded that RoboTable game developers need highly integrated working toolkits, so they do not have to implement multiplayer functionality from scratch for each new game and that the toolkits should simplify the process of RoboTable game development.

In order to comprehend what functionalities a distributed multiplayer RoboTable game requires, the following section uses an example to illustrate a game scenario and how it may work.

4.1.2 Illustration of a Scenario

To illustrate how the RoboTable games could work, Figure 4-1 presents a scenario of a distributed multiplayer RoboTable game.

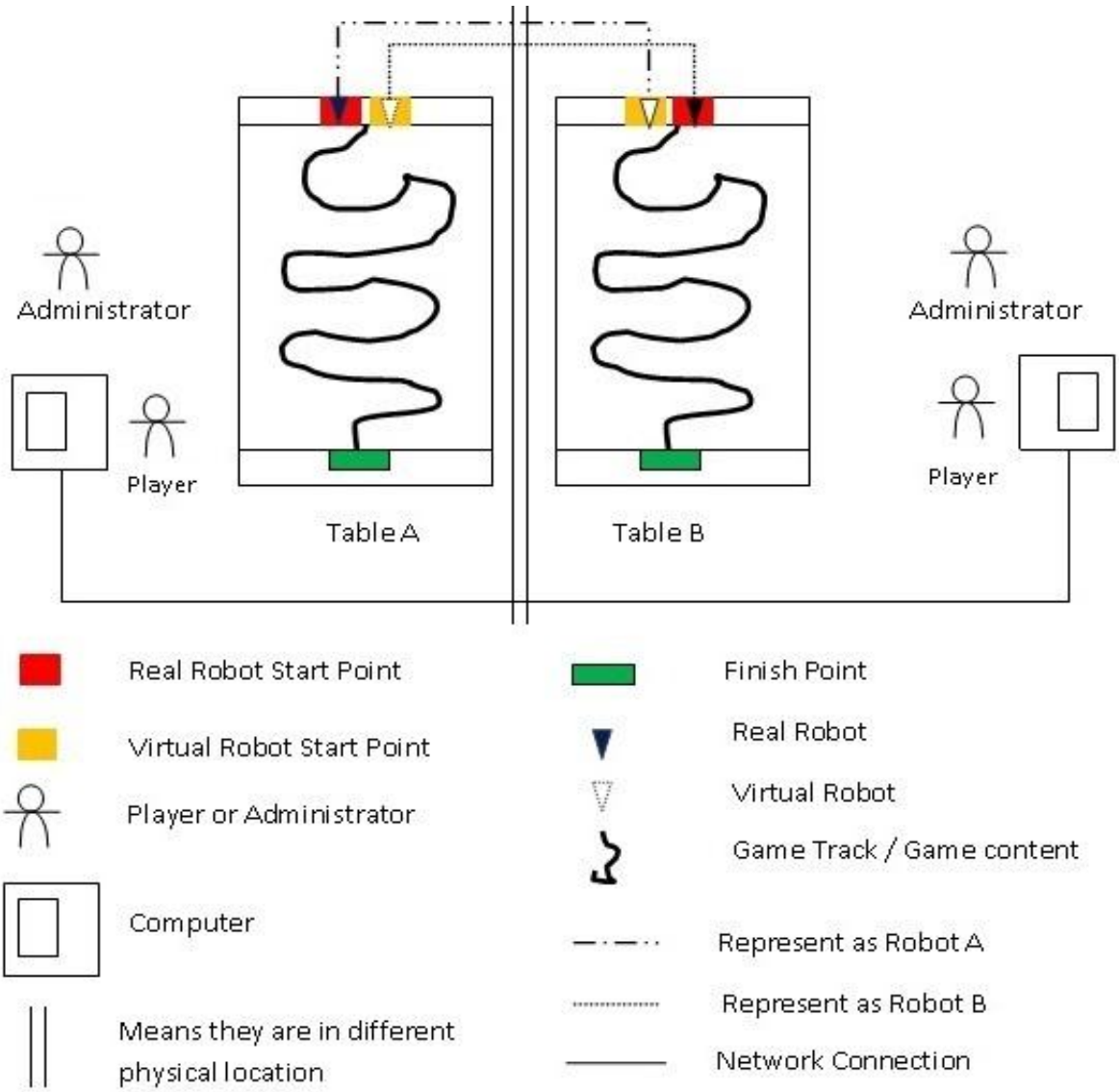


Figure 4-1 Scenario of a RoboTable Game

The game scenario shows a networked multiplayer RoboTable game (Race Track). There are two types of users in this scenario: administrator and game player. Table 4-2 gives examples of user types for RoboTable games.

Table 4-2 RoboTable User Type

Type of User	Role
Administrator	Lab administrator, Tutor, Teacher
Game Player	School children, student

Before the game starts, the administrators (Figure 4-1) at each table need to explain the game rules and train the students how to assemble and program their robots. Once the robots are constructed and programmed, the players at both sites need to place their programmed robot at the starting

point. Players turn on their robots when the game starts and the robots run along a black track, which is displayed on each table. At each table, the position and orientation of the local robot is tracked, and the latest robot update is sent to all tables through a computer so that a virtual robot representing the robot from the other table is displayed. The score is determined by the duration that a robot remains on the black track. For example, if a robot runs along the black track and does not move off the track for one second, the robot will get one score. Both tables receive the same "game end" message in order to calculate the final score. When the game ends, the calculated final score will be reported, and the winning table determined.

4.1.3 Conclusion of the Investigation

According to the scenario, a distributed multiplayer RoboTable game involves four main functionalities namely robot tracking, network communication, displaying local robot movement on remote tables and calculating game results. These four main functionalities are matched with the project requirements in section 4.1.1. Therefore, we need a solution that can handle robot tracking, network communication and game management on behalf of game developers and provide general interfaces to the game developers.

Currently, there are no general-purpose tools to support these main functionalities (as discussed in section 2.5). Hence developing a set of toolkits to provide real-time robot tracking facilities, deliver robust network communication infrastructure and manage game setup, display and play, is an appropriate solution for this project. The reasons we choose to develop toolkits as the solution are as follows.

- A toolkit can provide a software framework that developers can use to create applications easily as they only need to work with general interfaces provided by the toolkit.
- Each toolkit is independent of the other toolkits. Hence, if one toolkit needs updating, then we only need to replace the one toolkit and the whole system still works.
- Without toolkit support, developers may have to create all functionalities repeatedly for each new application. If a major function of an application without toolkit support needs to be upgraded, it is impossible to change the function for all applications using the same function.

Therefore, the design goals of the solution to fulfill the goals of the project are outlined below.

- Flexible and extensible
- Using the solution should be straightforward

- Good performance
- Good functionality

The following section explains the design goals outlined above.

4.2 Solution Design Goals

The solution should enable RoboTable game developers to develop games for educating young people about aspects of engineering without them having to develop their own code for robot tracking and network communication from scratch for each new game. In order to achieve this, the solution should match with the following goals.

4.2.1 Flexible and Extensible

The solution should be flexible and extensible so that game developers can extend an existing class easily without modifying the existing code. If a function such as Robot Tracking is required to be upgraded, it should be easy to be updated for all games developed using the solution.

4.2.2 Using the solution should be straightforward

The solution should be able to be used to create distributed multiplayer RoboTable games easily by someone who knows how to program a RoboTable game. The solution should manage the game setup, display and play on behalf of game developers and game developers should not need to create their own multiplayer functionality.

Game developers should not need to create their own functionality for robot tracking and network communication. The solution should handle robot tracking and network communication well within a solid box and provide general interfaces for game developers to use.

4.2.3 Good performance

The solution should track the robot position update and display on remote tables in real-time so that all distributed multiplayer RoboTable games can run to completion in real-time.

4.2.4 Good functionality

The solution should take overall responsibility for managing game states (See section 2.4) and report the current game state to game developers. The game states are maintained at the local table. Hence, maintaining a consistent game state for every table is essential.

The solution should support about 3 to 4 tables playing concurrently in real-time. If a table connection drops during the game session, the game should continue without interruption.

4.3 Solution Design

To provide appropriate support for the development and management of RoboTable games, a multi-toolkit approach is proposed, and as shown in Figure 4-2.

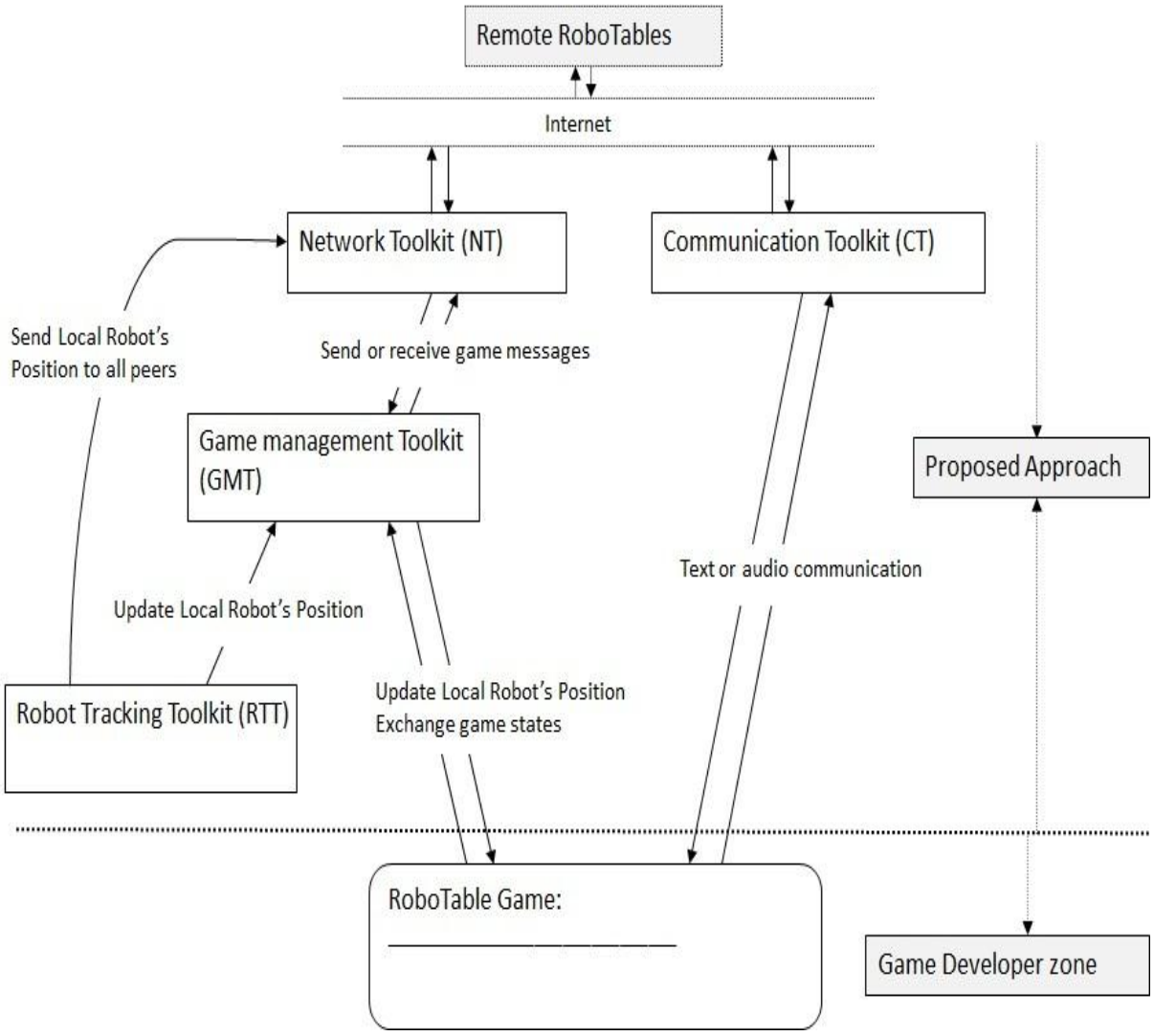


Figure 4-2 Design of the Proposed Approach

The toolkits comprise a Network Toolkit, a Robot Tracking Toolkit, a Game Management Toolkit and a Communication Toolkit.

The Network Toolkit is designed to maintain network communication channels, and it will use a Peer-To-Peer architecture for network communication. The reasons why we selected a Peer-To-Peer architecture for the Network Toolkit are as below.

- There are unlikely to be many tables (approximately up to 4 tables) available to play concurrently. Hence, having a server for a few tables is not necessary.
- Using a server is expensive as it requires hardware purchase and installation. Software licence fees may be an extra cost.

- Maintaining a server for a few tables and the tables are not going to be used 24 hours a day. Hence it is inefficient to employ a server for casual game play. Especially, as maintaining a server requires an extra cost for manpower.

Because we use Peer-To-Peer architecture, every peer will have a local server to receive game messages and send a game message to all peers. Hence we need an independent Network Toolkit to handle this in order to encapsulate all network functionality for ease of debugging. The independent Robot Tracking Toolkit will use the Network Toolkit to send position updates to remote tables, and hence it is better to keep the Network Toolkit independent rather than embed it within the Game Management Toolkit.

The Robot Tracking Toolkit is designed to track the position and orientation of a robot as well as communicate position updates to remote tables using the Network Toolkit. It will be developed based on the Wiimote tracking system. The reason why we choose the Wiimote tracking system is that the system has been shown to work better than the earlier tracking systems, and it has been adopted for the RoboTable at Lincoln University. The Wiimote Robot Tracking system (section 2.2.3) has been successfully developed independently of the current project and it is a self-contained system driven by the Wiimote. For that reason, we will upgrade the Wiimote Robot Tracking system to become an independent Robot Tracking Toolkit so that it can work self-sufficiently and automatically communicates position updates to remote tables using the Network Toolkit.

The Communication Toolkit uses a third party component to provide text and audio communication facilities as this provides a reliable and robust approach.

The Game Management Toolkit manages network communications, Robot tracking facilities, game setup and game play and is the integration point for game developers to integrate their game into the set of toolkits. It integrates with the Robot Tracking Toolkit and the Network Toolkit and provides generic interfaces to end users. The two toolkits can reduce the efforts for creating distributed multiplayer RoboTable games because game developers do not need to create their own multiplayer functionality and robot tracking functionality from scratch for each new game.

The reason we need a set of toolkits is that we want each toolkit to be independent and not tightly coupled with other toolkits. Hence, if a toolkit (for example the Robot Tracking toolkit) is out of date and we need to improve it, then we only need to replace the original toolkit and the whole system will still work. Each toolkit will be implemented as an independent DLL which means we can replace them easily, and the replacement will not interrupt other DLLs.

Moreover, we can develop and test each toolkit independently which means the design can simplify the process of implementation of the toolkits. “It would be infeasible to test a space shuttle as a system if you had to simultaneously question the design of every electrical component. It is similarly infeasible to test a large software system as a whole if you have to simultaneously question whether every line of code, every ‘if statement’, was correctly written” (Rodney, 1997). Therefore, implementing and testing on a small unit or module is better than a whole large software system. The Network Toolkit and Robot Tracking Toolkit are two small modules and the Game Management Toolkit will build on top of them.

In conclusion, the design shown in Figure 4-2 is a viable solution for this project and game developers are required only to work in the small developer zone. The solution hides the difficulties of robot tracking and multiplayer functionality so that game developers do not need to worry about them.

Design Rationale

We have discussed the design rationale of the toolkit developments in section 3.2 and the design pattern in section 3.4. The design rationale for GroupKit is useful for the design of the toolkits.

- **Minimize the effort for RoboTable game developers**: we will implement a Game Manager class to integrate the Network Toolkit and the Robot Tracking Toolkit. The Game Manager class reduces the effort required for developing a new RoboTable game. Game developers do not need to create their own process for network communication and robot tracking. The Game Manager also provides methods for game developers to interact with graphic rendering as well as communicating game messages.

- **Encourage developers:** we are going to use tool tips to help game developers use suitable methods to create games. The main methods that the game developers are expected to use will have tool tips. In addition, we will provide a skeleton project to help game developers start their game development.

- **Provide toolkit extensibility**: we will take their extensibility into account when designing the toolkits. The main class, Game Manager, will be based on abstract classes rather than concrete classes, and game developers will be able to create their own Network Toolkit or Tracking toolkit by extending the same base classes. The main components of the base Network Toolkit are also extensible, and this also provides flexible toolkit components to allow developers to build a wide variety of different systems. For example, game developers can create their own types of messages using the base game message class.

4.4 Game Management Toolkit Design

The Game Management Toolkit has overall responsibility for managing the state of the RoboTable games and is also the integration point for the Network Toolkit and Robot Tracking Toolkit. The State Pattern discussed in chapter 3.3 is used, because when a RoboTable game situation has changed, the game state can transition to another state easily using the State Pattern. The System State, which has overall responsibility for managing the whole game, uses the State Pattern, with a number of states as shown in Figure 4-3. The System starts in the first state (NotInitialized) and transitions to the last state (StoppedGame). The design of the Game Management Toolkit is focused on the system states because system state has control over the whole game from start to end.

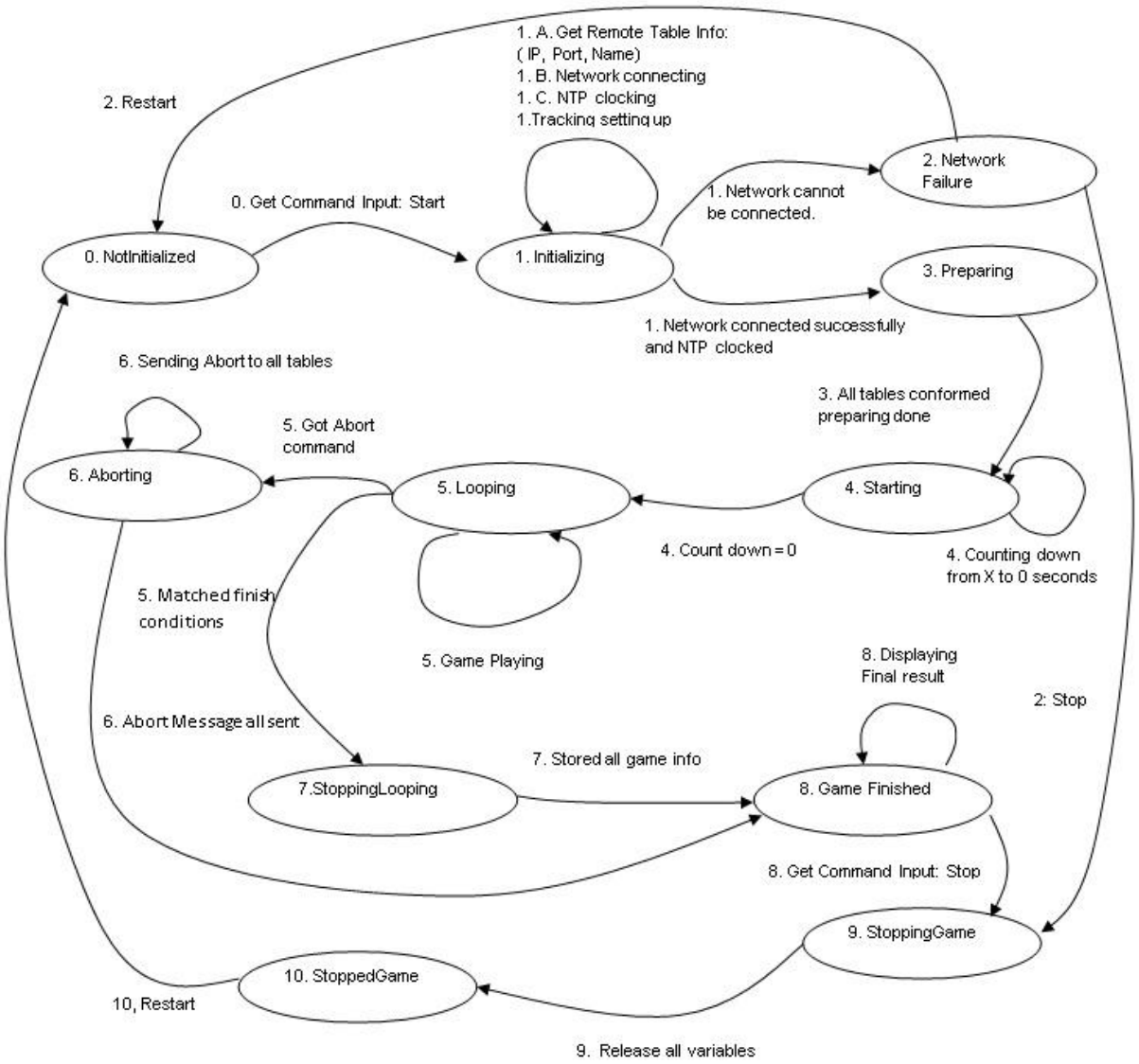


Figure 4-3 System State Transition Diagram

Table 4-3 outlines the interpretation of each system state.

Table 4-3 System States Definition

Step	System State	Definition
0	NotInitialized	A default state when a game starts to run on a PC. It is the earliest state of a Game
1	Initializing	Connecting to Network
2	Network failure	Network cannot be connected with current network information. User can choose to restart the game or terminate the game.
3	Preparing	Players are preparing for the game, e.g. placing robots on the start point of table, turning on Infrared Sensor
4	Starting	Game Starting. The starting state will use a Count Down state to display a countdown from 10 seconds to 0.
5	Looping	The game is being played
6	Aborting	The game is aborting, a message is sent to all peers to abort the game. The game will be aborted across all RoboTalbes.
7	StoppingLooping	The game loop is stopping and game result is being saved in a log
8	Game Finished	The game is finished and final result will be displayed
9	StoppingGame	The game is stopping and all game variables are reset
10	StoppedGame	The game is stopped and users can restart the game or exit the game.

According to the system state transition diagram shown in Figure 4-3, the main methods the Game Management Toolkit needs to provide to game developers are as below.

- Begin(): start the game manager engine. This method must be called before game initialisation.
- Initialise(): initialise the game toolkits.
- PreparingDone(): confirm preparation is complete. This will automatically start the game when all tables have confirmed.
- Abort(): abort the game during the game play session.
- Stop(): stop the game.
- Restart(): restart the game if the game is finished or aborted.

The Game Management Toolkit uses events to inform game developers of the latest updates and hence game developers are required to subscribe their application (game) to it. The possible events are listed below.

- Local Robot Update Event: update from the local robot.
- Error Event: receive notification of major errors.

- State change Event: updates to system state.
- Robot List Update Event: receive an updated list of robots.
- ReceiveOperationalMsgEvent: receive the operational messages sent by any table.
- CountDown Event: update the countdown value during game start.
- Goal Message Receive Event: receive the goal message sent by any table.
- TextMsgEvent: receive the text message sent by any table.

These functions and events are enough for game developers to integrate their game with the Game Management Toolkit. Because the Game Management Toolkit integrates the Network Toolkit, it provides general interfaces for game developers to send messages.

To send messages to all tables, the following functions provided by the Game Management Toolkit are designed for the game developers to use. The BaseMessage is an abstract class of the game message and we will discuss it in the following implementation chapter.

- SendGameMapLocation (string GameMapUrl): send the game map image location as a URL.
- SendLocalRobotImageLocation (string LocalRobotUrl): send the local Robot image location as a URL.
- SendLocalRobotColor (color RobotColor): send a colour that is picked by users as an RBG value.
- SendMessageToTables (BaseMessage Msg): send custom messages to all tables.
- SendMessageToSingleTable (BaseMessage Msg, string TableIP, int TablePort): send custom messages to a specific table.
- SendTextMessage (string TextMsg): send a text message to all tables.

4.5 Robot Tracking Toolkit Design

The current tracking system uses IR LEDs and a Wiimote as described in section 2.2.3.

This tracking system was developed as a proof-of-concept to evaluate the effectiveness of using the Wiimote for Robot tracking. In its current state it is not suitable for use with the proposed toolkits. Therefore we propose to repackage the tracking system into a tracking toolkit and provide methods to return the x, y position of the robot and its orientation.

The Game Management Toolkit will have overall responsibility for tracking the state of robots as shown in Figure 4-4 and defined in Table 4-4. To track the state of robots, the Game Management Toolkit will check the last received position update time for each robot. If a robot has not received a position update within a specific time such as 2 minutes, the state of the robot will be changed to the Not Visible state or the Lost state.

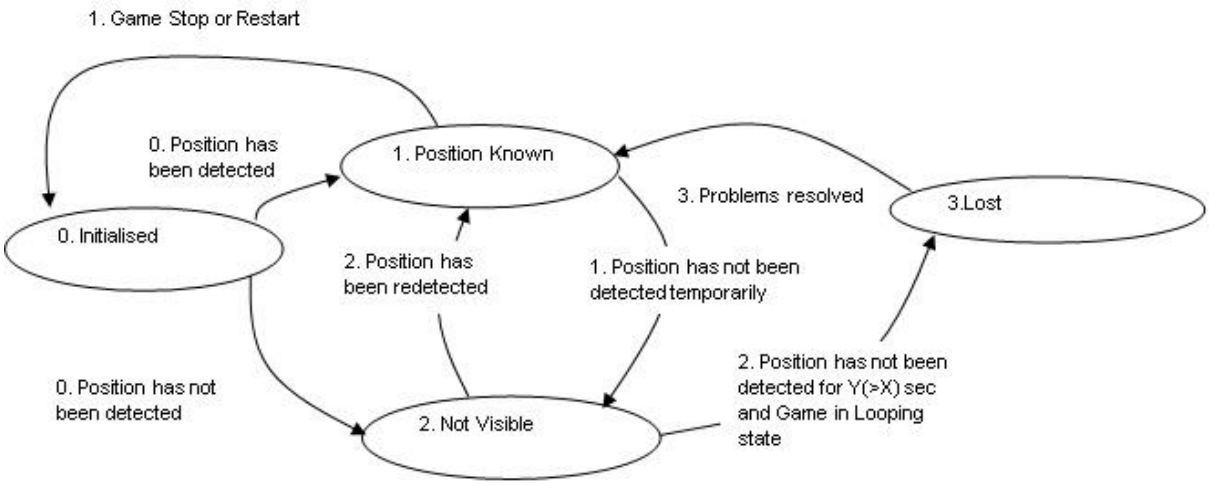


Figure 4-4 Robot state transition diagram

Table 4-4 outlines the description of each state.

Table 4-4 Robot State Definitions

Step	Robot State	Definition
0	Initialised	Beginning state to show the robot is ready
1	Position Known	The real robot currently is on the table and position information has been determined.
2	Not Visible	The real robot has not been located on the table between 2 seconds to 10 seconds(for example)
3	Lost	The real robot has not been located for more than 10 seconds(for example)

4.6 Network Toolkit Design

The Network Toolkit is responsible for managing the connections between a table and all remote tables. Figure 4-5 shows the transition diagram for the Network State of each connection. The Network State is used to check the network connection of peers and is used to determine if a peer has lost connection or temporarily disconnected. The local network connection is checked by the Game Management Toolkit during the initialising state, and if the local network connection is not available, the system state of the Game Management Toolkit will transition to Network Failure state (Figure 4-3). This design uses Peer-To-Peer architecture because if a peer's network state is Lost, the Network Toolkit will not send messages to it and the game can continue with other peers.

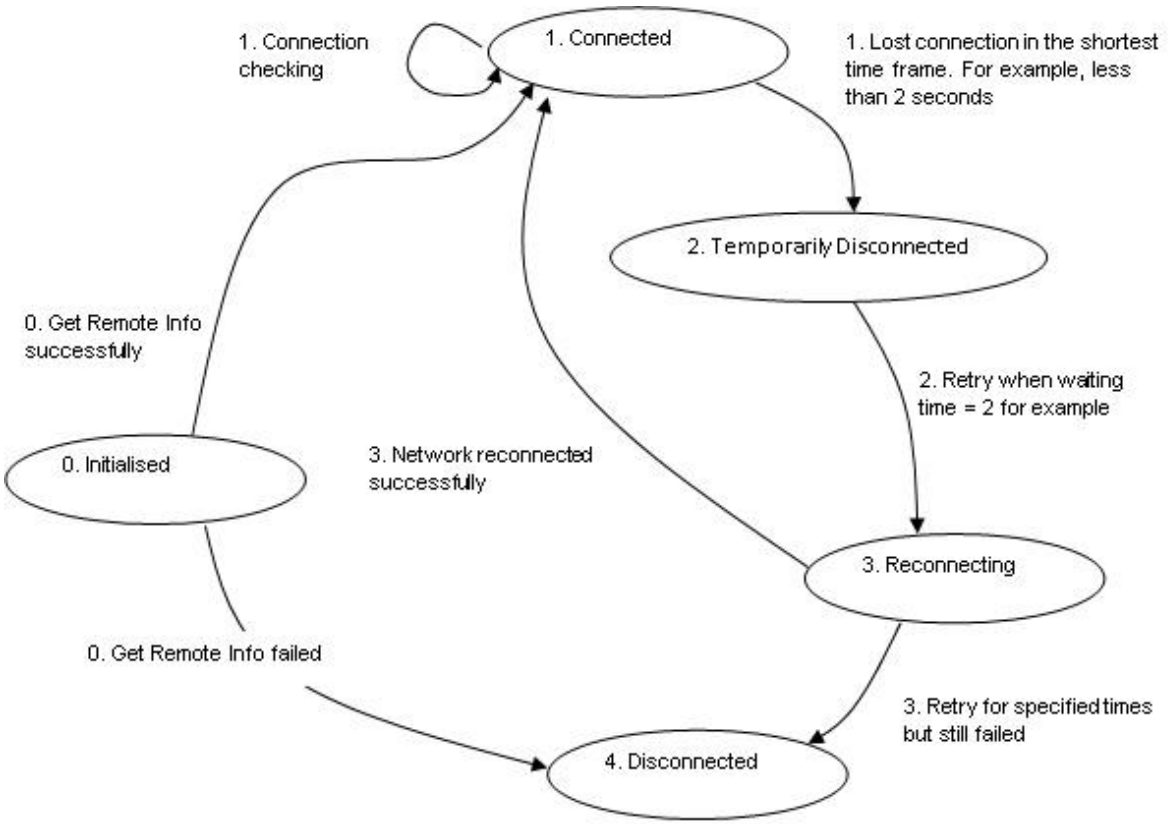


Figure 4-5 Network state transition diagram

Table 4-5 outlines each state.

Table 4-5 Network State Definitions

step	Network State	Definition
0	Initialised	Beginning state
1	Connected	The tables are connected
2	Temporarily Disconnected	The table has lost the connection in a short specified time frame e.g. 2 seconds
3	Reconnecting	The system is trying to reconnect Network
4	Disconnected	A table is disconnected from the Internet when it has lost the connection in a long specified time frame e.g. 10 seconds

The Network Toolkit class has the following main functions for game developers.

- Start(): starts the server communicator. A server communicator is a small module to provide network server facilities in order to receive messages from other tables. We will discuss this further in the implementation chapter.
- Stop(): stops the server communicators.
- SendMessageToPeers(BaseMessage message), sends messages to all peers.

- SendMessageToSinglePeer(BaseMessage message, string IP, int Port), sends messages to a single peer.

- Connect (string MasterIP, int MasterPort): connects with a master table.

The Network Toolkit uses peer-to-peer architecture to communicate between tables. The following paragraphs summarize network protocol selection, the concept of super peer and normal peer, the design of the game message and the message format.

4.6.1 Network Protocol Selection

Section 2.3.2 outlines some network protocols, which could be used for the proposed approach. XMPP could be one of the network protocol selections for the proposed approach to simplify Network communication for a RoboTable environment. However, XMPP has some unavoidable drawbacks such as large data overhead. SIP is mainly used for Internet telephony and is another option for the network protocol, pure Peer-To-Peer networking using SIP is achievable (Singh & Schulzrinne, 2005). However, SIP was not designed to support large data sets. AMQP is largely used for business message queuing although it could also be used for this project.

Because the workload for RoboTable game communication is extremely small, AMQP and XMPP were not selected. Because the game state is managed locally by each table, we must avoid inconsistent game state updating. To ensure that this does not occur, we have applied the TCP protocol for network communication because this protocol is said to guarantee delivery of data (Makofske, Donahoo, & Calvert, 2004), even though there are suggestions that the TCP may not truly guarantee delivery (OpenVOSBlog, 2012). We do not propose to implement a technique to make sure all peers receive consistent game state above the provisions of TCP. Use of an unreliable protocol, e.g. UDP, may require such a technique. This is outside the scope of the current work but is explored in section 7.2.2 during a discussion of future work.

4.6.2 Peer-To-Peer Architecture

The Network Toolkit operates in two modes; one is designated the “Master table” mode when it acts as a super peer to coordinate game setup, the toolkit then reverts to being a normal peer. In the second mode, the toolkit always acts as a normal peer. The reasons we need a super peer are:

- There are unlikely to be many RoboTables available to play concurrently. Hence, they can make an offline agreement easily using email, phone or video chat for a small group. An agreement would provide game details such as when to play, how to play, which game to play and which table is the super peer. The game agreement is required for the RoboTable game because we need to make sure all tables have the same game rules, and have enough time to train students to assemble and program their robot.

- A super peer is the game originator and invites other tables to join and play. The super peer needs to send its network information to other tables, and thus other tables will know the super peer's IP address and port number.

- It is not necessary for a normal table to know about all other tables before they can play a game as long as all tables know one table i.e. the super peer. This will reduce the difficulties of running a game because a table does not need to connect to all other tables by asking them to provide their IP address and port number one by one.

Figure 4-6 illustrates the concept before and after the game starts. One table is the master table or super peer according to the game agreement, and other tables are normal peers. The peer list, which is a collection of all peer references (each peer reference contains information such as Network information and peer name), is managed by the master table. Before a game starts, all tables connect to the super peer in order to receive the latest peer list. When the game starts, all tables are normal peers, which means if any of them is offline, the game can continue because each peer has the same game state as others because each table sends its updates to all tables. When the game has started, new tables cannot join the game.

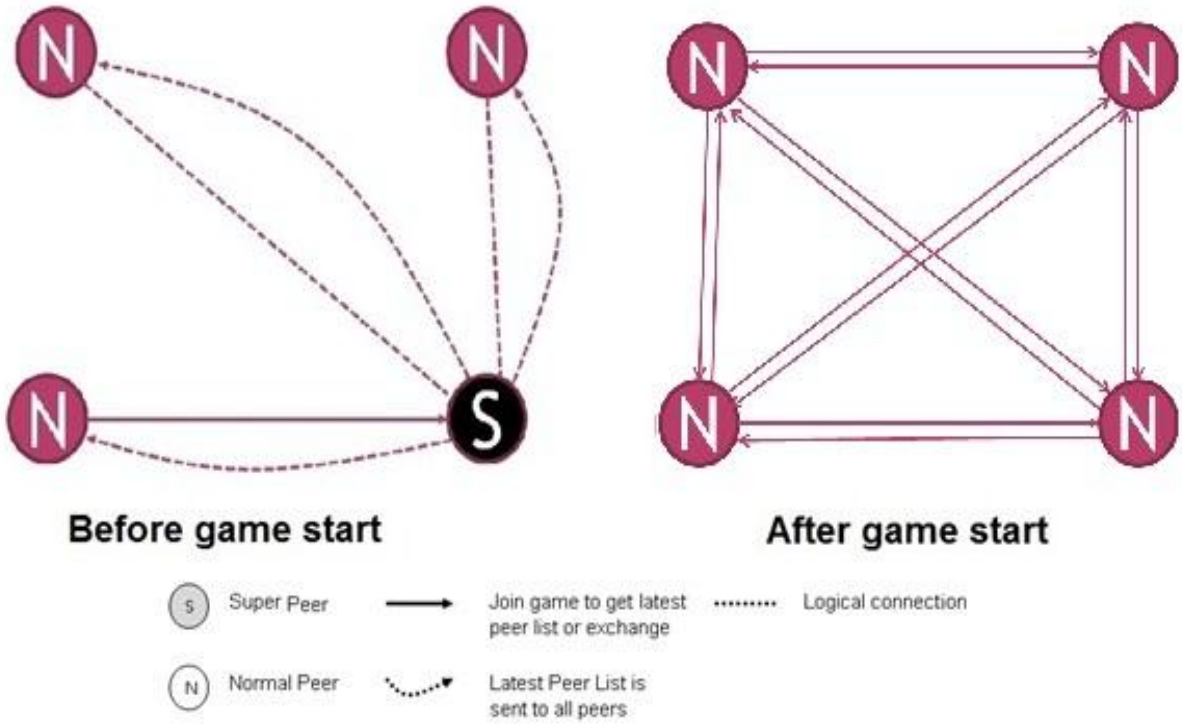


Figure 4-6 Super Peer and Normal Peer before game start and after game start

The main advantage of this Peer-To-Peer architecture is that it is easy to construct for a set of RoboTables without relying on broadcast networking. The disadvantages are that each peer has to send messages to all other peers, which may increase the volume and cost of Internet traffic (if they

are charged by the megabyte), while each peer may have a different network performance depending upon their local computer environment.

4.6.3 Game Message Types and Format

The game messages are used for network communication and they provide an easily understood interface to game developers. A game message is an object and game developers only need to add their message to the properties of the object and then pass the object to the Network Toolkit via the Game Management Toolkit. Game developers do not need to know how the Network Toolkit deals with the game message as long as they know how to deal with a normal programming object. The properties of the object provide tooltips to game developers and hence the difficulty of maintaining game messages is reduced. There are two types of game message in this case, operational messages and tactical messages. The position message is the only tactical message and the others are categorized as operational messages. The position message is used to communicate a robot update and other messages are used to set up the game such as connecting to a remote Master table, aborting a game, receiving the network status and the latest peer-list. Operational messages are mainly used for game control, network communication setup, network connection checking and peer list management. Hence, the operational messages are more important than tactical messages. Tactical messages are used to communicate robot updates on remote tables and missing a few robot updates is acceptable as long as a game can still run smoothly.

The following list outlines the possible messages associated with the two types.

- **Operational messages**
    - Network Message: messages about the status of network communication. They are sent between tables and are used for internal network status checking such as checking the network connection for a peer.
    - Game Control Message: messages about the controlling commands. They are sent between tables for important game controls such as connecting with a remote master table, or sent between developers and tables such as receiving/ sending a robot image, colour and game map.
    - Game State Message: messages about the current game state. They are sent between tables for state update (System State, Network State and Robot State).
    - Game Peer list Message: messages about the latest peer list. They are sent by the master table when it receives a new peer.

- ❖ Other Messages such as text messages, goal messages and custom messages. These messages are sent between developers i.e. a developer can send these messages to all other developers. Game developers can use the events provided by the Game Management Toolkit to receive the messages and use the methods provided by the Game Management Toolkit to send them to all tables.

- ➢ **Tactical messages**

  - ❖ Game Position Message: messages about the robot position and orientation.

Each message contains information about the IP address, Port number, RoboTable Name and Robot Name of the sender at the head of the message. The reason we need this information is because the Network Toolkit deals with messages from multiple tables. Therefore, tables need to know where the message comes from and which peer object is associated with the message. The combination of IP address, Port number and Robot Name is the key string of peer objects. We can use the key string to locate quickly the sender reference peer object from a peer list. Hence, the header information is important for this project. The Robot Name is optional and is reserved for future improvement when multiple robots can play concurrently on a table. We need to be able to identify each robot is if we have multiple robots running on a table. Hence, the Robot Name is important to be part of the key string.

The timestamp used in the game message is obtained from a NTP (Network Time Protocol) server, and it is used for synchronization across the different tables at different locations.

The format of each message uses XML (Extensible Markup Language[17]) because it is easy to manipulate using regular expressions. The message string format contains four parts, as follows:

- Document Head: contains information about the message's object type and the document identification to distinguish a game message from other messages. The object type is the assembly-qualified name of the system type (.NET environment) of the message object. The .NET environment provides a technique to create a new object based on the string format of the object type. Hence, we can reconstruct the original message object based on the string format of the message object.

- Message Head: contains general information such as IP, Port, RoboTable Name and Robot Name.

- Message Body: contains specific information.

[17] Extensible Markup Language (XML), ISO 8879, 2012, (http://www.w3.org/XML/)

- Message Tail: contains timestamp information for message sending.

There is serialization capability from the .NET environment that provides functionality to convert objects into XML files directly but the serialization has some disadvantages that we wish to avoid in this project. These disadvantages are listed below (Makofske et al., 2004).

- It is very inefficient because it may include information that is useless outside the context of the common language runtime (CLR)[18]. CLR is provided by the .NET Framework, which executes the code and provides services that simplify the development process.
- It contains overheads to provide flexibility that may never be required by the end user.
- Serialization and remote control cannot be used when a different wire format (standardized protocol) has already been specified. For example, a serialized message has been sent using TCP protocol, and then the message has been forward by the receiver to another receiver using UDP protocol. The receiver may fail to deserialize the message.

We have developed a class called NetStream, which is used to convert the properties of the network messages into a flat string and reconstruct the flat string back to the original message object based on its object type. A sample of the output string is as follows:

```
<Identification>NTStream</Identification>
<ObjectType>NetworkToolkit.Class.ControlMessage</ObjectType>
<Head>
        <IP>10.4.17.3</IP>
        <Port>3334</Port>
        <RobotableName>Lincoln-Table</RobotableName>
        <RobotName>Robot-Test</RobotName>
</Head>
<Body>
        <Controltype>System</Controltype>
        <Control>Connect</Control>
</Body>
<Tail>
        <SentTimeStamp>20/02/2012 5:34:42 p.m.</SentTimeStamp>
</Tail>
```

[18] Common Language Runtime, 2012, (http://msdn.microsoft.com/en-us/library/8bs2ecf4%28v=vs.71%29.aspx)

4.7 Communication Toolkit Design

Based on the requirements discussed in section 4.1, we use a third party software library (SkypeKit) to construct the Communication Toolkit. SkypeKit[19], which is popular for network communication, allows other applications to offer Skype voice and video calls and has comprehensive documentation.

4.8 Limitations of the Design

Limitations of the design are listed below.

- It is planned for a few RoboTables only (up to approximately 4 RoboTables). The reasons are that Peer-To-Peer architecture is not designed to support many multiplayer games, and it is unlikely that more than 4 RoboTables would be available to play simultaneously.
- The players must make a game-playing agreement such as when game play starts and how to play, offline before the game starts because the time for preparation is uncertain.
- Security issues such as preventing a vulnerability that could be exploited by a hacker through the server port of the RoboTable, are not considered in the proposed approach because it is not a requirement for the RoboTable game development at the moment.

4.9 Summary

The design comprises a Robot Tracking Toolkit, Network Toolkit, Communication Toolkit and Game Management Toolkit. Game developers only need to integrate their applications into the Game Management Toolkit and Communication Toolkit. Game developers need not be concerned with tracking the position of a robot and communicating with other tables. The design shows that the toolkits could simplify the process of developing a distributed multiplayer RoboTable game because game developers are only required to work in the developer zone (see section 4.3). The following chapter discusses the implementation of the design and we discuss an evaluation in chapter 6.

[19] SkypeKit, a collection of software and APIs, 2012, (http://developer.skype.com)

Chapter 5
Implementation

In the previous chapter, we discussed the design for a system that supports the creation and implementation of distributed multiplayer RoboTable games. This chapter discusses the implementation of this design, namely the creation of toolkits to support distributed multiplayer RoboTable game development.

The toolkits include a Network Toolkit, a Robot Tracking Toolkit, a Game Management Toolkit and a Communication Toolkit. The Game Management Toolkit integrates the Network Toolkit, the Robot Tracking Toolkit and provides generic interfaces for game developers to conveniently build distributed multiplayer RoboTable games. The Communication Toolkit is an independent toolkit, which provides text and audio communication facilities via the SkypeKit interface. Game developers can integrate the Communication Toolkit into their game application. The interconnection between toolkits is presented in section 4.3. The overall architecture of the toolkits is described in the following section.

5.1 Toolkit Architecture

The overall architecture of the system is presented in Figure 5-1. This shows how each toolkit integrates and communicates. The role of each toolkit is described below.

The Game Management Toolkit is the hub of the system managing most of the other toolkits and co-ordinating games.

The Network Toolkit provides network communication functions to the Game Management Toolkit.

The Robot Tracking Toolkit tracks the current position and orientation of a robot and then passes the update to the Network Toolkit directly. The Network Toolkit sends position and orientation updates to all connected tables. It also passes the position and orientation update to the local table on behalf of the Robot Tracking Toolkit.

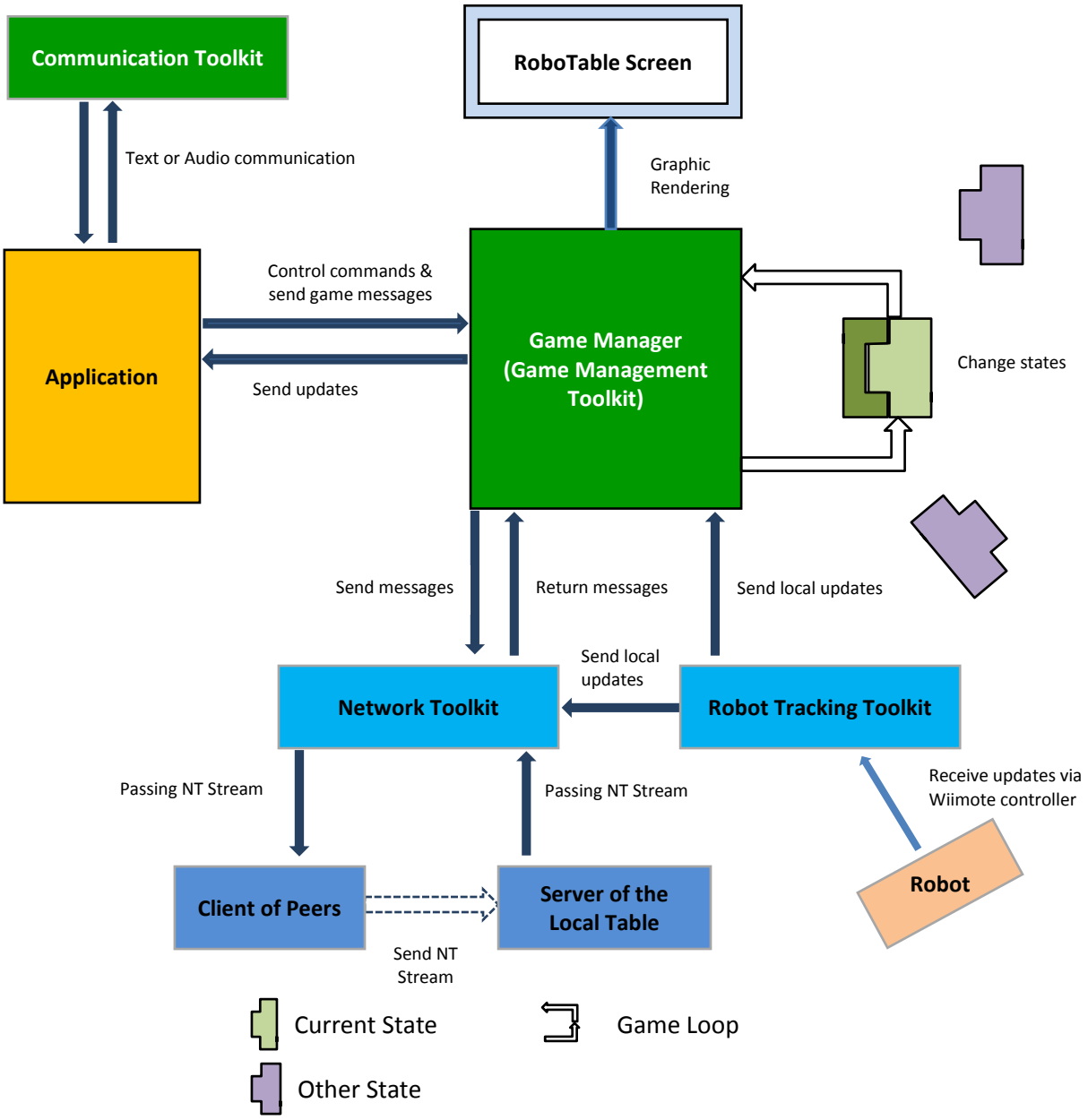


Figure 5-1 System Architecture

The application shown in the Figure 5-1 represents a RoboTable game. The game is responsible for initialising the Game Manager. The current game state will be reported to the game through event notification by the Game Management Toolkit. The game can control the Game Manager using pre-set commands and can send game messages to other tables through the Game Manager. The game subscribes to events provided by the Game Manager in order to receive game updates such as local robot updates, state updates, goal message updates, text message updates and error messages.

The Robot Tracking Toolkit detects the movement of a robot and transforms the position data into a standard position and orientation based on the display resolution of the RoboTable display. The standard position is defined as a point, measured in pixels, on the game map, and the value of the orientation represents the current heading of the robot.

The Network Toolkit converts incoming messages from the Game Management Toolkit into Network stream format, which is discussed in section 4.6.3, and sends the converted Network stream to all other tables. When a table receives a Network stream from another table, the Network Toolkit reconstructs the received Network stream into the original game message object and returns the game message object to the Game Management Toolkit. With the use of TCP, it was designed that the table was not necessary to send a confirmation back to the sender of the Network stream.

Providing support for displaying the location of a robot on a table is a requirement of the project. However, just displaying the location may not be sufficient. We may also need to display real-time game updates such as changing the colour of an area on the table or creating animation effects. In order to speed up the display components (compared with earlier display attempts), we have developed a set of graphics components for RoboTable game rendering, using DirectX.

In order to synchronize the clock for all tables, we have implemented a synchronous timer: NTP Clock. Network Time Protocol (NTP) is a networking protocol for synchronizing the timers of computer systems over packet-switched, variable-latency data networks (NTP.org, 2011). An NTP client is a UDP client that can receive the current NTP time. The NTP Clock uses a thread timer to update the NTP time every second and hence the virtual timers for all tables are synchronized because all tables send a time request to an NTP server and receive the current NTP time. The reason we need the NTP clock is that tables may be located in different time zones and may have different game results if tables are out of synchronization. Hence the NTP clock is important for synchronizing the game results across all tables.

Multithreading

In this project, we employ multithreading for the Network Toolkit and the Game Management toolkit. A thread is referred to as an independent stream of instructions in a program, which can be managed individually by an operating system scheduler (Nagel et al., 2008). Using multiple threads can allow different program units to work concurrently. The Network Toolkit needs threads to manage the local server and process received messages. The Network Toolkit also needs a thread to independently check the network connection status (Network State) for all peers. The Game Management Toolkit needs a thread to run graphics rendering independently and a thread to manage the system state.

Using multiple threads could cause race conditions and dead-lock to occur. A race condition occurs if multiple threads access the same data block (Nagel et al., 2008). A dead-lock can happen if multiple threads halt and wait for each other to release a lock (Nagel et al., 2008).

In order to deal with the above issues we use a lock statement to make sure an object or a block of data can be modified by only one thread at a time. The object or the block of data should not call another thread which could cause dead-lock to occur. To use a lock, we always use a private object as a key to manage the lock. Sample code in Figure 5-2, shows how we use the lock to synchronize a thread.

```
private object syn = new object();
public void MessageProcessing(string msg)
{
   lock (syn)
    {
            //a block of data that can be modified by multiple threads.
    }
}
```

Figure 5-2 Sample Code of Lock Statement

To modify the block of data, a thread is required to get control of the private object i.e. “syn”. If the object “syn” has not been released by the other thread, the current calling thread has to wait. The reason we use a private object rather than the keyword “this“ is that the keyword “this” will lock the whole class and hence is not a good way to make a code thread safe, because the rest of the class should still be able to be accessed by any other thread.

Programming environment

In this project, the toolkits have been developed using C#, the .NET framework 3.5. DirectX, Direct3D and Direct3DX are used for graphics rendering. The original Wiimote IR tracking system was written using C# and was implemented by Graham Smart. We have discussed the use of the Wiimote IR tracking system as a solution for robot tracking in section 4.3.

The .NET framework contains a large library and provides language interoperability across programming languages such as VB.NET, C# and J#. The .NET Framework is the most popular framework across the world and has good support for developing a robust distributed network infrastructure. For that reason, we chose the .NET framework as the platform to develop the proposed toolkits.

The reason we selected DirectX for graphics rendering is that the DirectX device can use hardware, such as facilities on a graphics card, to accelerate graphics processing. Hence, using DirectX can speed up the process of rendering game output. Another reason is that we can use Matrix operations to move or rotate graphics objects which is faster than redrawing the graphics objects pixel by pixel.

The following sections give details of the implementation of the toolkits.

5.2 The Game Management Toolkit

The Game Management Toolkit provides generic interfaces to the game developer to develop a RoboTable Game. Game developers are expected to work with the Game Manager class i.e. the Game Management Toolkit, to interface with the toolkits. The Game Management Toolkit is a state management machine. We have presented a state management literature review in section 3.3. The Game Management Toolkit has three main functions. The first one is managing the game states, the second is providing game control methods and the third is informing game developers of any changes of state, updating the robot list, receiving error notifications, and Network messages. A robot list is a collection of robot object references. We will discuss how to use control methods and event updates to receive information updates in a case study (section 6.3) and the developer user guides (Appendix B). In this section, we only discuss the states and the basic structure of the Game Management Toolkit. The network state is managed by the network toolkit and reported to game developers via the game management toolkit. We discuss the implementation for each type of state in the following three sections.

5.2.1 System State

The Game Management Toolkit uses the State Pattern (see section 3.3) to manage system states. The State Pattern for the Game Management Toolkit is shown in Figure 5-3.

The process method in the Game Manager is run by a thread that checks the current state continually. If the state, returned by Transit(), is not the current one held by the Game Manager, the Game Manager will update the current state to that state.

The base state class encapsulates the general data and methods for all states such as the SwitchState method. The SwitchState method is used to switch the current state to the input state. The method is final and does not need to be modified in the sub-classes. The base game state inherits from the base state. A reference to the Game Manager will be passed to the base game state, and thus all the concrete sub-classes of the base game state class will have a reference to the current Game Manager. The current Game Manager object contains a number of actions for each state object to execute. The state can only transition to the next state when all of its actions have been executed, or the state has satisfied certain transition conditions. The transition diagram for the system state is discussed in section 4.4.

The Game Manager not only provides a number of actions for the states to execute, but also provides a number of methods for game developers to use. Details of the methods can be found in Appendix C.

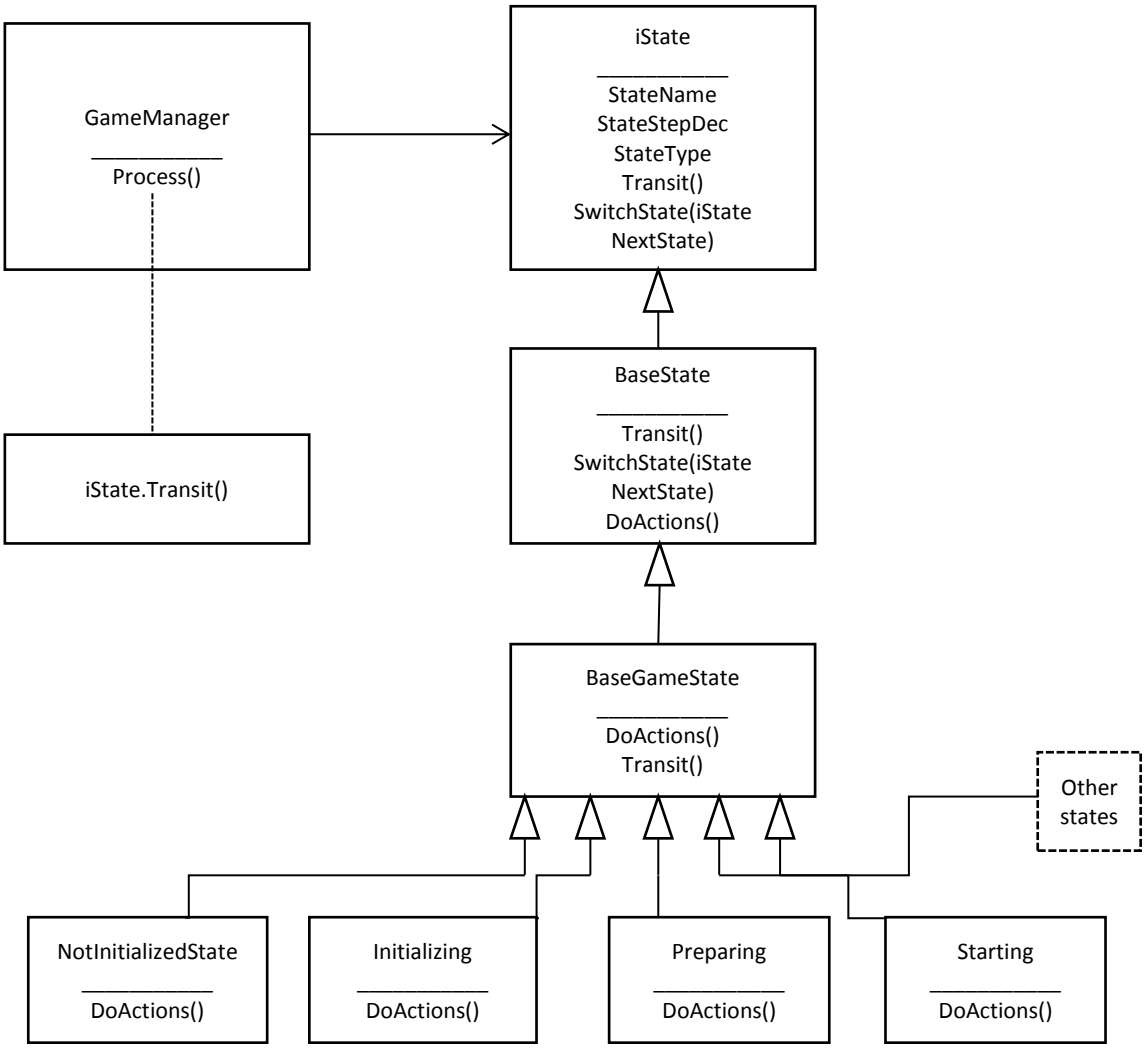


Figure 5-3 State Pattern for the Game Management Toolkit

Game state information is sent to all peers once it has been changed using the state message. The current state of all peers will be passed to the Game Management Toolkit by the Network Toolkit. The Game Management Toolkit will store the current states of all peers in its robot list. The robot list contains all robots representing the peers in the Network Toolkit (a local peer and remote peers). Each robot has a number of properties, and some are used to hold information about the current game state of robots. There are two types of robot, local robot and remote robot.

5.2.2 Robot State

The Game Management Toolkit also manages the Robot states, which we discussed in section 4.5, using an independent thread. The thread checks all robot objects for the latest position update time. The state of a robot object changes to Not Visible if the position of the robot has not been updated within 2 seconds (Table 4-4). If the position isn't updated within 10 seconds (Table 4-4), the Game Management Toolkit will change the robot state to the Lost state. When a robot state changes, the Game Management Toolkit will send a state message to inform all tables that there is a robot state

change in its peer list. The update time for the NotVisible and Lost state is configurable through the Table Settings.

5.2.3 Network State

The network state is managed by the Network Toolkit using an independent thread. The thread checks the connection status for all peers by sending a simple message. If there is any problem during the message sending, the network state will be changed to the Temporarily Disconnected state that is discussed in section 4.6. If the Temporarily Disconnected state lasts for a certain time, the state changes to Disconnected.

All states are reported to game developers through the robot list update event. The system state is reported to the game developer through the state update event.

5.2.4 The Basic structure of the Game Manager

The Game Manager integrates the Network Toolkit and the Robot Tracking Toolkit. Figure 5-4 outlines the structure of the Game Manager. The Game Manager holds references to the BaseNetworkToolkit class and the BaseTrackingToolkit class. The Network Toolkit class and the Wiimote IR tracking class are the concrete classes of the BaseNetworkToolkit class and the BaseTrackingToolkit class.

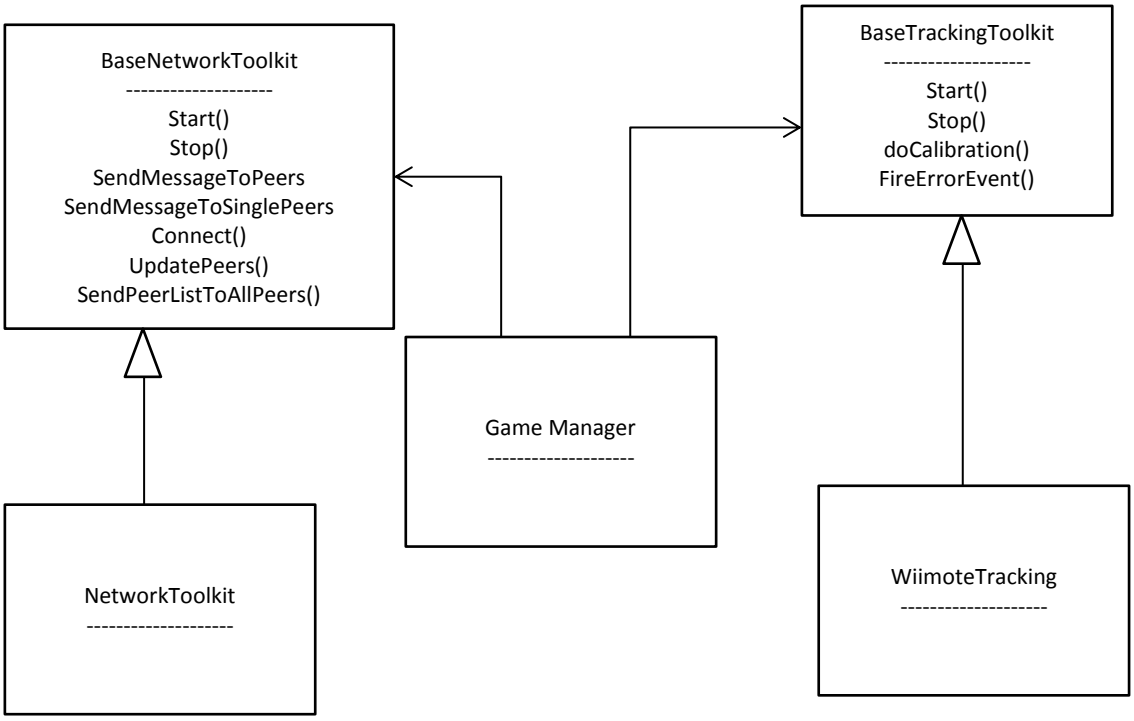


Figure 5-4 Basic structure of the Game Manager

5.3 RoboTable Graphics Components

The three main classes that comprise RoboTable Graphics are shown in Figure 5-5, namely RoboTableGraphic, RoboTableDrawingLayer and BaseRoboTableDrawing. An object of the RoboTableGraphic class is used as a graphics controller, an object of the RoboTableDrawingLayer class is used as a graphics layer and an object of the BaseRoboTableDrawing class is used as a graphics object.

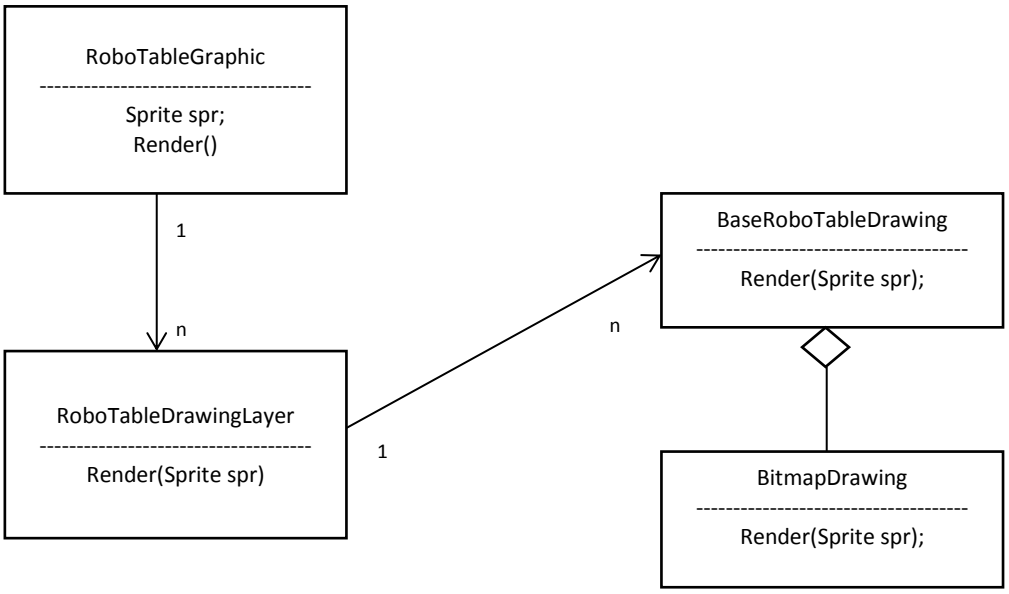


Figure 5-5 RoboTable Graphics Components

The following sections explain the design of the RoboTable Graphics Components in detail.

5.3.1 RoboTable Graphics Controller

The RoboTableGraphic class is the graphics controller, which initialises a DirectX device object and the main 2D drawing object: Sprite. Sprite is a 2D drawing object provided by DirectX. The Sprite object uses a texture to draw images. A texture acts as an image holder. The render method is the main method for a RoboTable game to draw on the table. The object of RoboTableGraphic class will loop inside a collection of graphic layers, and execute the rendering method for all RoboTable drawing objects. A sample code for the Graphics controller to render an entire game scene is shown in Figure 5-6.

The sample code outlines how to draw the whole game scene using a back buffer[20]. A back buffer is a render target that the DirectX device uses to present the drawing objects to the scene (display). Objects need only be created once and loaded into memory once. When we need to redraw the whole scene, we only need to clear the buffer and repaint all drawing objects into the buffer.

[20]What Is a Back Buffer, MSDN, http://msdn.microsoft.com/en-us/library/ff604993.aspx

Because all objects have been loaded into memory, the speed of redrawing the whole scene is much faster than creating a new instance of an object and drawing the screen pixel by pixel.

```
public void Render()
    {
        .....//other codes
        // Clear the back buffer to a blue color (ARGB = 000000ff)
        device.Clear(ClearFlags.Target, System.Drawing.Color.Blue, 1.0f, 0);
        // Begin the scene
        device.BeginScene();
        sprite.Begin(SpriteFlags.AlphaBlend);
        //draw the game on here
        RoboTableDrawingLayer TempLayer;
        foreach (KeyValuePair<string, RoboTableDrawingLayer> pair in _MultilayerRoboTableDrawing)
        {
          TempLayer = pair.Value.ShallowCopy();
          if (TempLayer.LayerVisible)
          {
            TempLayer.Render(sprite);
          }
        }
        sprite.Flush();
        sprite.End();
        // End the scene.
        device.EndScene();
        try
        {
          // Copy the back buffer to the display
          device.Present();
        }
        catch (DeviceLostException)
        {
          // Indicate that the device has been lost
          deviceLost = true;
         }
}
```

Figure 5-6 Sample Graphics Rendering Code

5.3.2 RoboTable Graphics Layer Class

The RoboTable Graphics Layer Class is used as a drawing object holder that can be placed on top of the other graphics layers, and provides graphics drawing functions that are easy to manage. Game developers can add, remove, hide or show the layer. Each graphics layer controls a collection of graphics drawing objects.

Currently the Game Manager provides six default graphics layers for game developers to interact with game scene rendering. These are shown in Figure 5-7. The lowest layer is for the initial game information, including the game name and the name of the game developer. The second layer is for the game map, and the third layer is for game utilities such as adding a new map that can be displayed with different land colours. The reason we need the game utility layer is that we do not want to alter the game map layer too often, because the game map normally acts as background,

and we do not need to redraw the game map repeatedly. The Game Info layer can be used to display state information. The Robot layer is the principle layer that the game engine needs in order to manage the movement of the robots. The Game Result Layer is used to display the final game result.

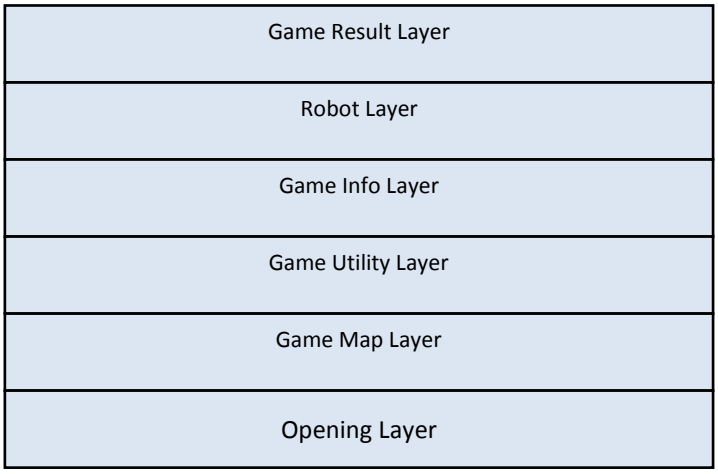


Figure 5-7 Default Graphics Layers

The default Graphics layers that are used for this project are created by the RoboTable graphics controller during game initialisation process. However, game developers can create their own layers easily using the abstract class of the Graphics layers.

5.3.3 RoboTable Graphics Drawing Class

A graphics object is an object that will be displayed in a RoboTable scene. A graphics object can be rotated, translated to different locations and scaled to different sizes. The graphics object content is usually an image, for example, a virtual robot is a graphics object that has a robot image. Similarly, a game map that displays on the game map layer is also a graphics object. A graphics object has a key name, and game developers can use this to control it. To control a graphics object in the RoboTable scene, game developers need to provide information specifying the layer the graphics object is in and the key name of the graphics object.

Currently the toolkits only provide one concrete graphics object class for a Bitmap image. However, game developers can implement their own graphics object class by inheriting the base graphics drawing class. Each graphics drawing class has a general interface Render (Sprite spr) for their parent layer to call. The BitmapDrawing class implements the base drawing class and provides the following code (Figure 5-8) to draw a bitmap.

```
public override void Render(Sprite spr)
{
  //change the current transform with the Identity matrix. The Identity matrix will change the current graphics
//object back to original state.
  spr.Transform = Matrix.Identity;
//transform with the new matrix (transition, rotation and scaling)
  spr.Draw(_BitmapTexture, _BitmapTextureSize, new Vector3(CenterLocation.X, CenterLocation.Y, 0), new
Vector3(0, 0, 0), this.Color);
}
```

Figure 5-8 Sample Code to Draw Graphics Object

The Transform property of a Sprite object is a matrix object and we use the structure class Matrix provided by DirectX to process the transformation in 2D. The method we have used is Matrix.Transformation2D, which is used to build a 2D transformation matrix in the XY plane.

The base graphics drawing class provides methods to control transformations of the drawing object, such as translation, rotation and scaling. The class modifies and calculates Matrix, and provides the following interfaces (Figure 5-9) to game developers.

```
public virtual void Rotation(float angle)
public virtual void Translation(int tX, int tY)
public virtual void Scaling(float sX, float sY)
public virtual void Resize(int width, int height)
```

Figure 5-9 Methods to Control Transformations of the Drawing Object

Every drawing object class inherits the same interfaces, so we can render the whole game scene easily. To move a drawing object from one position to another position, we can use the Translation method. To change the direction of an object, we can rotate it. We can also change the colour of the object though the properties provided by the base class.

5.4 The RoboTable Components

Interaction with the physical RoboTable is represented by a number of components comprising the RoboTableScreen class, the RoboTable class, RoboTableGraphic class and the Robot class as shown in Figure 5-10. The Peer class is managed by the Network Toolkit and is discussed in section 5.6.

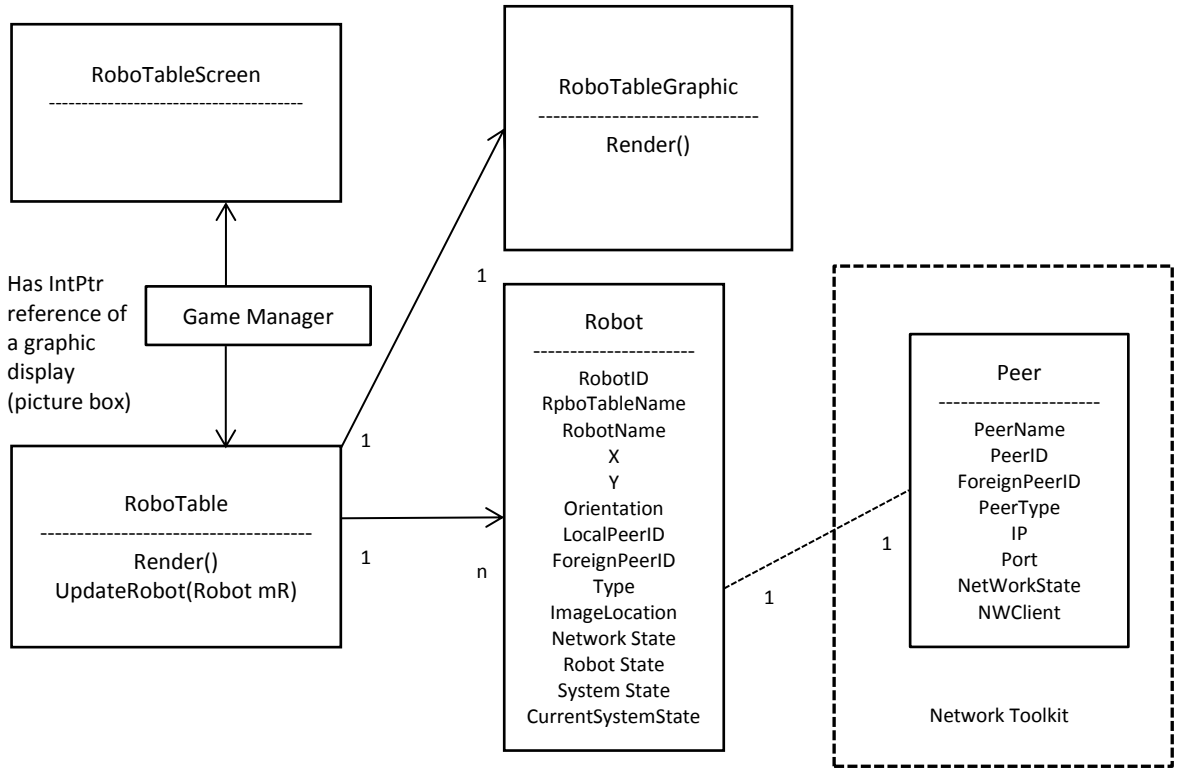


Figure 5-10 RoboTable Components

The RoboTable class is the main class that is used by the Game Manager, and it holds a pointer to a graphics display control from the RoboTableScreen. The graphics display control is a windows handle or a PictureBox handle in this solution. The type of pointer is an IntPtr as required by DirectX. IntPtr is a platform-specific type that is used to represent a pointer or a handle (MSDN, 2012). The RoboTable class has a reference to the RoboTableGraphic class (Graphics controller).

The Game Manager creates a RoboTable object and passes a windows handle IntPtr to the RoboTableGraphic object during object instancing. The RoboTable object creates a RoboTableGraphic object for drawing. The RoboTableGraphic object will draw the whole scene based on the IntPtr. The RoboTable also holds a list of robot objects that have a one to one relationship with the peer objects in the Network Toolkit. The robot object list is updated according to the peer list when it changes.

The robot class contains information about a robot such as its colour, image, name, type (local or remote) and its latest position and orientation. The RoboTable has a collection of robot objects (one robot object represents the local robot and the rest are remote robot objects) and the Game Manager renders all robot objects through the rendering method of the RoboTable object. Currently the RoboTable can only support one local robot because of the limits of the current tracking system. However, if a future tracking system can support multiple local robots, the whole solution can still work as we only need to replace the old tracking toolkit with the latest tracking toolkit as long as they use the same base class.

5.5 The Tracking Toolkit

The tracking toolkit has been developed from an existing tracking prototype. The original tracking prototype was a self-contained solution that contained both tracking and display functionality. In the tracking toolkit we only require the tracking functionality as display functionality is provided by other toolkits.

The tracking system uses a circuit board that has three IR lights as shown in Figure 5-11.

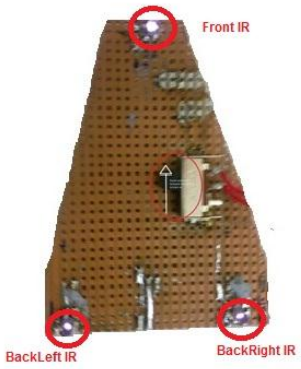


Figure 5-11 Tracking circuit board attached to a LEGO™ Robot

The circuit board is attached to the underside of a LEGO™ robot, as described in section 2.2.3. The original tracking code produces three position values: the location of the front IR, the back left IR and the back right IR with further calculations to locate the robot's position for subsequent display. In our solution, we require one location value, and we have chosen a centre point that is calculated from the location of the three IR lights. When we have calculated the centre point, we can calculate the orientation based on the location of the front IR and the centre point. The centre point can be calculated using the following equation:

$$\text{Centre.X} = \frac{\frac{(BackLeftIR.X + BackRightIR.X)}{2} + FrontIR.X}{2}$$

$$\text{Centre.Y} = \frac{\frac{(BackLeftIR.Y + BackRightIR.Y)}{2} + FrontIR.Y}{2}$$

The orientation θ of the robot can be calculated using the gradient between the Front IR point and the centre point.

$$\theta = \left| \tan^{-1} \frac{\text{FrontIR.Y} - \text{Centre.Y}}{\text{FrontIR.X} - \text{Center.X}} \right|$$

This equation gives a value between 0° to 90° within a quadrant. To get a 0°- 360° value we need to add an adjustment based on which quadrant the FrontIR is in. This adjustment is shown in Figure 5-12.

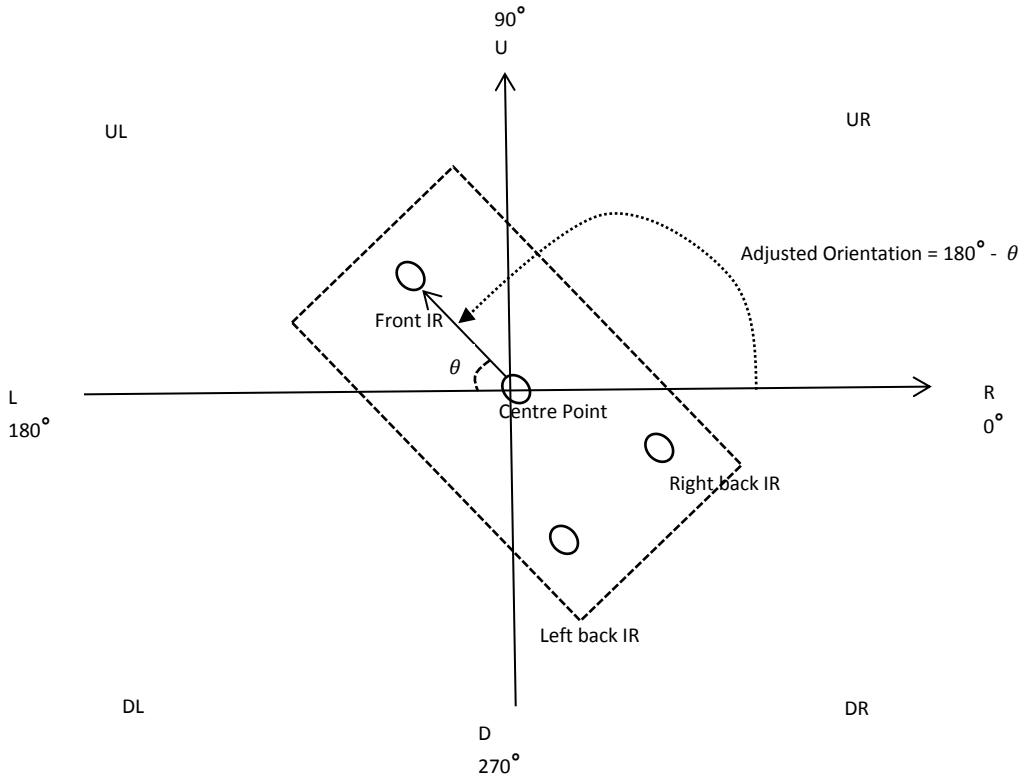


Figure 5-12 Orientation Calculation

A sample of how the value of the orientation is calculated is shown in Figure 5-12. Because the line from the centre point to the front IR point is inside the UL quadrant, the value of the orientation is between 90° and 180°. The adjusted orientation of the sample is 180° - θ.

The tracking toolkit is an independent toolkit, which automatically detects the latest position of the robot and sends the standard position and orientation values to the Network Toolkit. The Network Toolkit encapsulates the values into a position message. The position message is then sent to all tables through a network socket. The local position message is passed to the local table directly, without travelling over the network. The following section discusses the implementation of the Network Toolkit.

5.6 The Network Toolkit

The Network Toolkit has two parts, the first is a server that listens for incoming connections, and the second part is a client for communicating with other tables. The Network Toolkit also maintains a list of peers. Each peer has a client that connects with other tables to send position messages. When a position message comes from the tracking toolkit, the Network Toolkit loops through all peers and sends the message to their servers via the client.

The basic structure of the Network Toolkit is outlined in Figure 5-13.

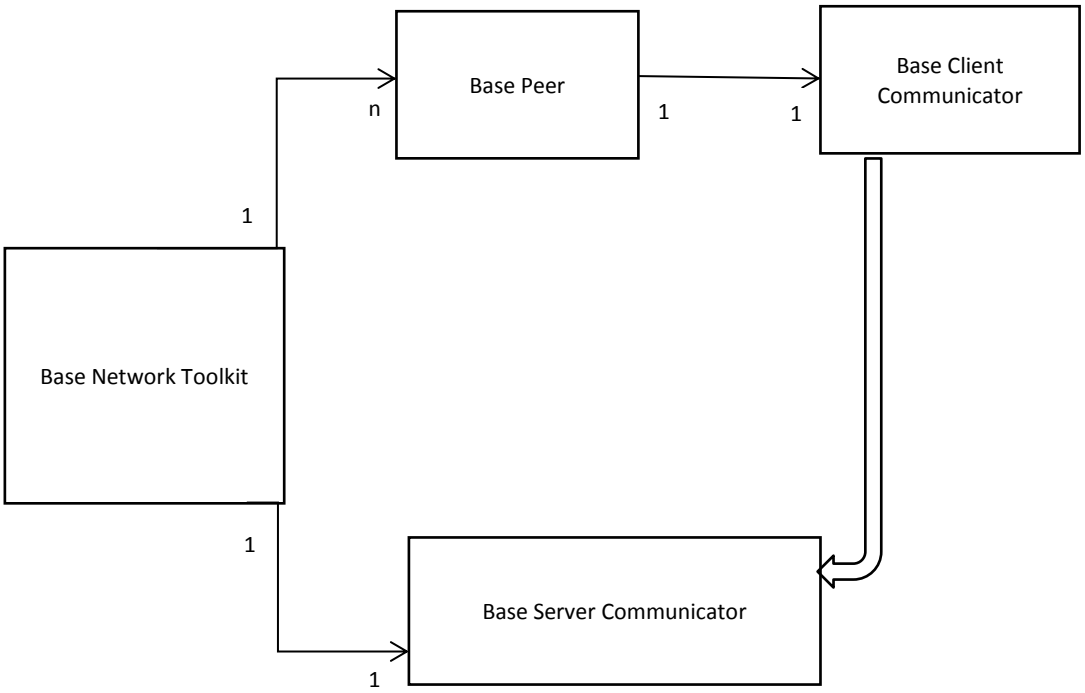


Figure 5-13 The Network Toolkit structure

Currently the Network Toolkit uses TCP client communicators and a TCP server communicator to provide reliable connection-oriented network communication. TCP can require more bandwidth than other protocols (see section 2.3.2). However, this drawback is acceptable within the requirements of the RoboTable game. The reason is that the packet size of the position message is small, and it is unlikely that many tables will play simultaneously. The position message (tactical message) is the main message that the RoboTable game will send out frequently.

5.6.1 Base Communicator

There are two base communicators: the base client communicator and the base server communicator. The base client communicator has three interfaces:

- Connected(): check if the current client is connected with the remote master server.
- Connect(): connect to the remote master server.
- SendMessage(): main method to send messages.

The base server communicator has three public methods:

- Start(): start the server.
- Stop(): stop the server.

- MessageProcessing(string msg) is used to filter the income string messages into two processing channels.

Figure 5-14 outlines the structure of the two communicators. Currently the BaseClientCommunicator class has one concrete class TCPClientCommunicator. The BaseServerCommunicator has one concrete class TCPServerAsync.

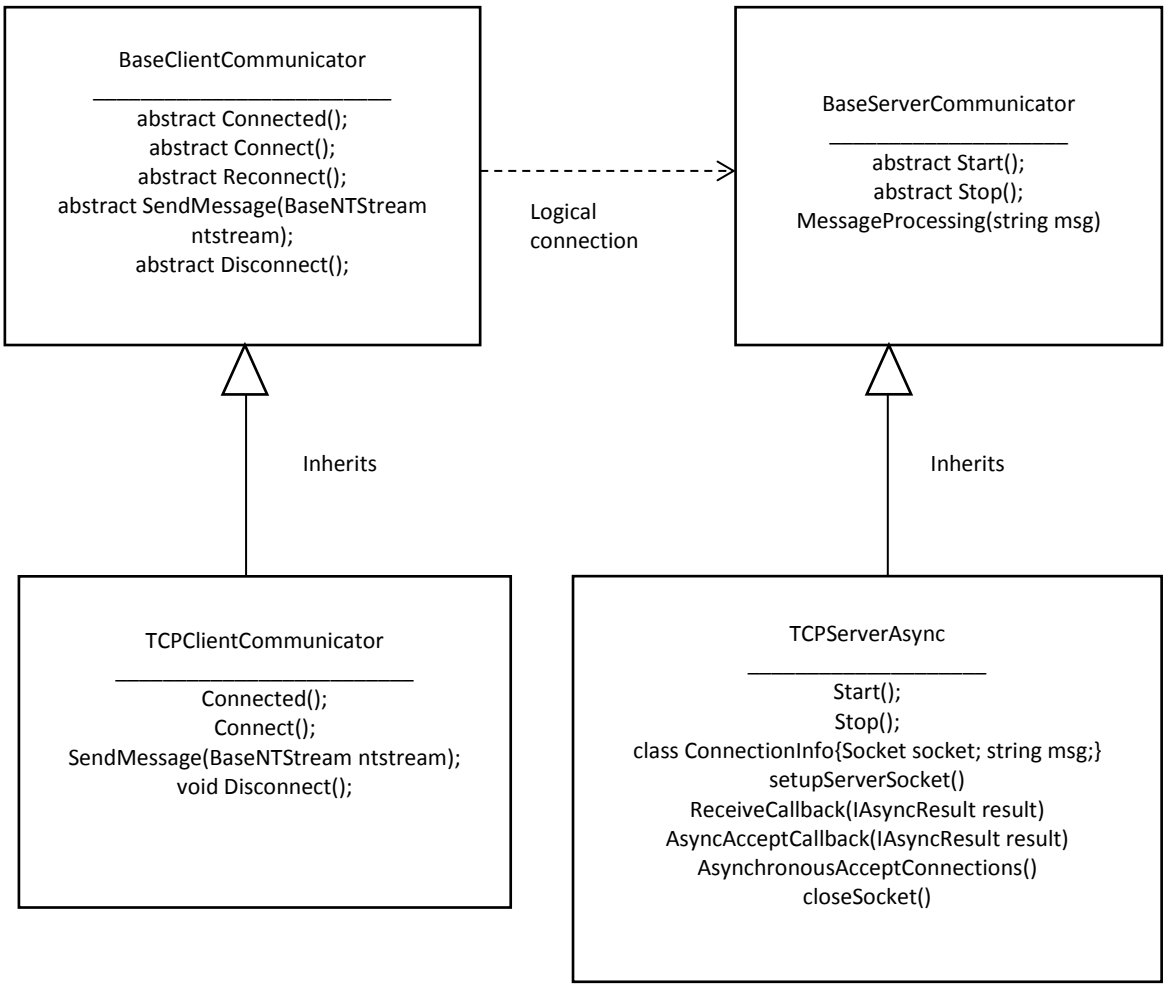


Figure 5-14 The Structure of the Communicators

The server communicator TCPServerAsync uses an asynchronous operation to accept an incoming connection and asynchronously receives data from a connected socket. Because the server will be used to receive tactical messages and operational messages concurrently, we speed up the processing of the received messages by filtering them into two independent processing queues. The tactical message is the robot position message sent from all connected tables. The number of tactical messages received per second will depend upon the number of connected tables.

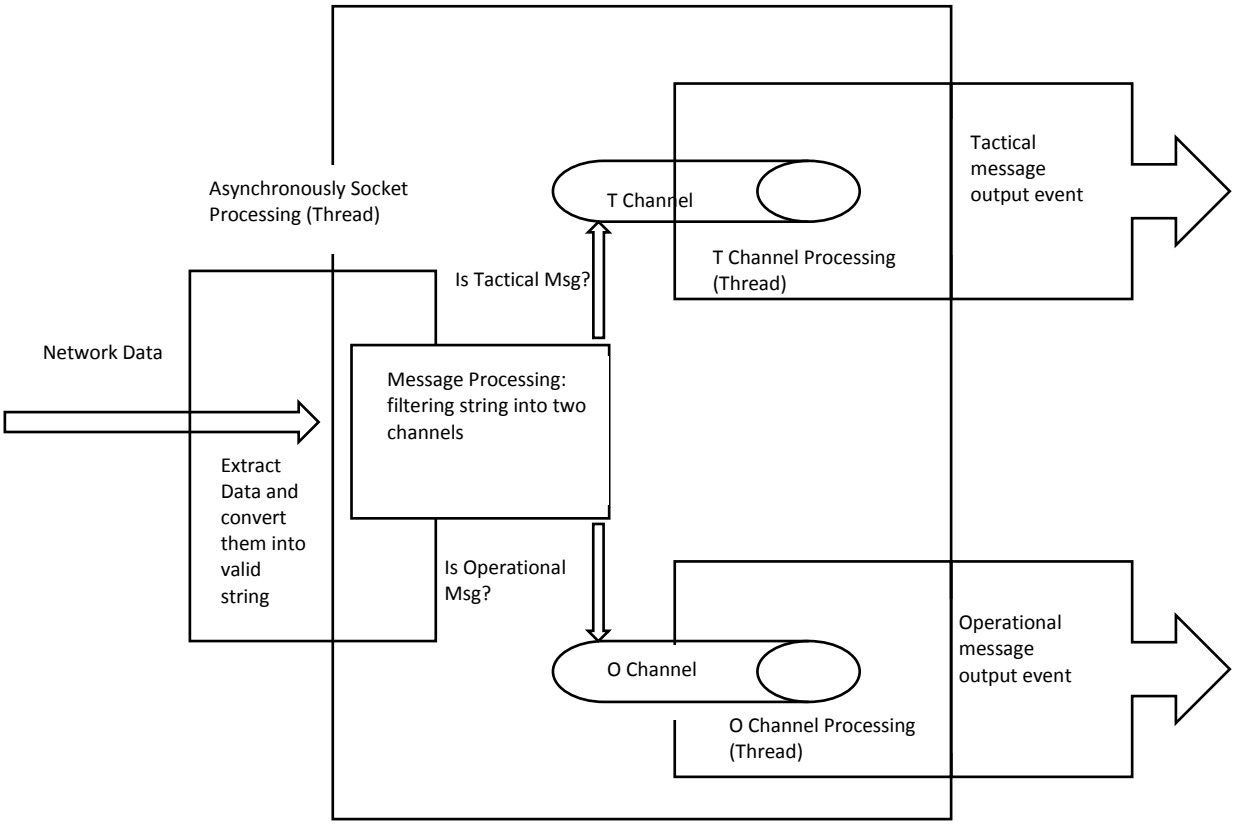


Figure 5-15 Message Processing Technique for the Server Communicator

The incoming buffer size for messages is 8092 bytes, and the size of a position message is much smaller than the buffer size. Hence there can be multiple messages in the message buffer. The buffer is required because each table may send out as many as 100 position messages per second, but each table only has one server to receive all messages from all tables. If the server cannot process the messages quickly enough, the server could be blocked by messages waiting for processing or the server could lose messages. To increase the efficiency of network message processing, we have implemented a batch processing technique using the buffer (see Figure 5-15).

The batch process extracts all messages from the buffer and leaves uncompleted messages to be combined with the next buffer received. The extracted messages are then filtered into two different message queues (channels) based on the message type. One message queue is used for the tactical messages with a limit of 100 items. If the message queue is full while there is still a new message arriving, the message queue discards the first unprocessed item and allows the new message to be added to the queue. This is because the tactical message is used for the movement of the virtual robots, we only need sufficient position information to make movements appear smooth. Another reason is we use one server to receive messages from all tables and hence the mechanism has the drawback of having to handle many messages at the same time. It could cause a message jam on the server side if there are many messages waiting to be processed. For that reason, we have set a maximum limit for the number of tactical message waiting in the queue.

The other message queue is used for operational messages such as control messages, goal messages, Network Toolkit (NT) messages, text messages, peer list messages, state messages and other custom messages. The operational messages are more important than the tactical messages as they are used for system control, state management and communication. Therefore there is no limit to the number of operational messages that can be stored in the operational message queue. The operational queue will make sure that all operational messages are processed.

Having too many message items waiting inside a queue is not acceptable, so we have two independent threads to process them as quickly as possible and pop them out into their event subscribers, i.e. the Network Toolkit.

5.6.2 Network Toolkit Class

The technologies of Peer Name Resolution Protocol (PNRP) and People Near Me Service (PNMS), which we have discussed in section 3.1, can be used to build Peer-To-Peer applications. The PNRP is a part of Windows XP SP2 and the PNMS only ships with Vista. For that reason, the PNRP and the PNMS are not suitable for the Windows 7 environment and thus they are not employed in the Network Toolkit. However, they provide an idea of how to construct a Peer-To-Peer architecture. The idea is that every peer needs to know a few peers nearby in order to connect and communicate with them. We have employed a similar idea in the development of the RoboTable peer-to-peer architecture, in that there must be a table that is known by all other tables, and that table is the master table.

The Network Toolkit is mainly used to manage peers, send messages to all peers and forward messages to the Game Manager or other subscribers. The Network Toolkit maintains a list of peers. The latest peer list is sent out by the master table, which we discussed in section 4.6.2. All tables need to connect to the master table in order to obtain the latest peer list before they can start to play the game. The Network Toolkit will update the peer list and send out the latest peer list directly if it has been set as the master table. The essential control messages and network messages (NT messages) are maintained inside the Network Toolkit. Other messages are forwarded directly to their subscribers i.e. the Game Manager.

The following section discusses some implementation details about the NT Stream and the Network message (Game message).

5.6.3 Network Messages and NT Stream

In chapter 4, we discuss the design of Network Messages and the NT Stream. There are two types of network messages: the operational messages and tactical messages. The position message is the tactical message and the others are operational messages. Each message has the properties:

Document Head, Message Head, Message Body and Message Tail. The NTStream represents the string format of the Network message. An NTStream example is discussed in section 4.6.3.

The base network message class has a method to convert a concrete network message into an NT Stream object and has a method to reconstruct the input NT Stream object back to the original message object. The two methods are:

- CreateNTstreamBody(), transfers the current message properties into an NTsteam object.
- ReconstructMessageBody(), reconstructs the current network message from a given NTstream object.

Each network message inherits from the base network message class.

Figure 5-16 shows a number of network messages that have been implemented.

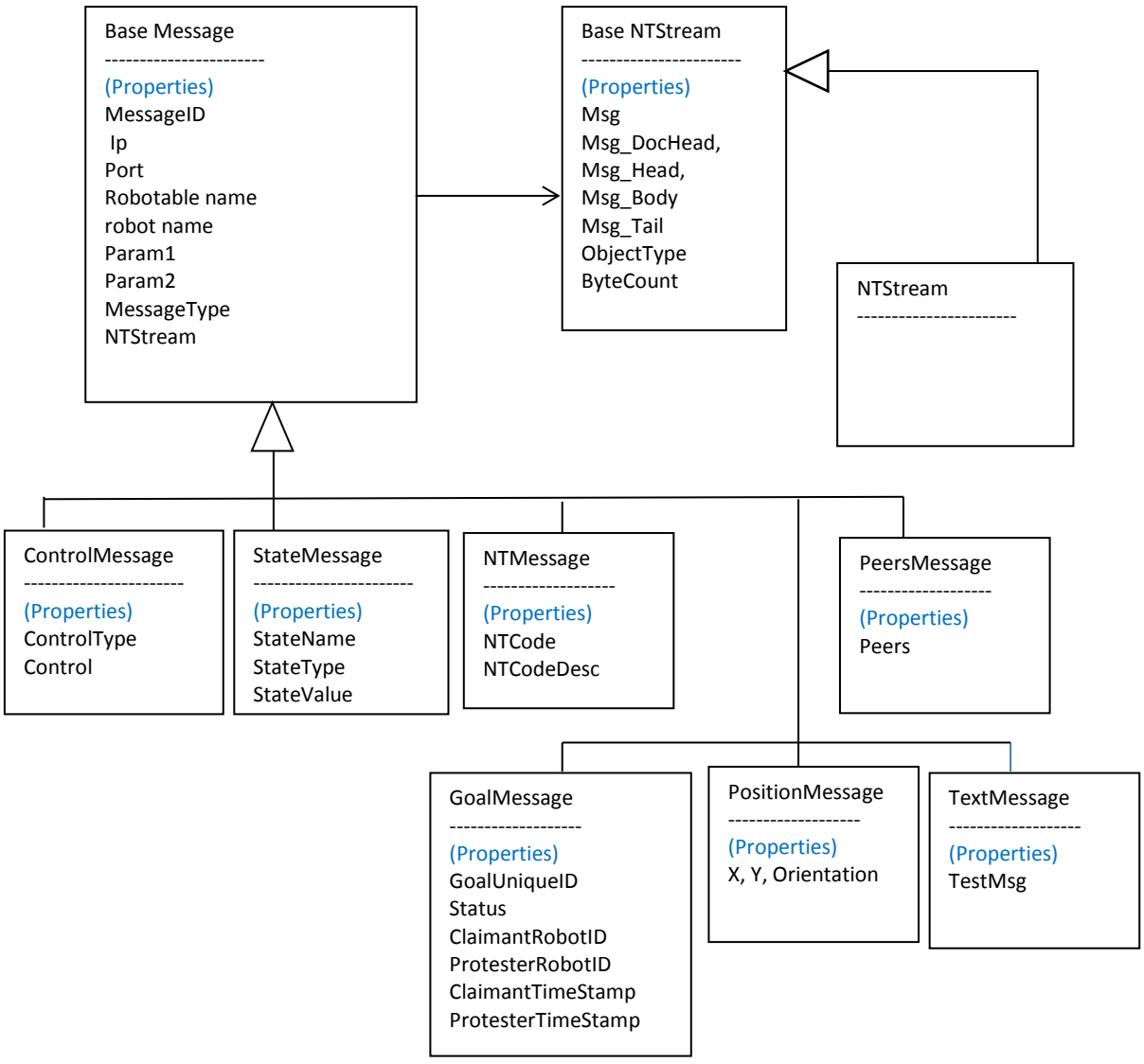


Figure 5-16 Network Message

The NT Stream class is used to convert network messages into network strings and vice versa. Section 4.6.3 discusses the design of the NT Message classes and the NT Stream class. Figure 5-17 outlines the workflow for sending a Network message across the network and reconstructing it at the other end.

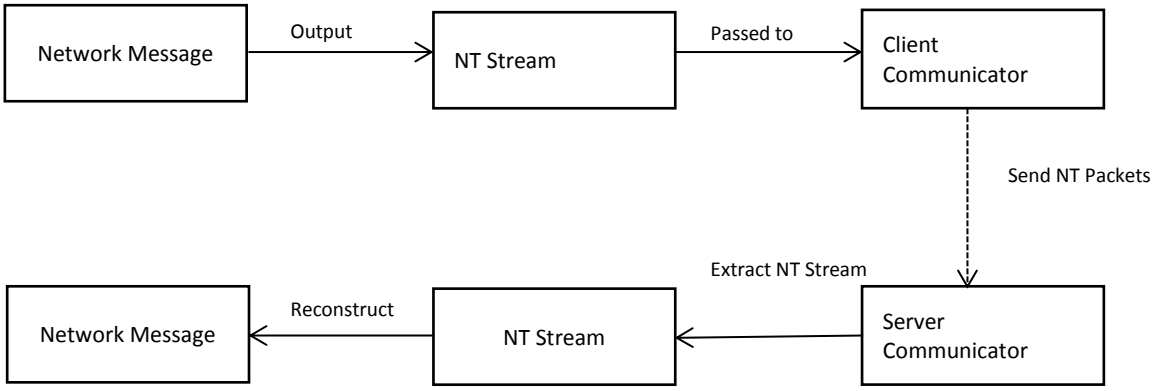


Figure 5-17 Network Message and NT Stream

5.7 The Communication Toolkit

The Communication Toolkit uses SkypeKit. To use SkypeKit, game developers are required to register as the developer of SkypeKit through the website http://developer.skype.com. The key file and the Skype™ runtime software are required to develop a communication application using Skype™. There are many documents and sample programs on the website to help game developers to integrate Skype™ into their application. In order to simplify RoboTable game development, we have implemented the SkypeToolkit class as the Communication Toolkit. To use the toolkit that we have implemented, a user needs to register as a Skype™ user and the participants in a conversation are added into the users contact list through the Skype™ application or website. Implementing Skype account management is future work for the project.

The SkypeToolkit is a class that requires an object of the SkypeFormControl class and the calling form object as parameters. Figure 5-18 outlines the design of the SkypeToolkit. It is an event-driven toolkit. The SkypeFormControl class is simply a collection of form controls that are required for the minimum Skype™ communication requirement. The controls are used to control the process of Skype™ communication such as sending a text, making a call, holding a call or muting a call.

The SkypeToolkit initialises the form controls and downloads the contacts from the Skype™ server. SkypeToolkit uses events to update changes of contacts. A contact becomes a participant when the contact has been added to a conversation. Only participants can communicate with other participants. The SkypeTookit uses events to update incoming messages and incoming conversations. Game developers use the SkypeFormControl class to control the SkypeTookit.

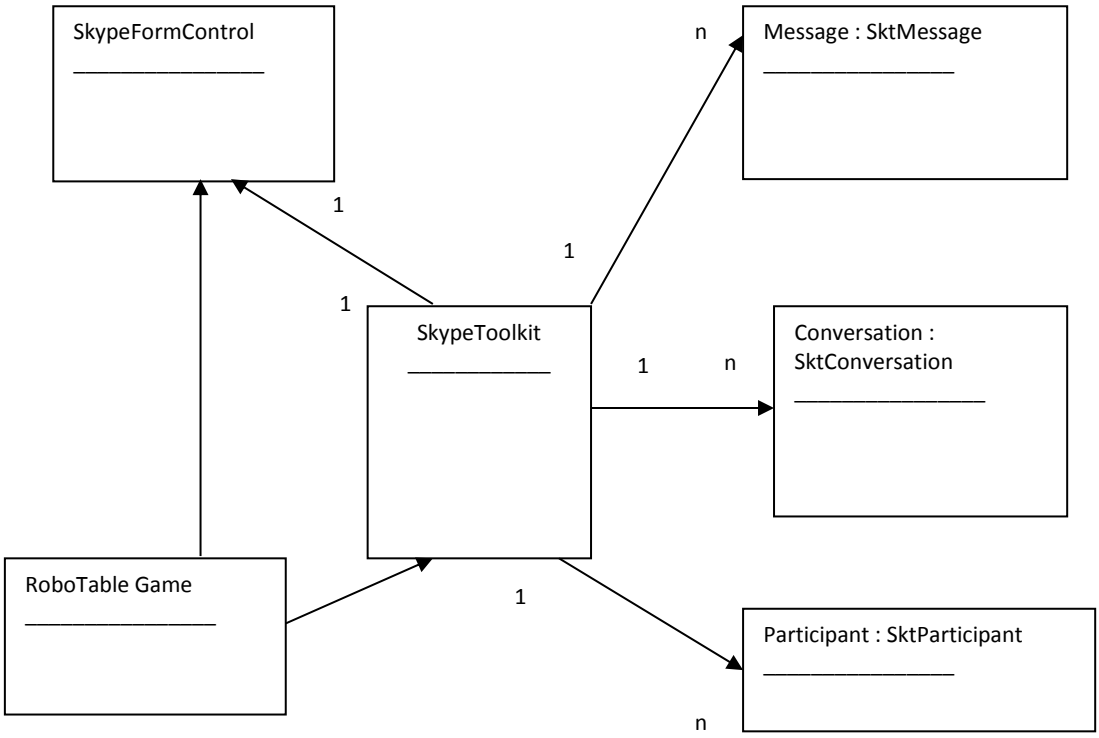


Figure 5-18 Design of the Communication Toolkit

Figure 5-19 presents a sample of the user interface of the Communication Toolkit i.e. the SkypeToolkit.

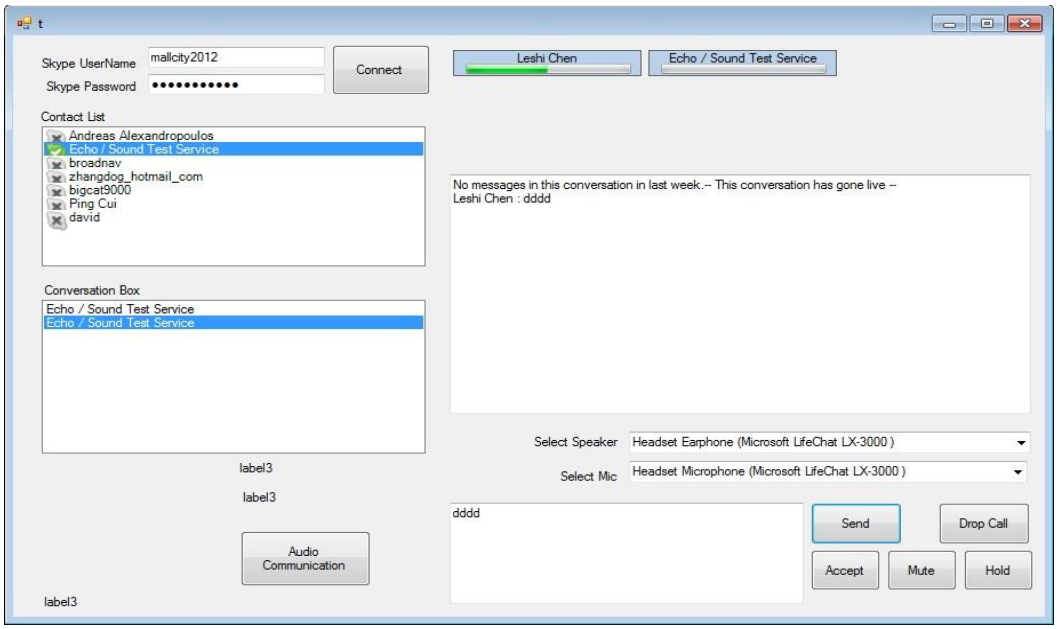


Figure 5-19 A sample Communication Tool Using SkypeToolkit.

SkypeKit provides features for developing a desktop Skype communication tool and using SkypeKit is straightforward.

The toolkits also provide another convenient technique to send text messages to all tables using the Network Toolkit if game players do not have a Skype account. We discuss network messages in section 5.6.3. The message class, called TextMessage, is inherited from the base message class and thus has all the features of other messages. To send a text message to all peers using the TextMessage class, the developer needs to create a TextMessage object and send it using the method provided by the Game Management Toolkit. This can be used by developers who do not wish to integrate more advanced communication functionality into their game.

5.8 Base Game Rule Control Class

The toolkits aim to help game developers develop a robust distributed multiplayer RoboTable Game. A game must have game rules. Developing game rules is the responsibility of the game developer, and this is likely to be the most challenging part for game developers. However, the toolkits provide some basic classes to help game developers start to develop their game rules.

Figure 5-20 Base structure of Game Rules

Figure 5-20 presents a simple structure for game rules. The BaseGameRuleControl and BaseGoalObject are the two abstract classes that the game developers can use. The BaseGameRuleControl and BaseGoalObject classes provide structures to help developers build their own game rules and goal objects. A goal is something that the game players are going to achieve or acquire. For example, in the Land Grabbing game, a piece of land is a goal as players try to collect as many pieces of land as they can. The BaseGameRuleControl class provides general interfaces and utility functions for game developers to create their own game rules.

The BaseGameRuleControl class provides three main interfaces for the game developer to implement.

- PostLocalRobot(Robot robot): add a received robot object into the game rule control class object in order to send goal claim messages.
- PostGoalMessage(GoalMessage mGoalMessage): add a received goal message into the game rule control class object for reconciling game results.
- UpdateRobotList(): add the current Robot list into the game rule control class object. The robot list contains all robot objects, and the game rule control class object can access all robot information easily.

Game rules implementation Processes (Algorithm)

The base processes for implementing game rules are outlined in Figure 5-21 and followed by descriptions of the processes.

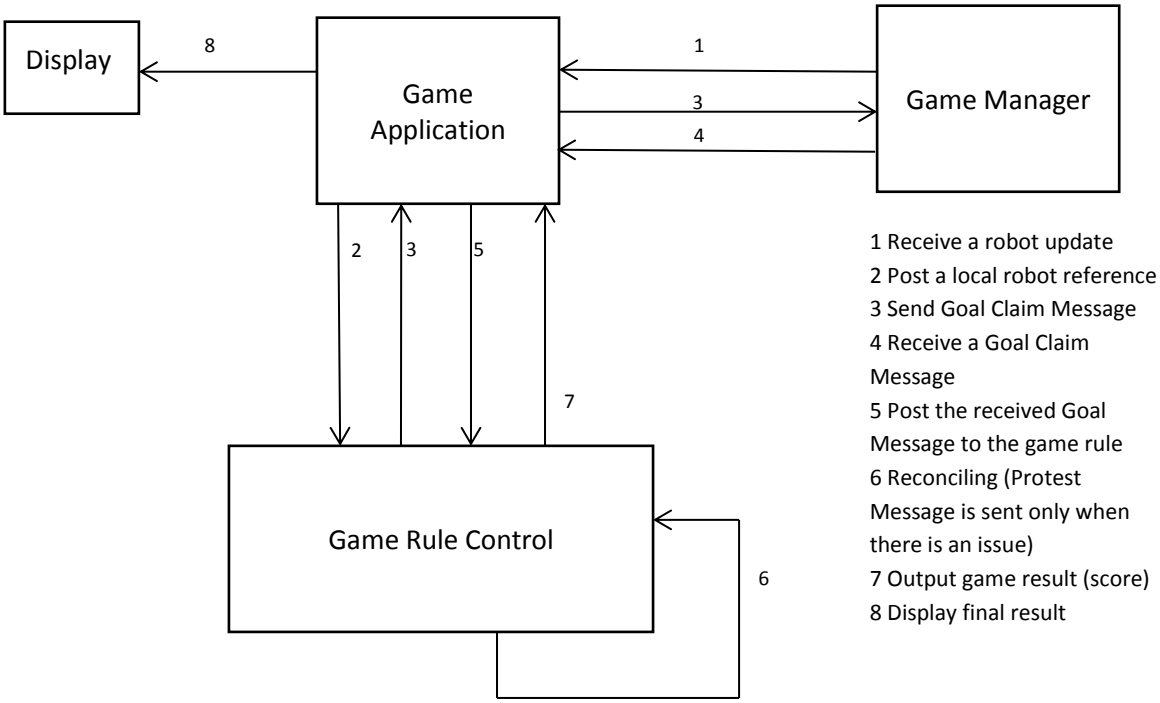


Figure 5-21 Processes for Implementing Game Rules

a) When the game application has received a local Robot Update (Robot Object), the game application adds the robot object to the game rule control class object (Process 1 and Process 2).

b) The game rule control class object checks if the goal object that the local robot tries to claim is available or not. If it is available, the game rule control class object will send a goal claim message to all tables including the local table. Because the local message will pass to the local table directly, all local goal claim messages will be processed first (Process 3).

c) When all tables have received the goal claim message, the tables (Game application) add the received goal claim message to the game rule object. The game rule object will check with its temp box, which is a collection of goal objects that have been marked as temporarily occupied (Process 4 and 5).

d) If a table has received a goal claim message that claims a goal object which is already occupied by another table, the table will compare the timestamp and will send a protest goal message to all tables if the occupied time, which is shown by the timestamp, of the claimant is later than the current table. When all tables have received the protest goal message, they all need to update the ownership of the goal to the protester (Process 6). A protest goal message is required for other tables to correct the ownership of the goal object.

e) If the occupied time of the claimant is earlier than the current table, the current table needs to correct this by changing the ownership of the goal object from itself to the claimant i.e. the current table loses the ownership of the goal object (Process 6).

f) If the goal object is not occupied, all tables need to update the new owner and move the goal object to the temp box. The goal objects inside the temp box can be moved to the permanent box after a certain elapsed time (Process 6). The certain elapsed time is used to tolerate some issues such as network latency.

g) The final result will be reported based on the goal objects in the permanent box (Process 7).

h) The final output result will be passed to the RoboTable display for reporting (Process 8).

Thread Safety

Because the game rule control class object is independent of the main control form and uses multithreading, we need to be careful to avoid issues such as the main control form and the game rule control class accessing an object concurrently. Hence, thread safety is an important issue. There are three objects that will be accessed by multiple threads when using the BaseGameRuleControl class and they are: GoalObject collection, TempOccupiedGoalObjects collection and FinalOccupiedGoalObjects collection.

The BaseGameRuleControl class has two independent threads, one is the main thread to update the graphics rendering via the Game Management Toolkit and to check the game finish condition. The Game Management Toolkit manages the graphics rendering loop and provides a generic graphics interface to game developers. Hence, the main thread uses the graphics interface to update image objects' properties such location, rotation, colour and size. The other thread is used to send goal claim messages and to reconcile goals. The reason we implemented two threads for the BaseGameRuleControl class is because graphics updates may require more time than the reconciling process which has a strict time requirement. Hence, having two independent threads to process graphics updates and goal reconciling is a good idea for the base game rule control class.

5.9 Game Developer Guide

We have developed a skeleton project and a set of instructions to help game developers use the toolkits. A game developer guide using the skeleton project is presented in Appendix B, which includes an introduction to the project and development process tutorial. The skeleton project contains all necessary sample methods for game developers to learn and use to develop their games.

5.10 Implementation Summary

In order to meet the proposed requirements, we have developed toolkits to simplify the process of RoboTable game development. The Robot Tracking Toolkit tracks the position of a robot and sends position information to all tables. The Network Toolkit handles all network communication requirements and sends/receives messages to/from all peers. The Game Management Toolkit is the main toolkit that game developers need to integrate into their applications. It incorporates other toolkits and provides generic interfaces to game developers. The Communication Toolkit uses SkypeKit and provides a tool (SkypeToolkit) to help game developers to integrate Skype™ into their desktop applications. The following chapter discusses the testing and evaluation of the toolkits.

Chapter 6
Testing and Evaluation

We have developed toolkits to support distributed multiplayer RoboTable games as described in chapter 5. In this chapter, we discuss the evaluation of the toolkits to determine their performance with respect to the requirements outlined in section 6.1.

6.1 Evaluation Requirements

The experiments we conducted are based on a game developer evaluation and a multiplayer game scenario. The baseline performance measures are benchmarks to compare the results of the experiments related to the multiplayer game scenario evaluation. The requirements for the toolkits are as follows:

1. Using the toolkits should be straightforward

   The term “straightforward” means that the toolkits can be used to create a distributed multiplayer RoboTable game easily and game developers do not need to create their own network communication and robot tracking functionalities.

2. Real-Time performance

   The term “acceptable” for Real-Time performance is that the robot update rate should be higher than the video frame rate of 24 position updates per second (41.7 milliseconds per update).

3. Distributed multiplayer performance

   The term “acceptable” for distributed multiplayer performance means that a maximum of 4 tables should run concurrently and the toolkits should still provide Real-Time performance. Otherwise, it is “unacceptable”.

4. Tracking subsystem performance

   The term “acceptable” for the tracking subsystem performance is that the tracking system should produce position updates consistently and will not be affected by robot speed.

6.2 Evaluations

To evaluate the toolkits, we have undertaken a number of assessments. The first approach was a case study of the game development process using the toolkits. The second approach was to establish baseline performance benchmarks for the system. The third approach was to carry out

experiments to evaluate their real-world performance and the scalability of the toolkits using the game created in the case study.

6.2.1 Game Developer Evaluation

To evaluate the toolkits for game developers, we used the developer manual, which is discussed in Appendix C, to develop a Land Grabbing game. The flow chart for developing game rules is discussed in section 5.8. The development of the game was evaluated as a case study, which is discussed in section 6.3. The case study introduces the concepts of the Land Grabbing Game and describes how to implement it from scratch. After we implemented the Land Grabbing Game, we applied the game to a multiplayer scenario evaluation. The requirement for the case study is that developing a new distributed multiplayer RoboTable game using the toolkits should be straightforward. The reason we did the game developer evaluation first is that we will use the Land Grabbing game to do the multiplayer scenario evaluation later. The multiplayer scenario evaluation tested the performance of the Land Grabbing game against the performance requirements outlined in section 6.1.

6.2.2 Benchmark Measures for Multiplayer Scenario Evaluation

In order to establish performance benchmarks for the toolkits, a group of baseline performance measurements are required. The following have been used:

- Average Robot Speed
- Tracking Rate at Maximum Speed
- Tracking Rate at Rest

In order to determine the performance of the tracking subsystem, we performed two experiments. The first experiment tested the tracking rate with the robot at rest and the second tested the tracking system with the robot running at maximum speed. The reason we did both experiments is that we want to find out whether the robot speed affects the performance of the tracking hardware (Wiimote).

To measure the performance of the tracking subsystem, we implemented a logging system to log timestamps as data flowed through each part of the subsystem. It is acknowledged that this logging may reduce the performance of the toolkits. However, if the performance with logging is acceptable, we can be confident that performance without logging will also be acceptable. The values we have measured are below.

- Time between Wiimote updates
- Time to calculate position and orientation

- Time to send to network interface
- Time to receive from network interface
- Time to update memory to store new robot position
- Time to update robot display

From these benchmark measures, we can evaluate multiplayer scenarios and measure changes in performance.

6.2.3 Multiplayer Scenario Evaluation

In order to evaluate the scalability of the toolkits, we ran a series of multiplayer scenarios to measure the impact on performance. The game we used to do this evaluation is the Land Grabbing game we created in the case study.

Because each table tracks a robot locally and displays the position of remote robots virtually, we had a single physical RoboTable in full operating mode and between 2 to 4 simulated tables sending position updates for their robots. We were primarily interested in changes in performance by adding extra tables one by one to determine the point at which performance becomes unacceptable. We compare the benchmark measures results with the requirement of the benchmarks.

6.2.4 The Equipment

The main equipment used to conduct the evaluation is outlined in Table 6-1. We used the game that we developed for the case study, to test base line performance. We added performance logs to the toolkits to record the start and end time for different activities such as graphic rendering, sending and receiving messages, robot tracking and Network processing. The logs use the Microsoft trace function (see Appendix D).

Table 6-1 Equipment for Evaluation

Equipment	Configuration
RoboTable	A semi-transparent Table top with a 45 sloped mirror and a stable steel frame.
Wiimote controller	Resolution is 1,024 × 768
IR pen	
Bluetooth dongle	
Projector	Resolution is 1,024 × 768
LEGO™ Robot	Mindstorms NXT2.0 controller, fresh batteries

PCs	32-bit Windows 7 Operating System Supporting DirectX11. Administrator permission is required for each PC
Microphone	
Network	Internal Network at Lincoln University, 100 Mb/s
Testing Platform	The Land Grabbing Game

6.3 Case Study: Developing a Multiplayer RoboTable Game Using the Toolkits

6.3.1 Overview

The Land Grabbing game is a multiplayer RoboTable game developed for this case study. It contains a number of rectangular land objects as shown in Figure 6-1. Robots on different tables run concurrently and try to grab (run over land objects) as many land objects as possible in order to win the game. Each robot has its own colour and when a robot grabs a land object, the colour of the land object will be changed to the colour of the robot. When the game play time exceeds 5 minutes, or there are no more land objects available for capturing, the game will finish and the final result reporting the number of objects grabbed by each robot, will be displayed.

The game map, which is shown in Figure 6-1, is a uniform grid of rectangles of 50 (5 Rows and 10 Columns). The background colour of each square at the start is white, and the frame colour of each square is blue. The square is the land object that robots will try to grab.

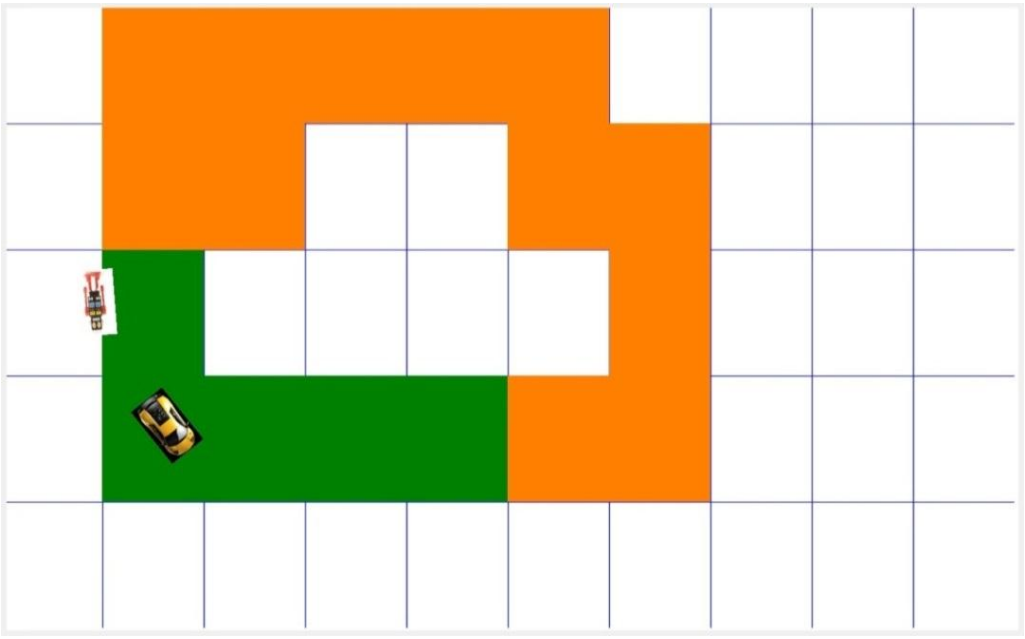


Figure 6-1 Land Grabbing Game Map

The Land Grabbing Game contains two main components i.e. the control form, which integrates with the toolkits and display the game output as shown in Figure 6-1, and the game rules, which check which robot wins a land object and when the game should stop. Hence, the development of the Land Grabbing Game will be separated into two main parts:

1. The first part is to implement the game body, which comprises a control form, a connection form and a communication form.
    a. The control form is a form class that contains the main controls for game players to manage a game i.e. start the game, play the game and abort the game.
    b. The connection form is a simple form class for game players to enter a remote master table's IP address and Port number in order to connect the local table with the remote master table.
    c. The communication form is a form class that provides Skype facilities to game players to talk with other tables.
2. The second part is to implement a game rule control class and a goal object class.
    a. The game rule control class is a class that game developers are required to implement in order to control their game rules. One of the main game rules is to determine how to assign a goal object to a robot.
    b. The goal object class is a class that represents a goal that game players need to complete in order to win a game.

Because the connection form and the communication form provide functions that are the same as all other multiplayer RoboTable games (developed using the toolkits), we use the connection form and communication form from the skeleton project provided by the toolkits directly. Hence the case study mainly focuses on the development of the game control form and game rules.

The game rules of the Land Grabbing game are outlined as below:

- If a robot is inside a square, the game will try to claim the robot as the new owner of the square.
    - If the square that a robot is trying to claim is already owned by another robot, the game makes no changes.
    - If there are two robots trying to claim the ownership of a square simultaneously, the game will check the timestamp of the claim message. The first claimant becomes the owner of the square.

- If the total play time exceeds 5 minutes or all squares are claimed, the game will finish.
- The final results will be displayed on all tables.
- The user can abort the game, drop the game or terminate the game at any time.

6.3.2 An overview of the implementation processes

The processes listed below may be divided into two parts. The first part is to implement a game control system, and the second part is to implement game rules. Processes 1 to 3 explain the implementation of a game control system. Process 4 discusses the implementation of the game rules and process 5 explains how to add the game rules to the game control system.

1. Design Land Grabbing Game body structure
2. Design game flow
3. Design and implement the main Land Grabbing Game user interface i.e. the control form
4. Design and implement a game rule control class and a goal object class
5. Add the game rule control class object to the control form and integrate it with the game loop

The following sections explain the implementation processes in detail.

6.3.3 Processes

Process 1: Land Grabbing Game Body Structure

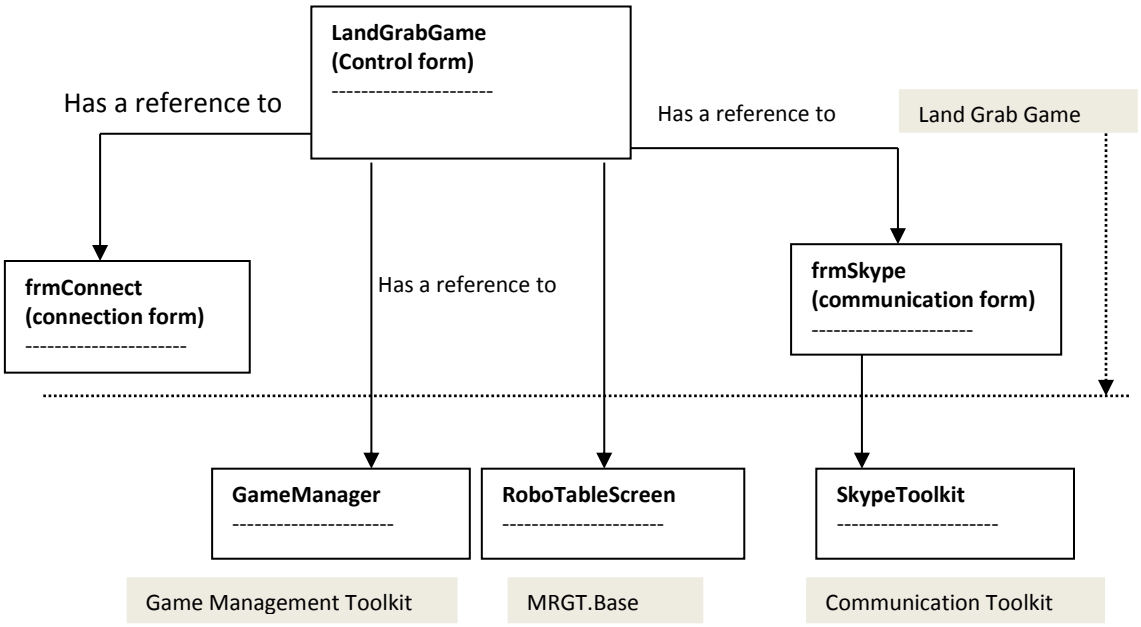


Figure 6-2 The Land Grab Game Structure

Figure 6-2 presents the basic structure of the Land Grabbing Game. The description of this structure is as follow.

- LandGrabGame is the main form class that a game player can use to control the game. The form is created by game developers or game developers can use the existing control form from the skeleton project provided by the toolkits but some modifications are required.
- frmConnect is a basic form class that allows users to connect to a remote master table. Game developers can directly use the existing connection form from the skeleton project provided by the toolkits without change.
- The frmSkype class integrates the communication toolkit to provide text chat and audio chat via SkypeKit components. Game developers can directly use the existing communication form from the skeleton project provided by the toolkits without change.
- Game developers need to integrate their games with the Game Manager class. Appendix C provides detailed documentation about this.
- The RoboTableScreen class is used to display the game output on the RoboTable or a second monitor. The toolkits provide this and game developers can use this class directly.

Process 2: Game Flow

To control the game flow, we only need to start the game engine (GameManager.Begin()), initialise the game engine (GameManager.Initialise()), wait for all tables to finish preparation, which includes connecting with the master table, and offline activities such as building robots. The game engine will start the system state management routine when the Begin() method is called. The Initialise method is required to tell the engine that information has been input to the game engine and the game should transition to the next game state i.e. Game Preparing State.

The game flow is illustrated in Figure 6-3. A game finishes when the goal conditions are met or when the game is aborted by a player.

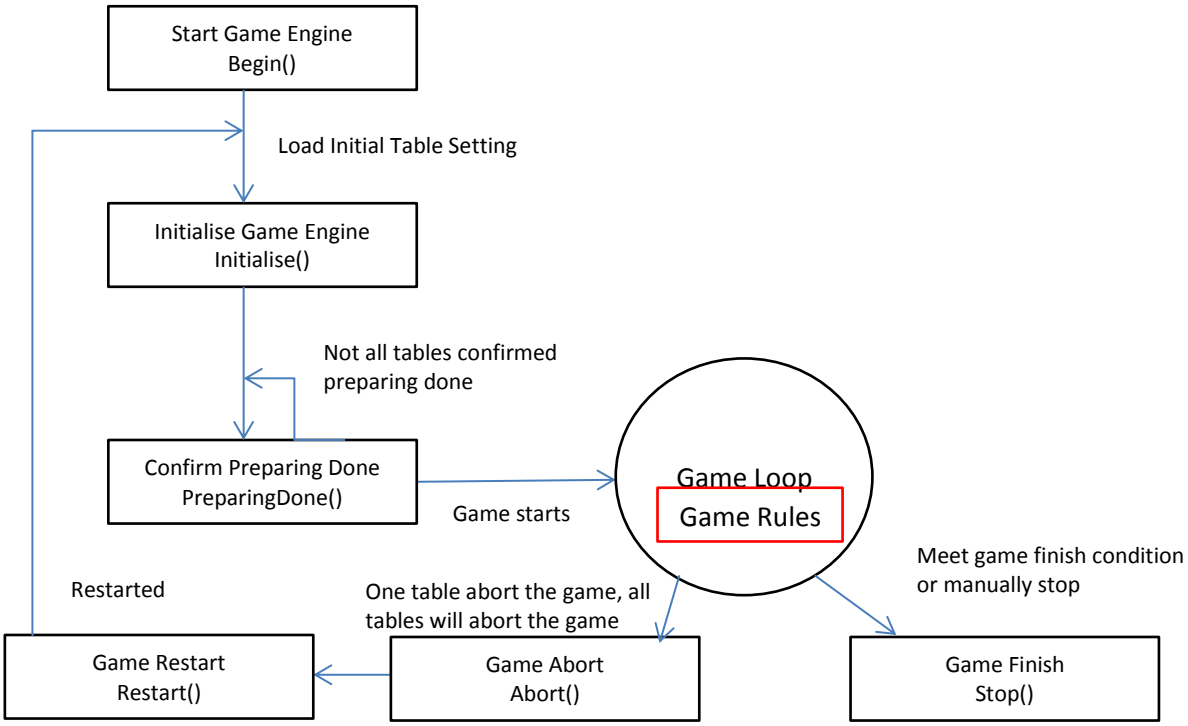


Figure 6-3 Basic game flow

The detailed game flow for this case study is outlined below.

1. A user starts a game with the main control form
2. The main control form creates a reference to the GameManager class and starts the GameManager engine by calling the Begin() method.
3. The main control form also creates a reference to the custom designed and implemented game rule control class. The game rule control class is a class that manages all game rules.
4. The main control form sets initial values for the game rule object and uses the game rule object to create the game map (load game map) and goal objects.
5. The user enters a local IP address and port number, selects if it is a master table or normal table, selects if it is a virtual table or real table, selects a robot image and selects a robot colour.
6. The user starts the game by initialising the game i.e. loads the initial Table Setting.
7. The main control form loads the Table Setting into the game manager object and calls the initialise() method of the game manager object.
8. The main control form will pop up a connection form to ask the user to enter the remote master table's IP address and port number if the table is not a master table.

9. The main control form will ask the user to calibrate the real RoboTable screen if it is not selected as a virtual table.

10. When a user has been connected with a master table and completed all preparations such as placing a robot on a start point, the user confirms that preparations are complete (Master table will be same as normal table).

11. The main control form calls the preparingdone() method of the game manager object.

12. When all tables confirm preparations are complete, the game starts and enters to the countdown state where a countdown timer pops up.

13. When the countdown state counts down to 0 seconds, the game play automatically starts.

14. When game play starts, all robots are required to be switched on manually. The game will communicate with the game rule control class so that game goals can be accumulated and the game result can be reported when the game is finished or aborted.

15. When a user clicks the abort button from the main control form, the game will be aborted.

16. When the game meets any game finish condition, the game will stop.

17. Final results will be displayed on the screen.

Process 3: Design and Implement Land Grabbing Game User Interfaces

In this case study, there were three form objects required, namely the main control form (LandGrabGame), the form for connecting to the master table (frmConnect) and the form for controlling the communication toolkit (frmSkype).

The design of the main control form is shown in Figure 6-4. It has five small text boxes for the local IP address and port number, which are populated automatically by the system, and two text boxes for the Robot name and RoboTable name. It also contains a number of button controls. One button is used to pick a colour for the owner land, and two radio buttons are used to determine if the game is a master table. If it is the master table, no remote connection is required. The checkbox “virtual table only” indicates if the RoboTable is a virtual or real table. The virtual table uses pre-recorded data to simulate the movement of a robot. However, the physical table uses the tracking toolkit to detect the movement of the real robot. The virtual table uses a picture box to display the game output. In contrast, the physical table uses a projector to display the game output on a RoboTable. The list box under the radio button is a list of Robot Images available for game users to choose.

The Main Control Form(LandGrabGame)

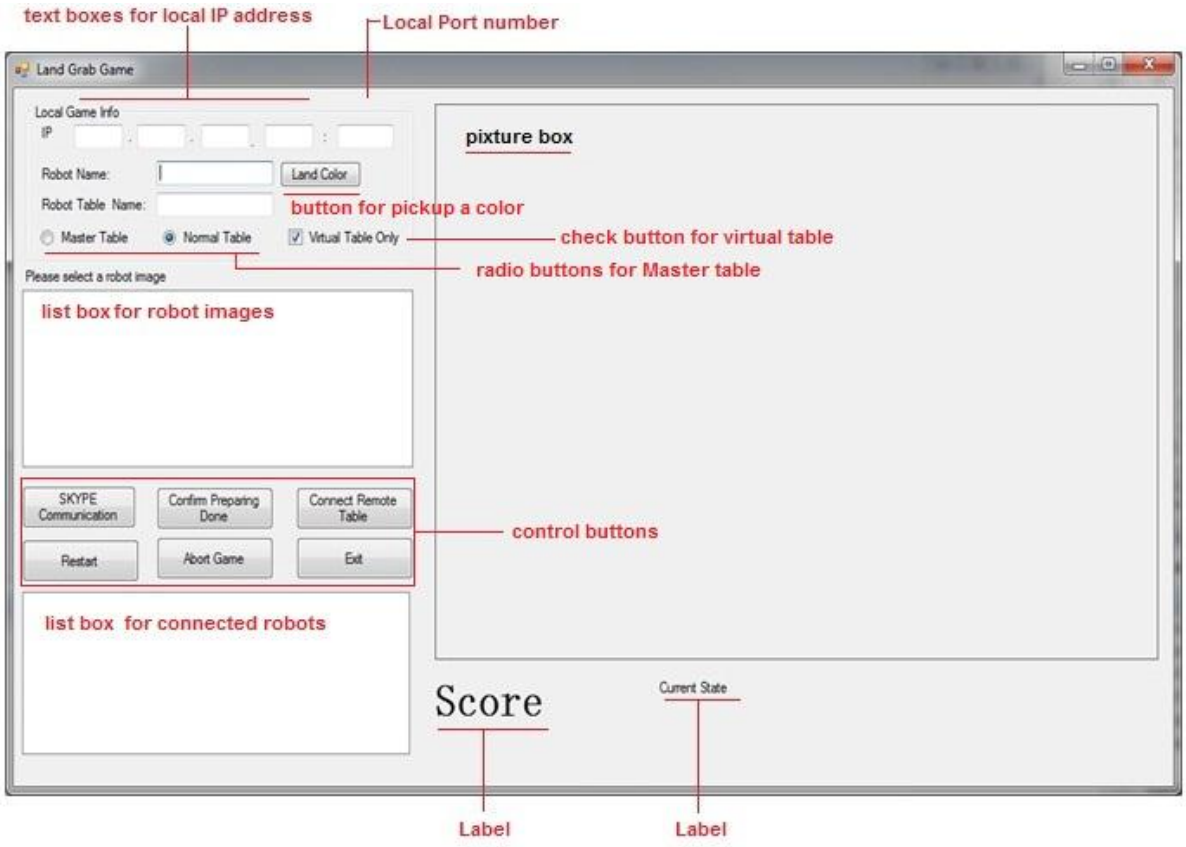


Figure 6-4 Main control form for the Land Grabbing Game

The implementation of this control form follows the game flow and hence is straightforward. Basically what we have done is use event methods to receive robot updates and buttons to control the game flow. The toolkits will manage the game output and system state.

The Connection Form

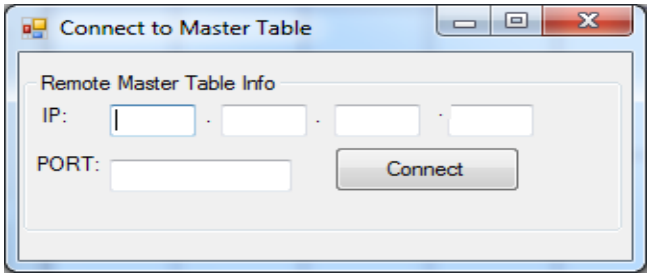


Figure 6-5 The Connect Form for connecting the remote master table

The connection form requests users to enter the IP address and port number of the remote master table. Figure 6-5 presents a design for this form. We used the form provided by the Skeleton project (Appendix B).

The Communication Form

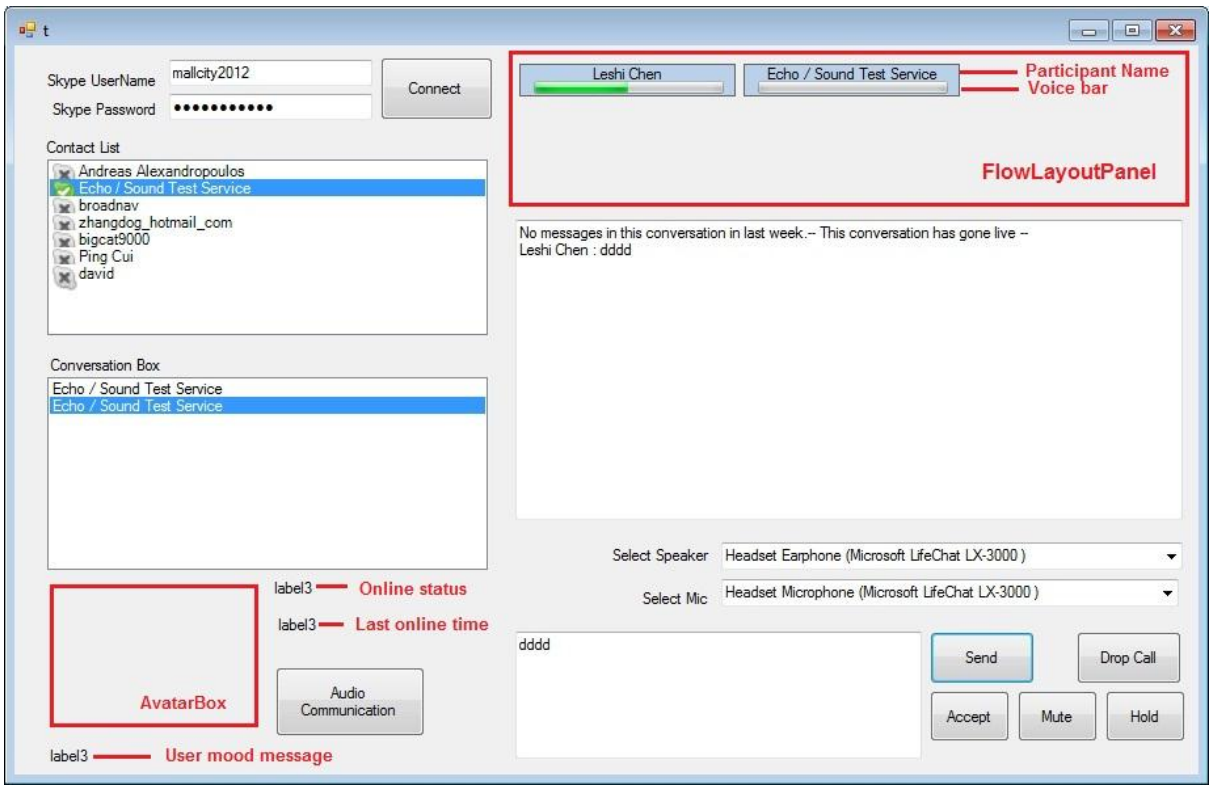


Figure 6-6 Communication form using SkypeKit

The design of the communication form is shown in Figure 6-6. This form is used with the communication toolkit. It contains two single line text boxes for the username and password, and two list boxes for contacts and conversation members. One button is used for connecting to Skype™, and there is one picture box for the avatar icon. Three labels are used for displaying the online status, the time last online and the user mood message. The control at the top right of the form is a FlowLayoutPanel, which is used for displaying the name of the participants and the voice bar in a conversation. A multiple-line text box is used to display the messages for log and text communication. There are two combo selectors and they are used for selecting a speaker and microphone. Another multiple-line text box is used as input for text communication. There are five buttons for communication control namely Send for sending text messages, Call for making an audio call, Accept for accepting an audio call, Hold for holding the current call and Mute for muting the microphone. AvatarBox is a picture box for showing a user image. We used the form provided by the Skeleton project.

Process 4: Game Rules

The following describes the processes for game developers to implement their own game rules.

- Create a number of goal objects. A goal object relates to a location on the game map and has a score or action associated with it. Goal object use the location on the game map as their key in a collection.

- Read the latest position information of a local robot and check if the local robot can complete a goal object.

- If the local robot can achieve a goal object, the goal object should set the robot as the temporary owner. If there is an issue such as two robots trying to claim the same goal object, a protest goal message should be sent to all tables to ask for the correction of the temporary owner.

- If the reconcile time is up and there are no more issues, the temporary owner of the goal will be changed to the permanent owner of the goal.

- The final result is calculated based on the permanently-owned goal objects.

For example, there are four apples located in a position of (1,1), (5,5), (10,10) and (20,20). Hence an apple with the key as its position is a goal object. If a robot runs over the location of (10, 10), the robot will win the apple. If two robots claim the same apple at (10, 10), they might need to do goal reconciling by checking their claim message time stamps. The first robot that runs over the position (10, 10) will win the apple.

Processes 3 and 4 employ a goal reconciling technique. If the game does not require reconciling, game developers can skip these processes and mark the claimed goal object to be permanently claimed by the robot.

The flowchart that outlines the implementation of game rules has been discussed in section 5.8.

The toolkits provide some basic classes to help game developers start to develop their game rules.

Figure 6-7 Structure of Game Rules

A simple structure for game rules is presented in Figure 6-7. The BaseGameRuleControl and BaseGoalObject are the two abstract classes that game developers can use. BaseGameRuleControl and BaseGoalObject classes provide basic structures to help developers build their own game rules and goal objects. A goal object is something that the players are going to play with. For example, in

the Land Grabbing game, a piece of land is a goal object. The BaseGameRuleControl class provides general interfaces and utility functions for game developers to create their own game rules.

The BaseGameRuleControl class provides three main methods for a game developer to implement.

- PostLocalRobot(Robot robot): add a received robot object into the game rule object in order to send goal claim message.
- PostGoalMessage(GoalMessage mGoalMessage): add a received goal message into the game rule object for reconciling game results.
- UpdateRobotList(): add the current Robot list into the Game rule object. The robot list contains all robot objects, and the game rule can access all robot information easily.

To implement goal objects and game rules for the Land Grabbing Game, we created a sub-class of BaseGoalObject for a goal in the game and a sub-class of BaseGameRuleControl for controlling the Land Grabbing Game rules. The goal of the Land Grabbing Game is the land object.

Land Grabbing Game rules

Based on the outline above, we have implemented a class named LandGrabRule, which is inherited from the base game rule control class to handle game rules for the Land Grabbing game. The LandGrabRule object has the following properties:

- The number of land columns and land rows, which gives the total number of land objects we need.
- The reconcile time, which is the minimum waiting time in seconds until a land object is claimed permanently.
- A method to dynamically create a game map based on the number of land columns and land rows. The goal objects are also created in this method.
- A method to update the graphics objects in the graphics Utility layer so that land that has changed colour can be displayed.
- A method to enforce game rules.

The structure of the game rules is shown in Figure 6-8.

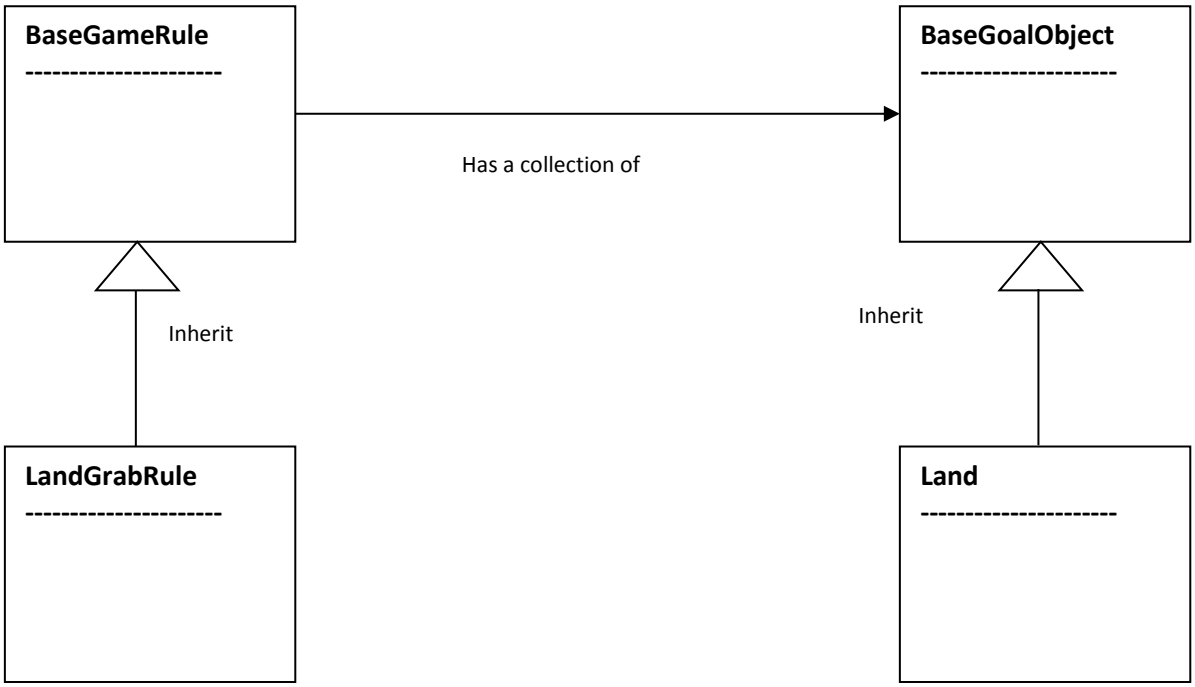


Figure 6-8 Land Grab Rule structure

To implement the goals in a sub-class of BaseGoalObject we provided an overriding implementation of SetOccupied. The constructor of the BaseGoalObject is *BaseGoalObject(double mGoalScore)*. mGoalScore is the score for your goal object.

The SetOccupied method is called if that goal object is occupied or achieved. It should update internal properties and SetOwnership to the robot passed in. An implementation of the Goal class "Land" from the Land Grabbing game is shown in Table 6-2. The method CreateLandImage() is a method that is locally implemented for the Land class to create a Bitmap image for the Land i.e. to draw an image based on the RectangleF object, with initial colour, passed in. The image object will be passed to the Game Utility layer (Appendix C) when the method CreateGameMapAndGoalObjects of the LandGrabRule object has been called (Appendix B).

The Land class is outlined in Table 6-2. The methods and properties provided by the BaseGoalObject class can be found in Appendix C.

Table 6-2 Land Class

Class Land inherited from BaseGoalObject		
Initialising parameters: RectangleF mLand, double mGoalScore, Color TempOccupiedColor, Color UnoccupiedColor		
Own Properties	**Type**	**Description**
LandBitMap	Bitmap	Hold a reference to the bitmap for this land

GetLand	RectangleF	Hold a rectangle representing the land
CurrentColor	Colour	Colour of the land
UnoccupiedColor	Colour	The default colour
OccupiedColor	Colour	The occupied Colour
TempOccupiedColor	Colour	Temp Occupied Colour

Constructor

public GoalObject(RectangleF mLand, double mGoalScore, Color UnoccupiedColor): base(mGoalScore)

The constructor requires a RectangleF object as a Land object, the goal value and the initial Land colour as the initial parameters for the goal object.

Methods

SetOccupied (Robot robot, DateTime CurrentTime)
the method is overridden from the inherited based class

```
try{
  base.SetOwnerShipTo(robot.RobotID, CurrentTime);
if (robot.RobotColor != null)
   _OccupiedColor = robot.RobotColor;
   _CurrentColor = _OccupiedColor;
   return true;
 }catch(Exception e)
 {
   Utility.MTLogger.Error(LogType.Debug, e, "Land -
SetOccupied");
 return false;
 }
```

CreateLandImage()
the method is the local method of the Land class. This method is used to create your own image for this goal object. i.e. if it is an apple, you need to draw an apple or load an apple image.

Used to draw the land's bitmap.

```
if (_LandBitmap != null) _LandBitmap.Dispose();
_LandBitmap = null;
_LandBitmap = new Bitmap((int)this.GetLand.Width,
(int)this.GetLand.Height);
Graphics Tempg = Graphics.FromImage(_LandBitmap);
SolidBrush b = new SolidBrush(this.CurrentColor);
Tempg.Clear(Color.White);
Tempg.Clip = new Region(this.GetLand);
Tempg.FillRectangle(b, this.GetLand);
Tempg.Dispose();
b.Dispose();
```

Once we have implemented the goal objects we need to implement a sub-class of BaseGameRuleControl for the Land Grabbing game. The LandGrabRule is the sub-class of the BaseGameRuleControl class and its definition is outlined in Table B.2 in Appendix B. The methods and properties provided by the BaseGameRuleControl class can be found in Appendix C.

The base game rule control class uses two threads and has been discussed in section 5.8. Hence, the implementation of the game rule control class will bear thread safety in mind and make sure one thread only can access the above collection each time. We use a *lock(){…}* statement to prevent multithreading problems.

The constructor of the BaseGameRuleControl class is *BaseGameRuleControl(GameManager gamemanager)*. The parameter is a reference to the GameManager object that is created in the main game control form. The reason for this is that the game rule control class will need the methods provided by the GameManager class to control the game.

The methods of DoActions, AddToAGoalObjectCollection, PostLocalRobot, CreateGameMapAndGoalObjects and UpdateGraphics are abstract methods of the BaseGameRuleControl class, and hence these methods must be implemented. Other local methods are also created in our example. getLandInfo and getCurrentLand are examples.

The sample codes, which are shown in the methods UpdateGraphics(), which changes the colour of a Land object in the game map, and CreateGameMapAndGoalObjects, which creates a game map and a collection of goal objects, may look complicated. However, the algorithms for these two methods are straightforward.

The UpdateGraphics() method firstly merges the temporary goal list and the permanent goal list into a temporary list and then changes the colour of the object in the temporary list according to the colour of the owner. When an object has been successfully painted, a flag will be updated to the original object in the permanent goal list or temporary list. The following sample code is used to make a hidden graphic object visible and change the colour of the graphic object.

```
this.GameManager.ShowGraphicObject(DefaultRoboTableDrawingLayers.GameUtilit
y, pair.Key);

this.GameManager.ChangeGraphicObjectColor(DefaultRoboTableDrawingLayers.Gam
eUtility, pair.Key, ((Land)pair.Value).CurrentColor);
```

The basic idea for the CreateGameMapAndGoalObjects() method is to create the game map and land goal objects, and save the goal objects in the initial collection i.e. GoalObjects. The following sample code is used to add a new graphics object i.e. the new created Land object, to the game Utility layer, and then translate the new created graphic object to its original location. Initially, all new graphics objects will be temporarily hidden.

```
this.GameManager.AddGraphicObject(DefaultRoboTableDrawingLayers.GameUtility
, newL.LandBitmap, newL.KEY);

this.GameManager.ChangeGraphicObjectLocation(DefaultRoboTableDrawingLayers.
GameUtility, newL.KEY, newL.GetLand.Location);

this.GameManager.HideGraphicObject(DefaultRoboTableDrawingLayers.GameUtilit
y, newL.KEY);
```

The code UpdateGameMap((Bitmap)landmap.Clone()) is used to update the game map using the method provided by the base class.

The game result is controlled by the base game rule control class. The method for displaying the final game result is DisplayFinalResult(), which is called by the DoActions() method (Table 8) when the finishing condition is matched. Figure 6-9 shows the final result of the Land Grabbing Game.

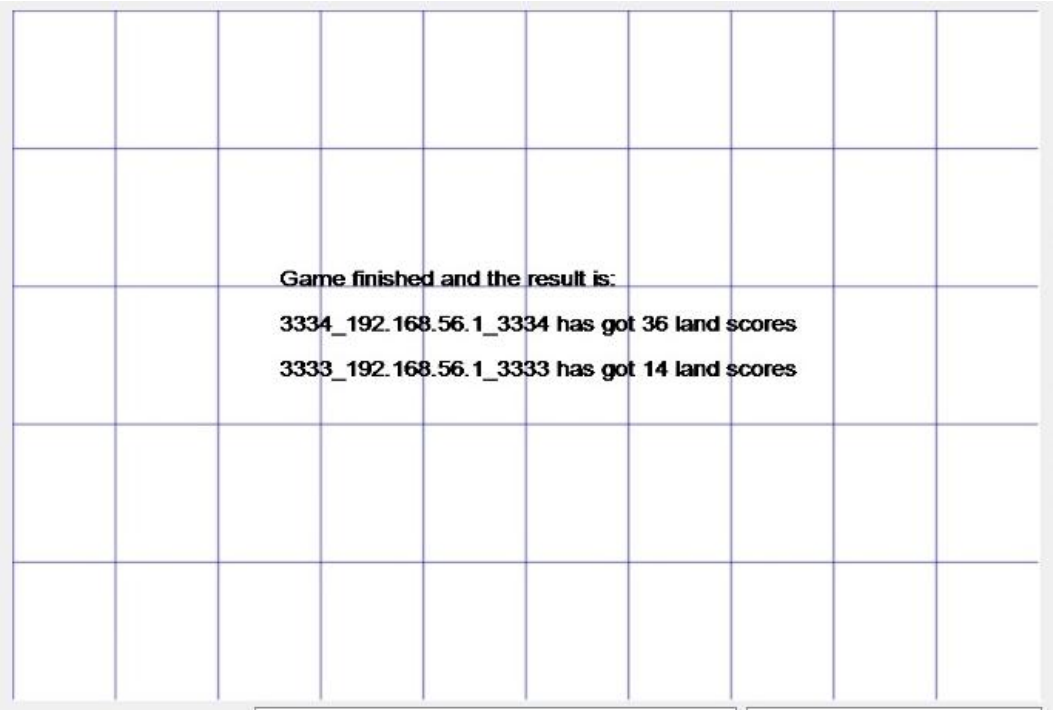


Figure 6-9 The Final Result of the Land Grabbing Game

Process 5: Add the Land grabbing game rule control class to the game control form.

Once the structure and rules of the game are finalised, we can add the game rule control class to the game control form. The detailed sample code for adding the game rule control class object to the control form can be found in Appendix B. The method LoadTableSetting that we have implemented in the main control form (LandGrabGame) is used to load initial table settings to the toolkits when a game has been initialised. We define the game rule class as a variable as the following sample code shows.

```
private LandGrabRule _Landrule;
```

We update the LoadTableSetting method to initialise the game rule control class. The sample code is shown below.

```
void LoadTableSetting()
    {
     ......
    _Landrule = null; //your game rule contorl class object

    //Create a new game rule control class object
    _Landrule = new LandGrabRule(_RoboTableGMT);
```

```
//add an event to receive goal score update
_Landrule.RobotScoreUpdateEvent += new UpdateGoalScoreEventHandle(RobotScoreUpdate);

//set the game stop time in minutes
_Landrule.GameStopTime = 5;

//set the unit of goal score
_Landrule.GoalScore = 1;

//set the game rule details
_Landrule.NumberOfLandColums = 10;
_Landrule.NumberOfLandRows = 5;
TotoScore = _Landrule.NumberOfLandColums * _Landrule.NumberOfLandRows;

//set the minimum reconcile time in seconds
_Landrule.ReconcileTime = 4;

//create game map and goal object, you need call this method once
_Landrule.CreateGameMapAndGoalObjects();
.....
}
```

We now add robot update to the game rule control class object via the events provided by the GameManager class. The following sample code adds robot list update to the game rule control class object.

```
private void GameListUpdate(Dictionary<string, Robot> robotlist)
{
//post received game list to your game rule control class
_Landrule.UpdateRobotList(robotlist);

}
```

The following sample code adds robot update to the game rule control class object.

```
private void RobotUpdate(Robot robot)
{
//post received robot update (from all tables) to your game rule control class
_Landrule.PostLocalRobot(robot);

}
```

The following sample code adds goal message update to the game rule control class object.

```
private void UpdateGoalMessage(GoalMessage gmsg)
{
//post received goal messages to your game rule control class here
_Landrule.PostGoalMessage(gmsg);
}
```

This completes the implementation of the Land Grabbing game.

6.3.4 Case Study: Result and Discussion

We tested the Land Grabbing Game with four computers networked on the Lincoln University Network. We also used the game for the Multiplayer Scenario Evaluation outlined in section 6.2.3 and we discuss the result in section 6.4. The tests show that the toolkits can be used to develop a distributed multiplayer RoboTable game without a custom server required.

A limitation of the game is that synchronizing the final game result across multiple tables does not always work correctly. The problem is that when we use a random method to generate position messages, the result might not be the same across all tables. However, if we use simulated data, all results can be synchronized. We suggest some improvements to the current reconciling process and this will be discussed in Chapter 7: Future work. The following section discusses the result of multiplayer scenario evaluation.

6.4 Multiplayer Scenario Evaluation: Results and Discussion

This section discusses the performance of the distributed multiplayer RoboTable Game Toolkits by analysing the results of the evaluations. The evaluations estimated mainly the performance measures and the multiplayer scenario performance.

6.4.1 Baseline Performance Result

To determine the average robot speed, we constructed a robot with the following characteristics.

- Large wheels driven directly from motors

  The diameter of the largest wheel of the Robot that we used for evaluation was 8 cms.
- Fresh batteries

We programmed the robot to run each motor on full power. The robot was configured in this way because most applications of LEGO™ robots require responses to sensor input and therefore, a high linear speed is not usually required.

To obtain the average robot speed, we measured the size of the viewable RoboTable display, which is shown in Figure 6-10.

The size of the RoboTable display can give a scale between its lengths in cms vs. pixels. The image that is projected from a PC onto the RoboTable has a size of 1024 × 768 pixels. The size of the image displayed on the RoboTable screen is 97 cms × 70 cms. Hence one cm corresponds to 11 pixels approximately.

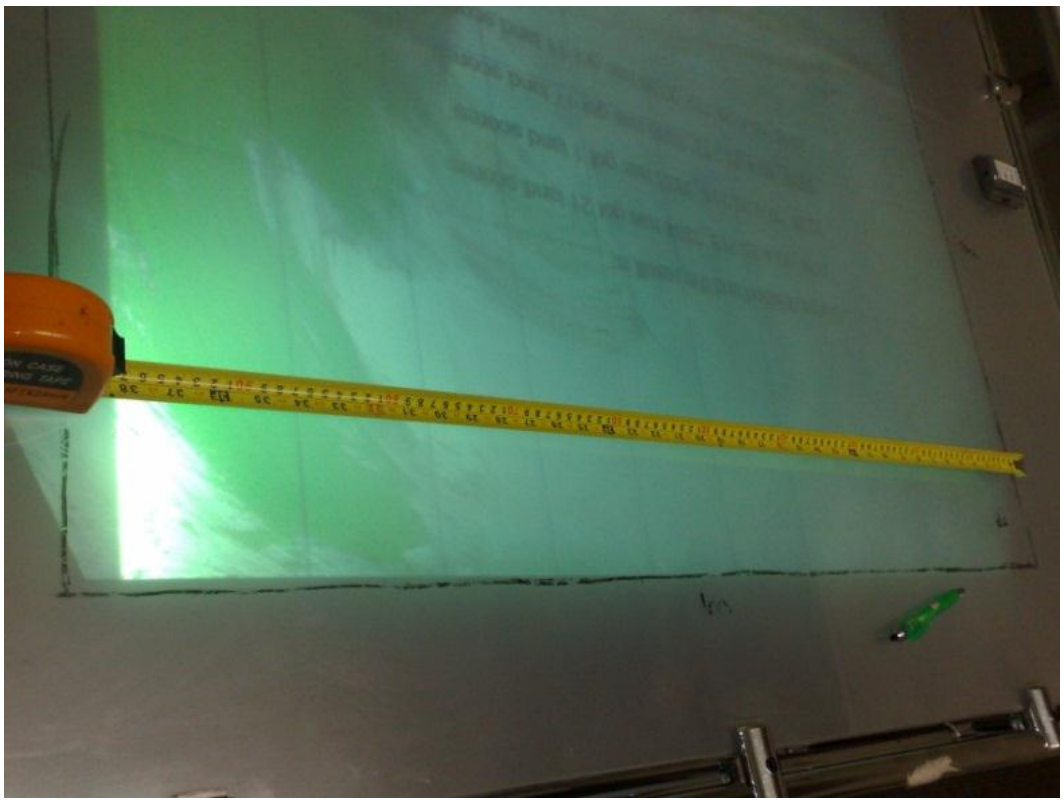


Figure 6-10 RoboTable Display

Average Robot Speed

The Robot speed can be calculated by measuring the distance that the testing Robot runs on the table and recording the time taken. In order to get an average speed, we recorded the time required for the robot to run for 30 cms, using a stopwatch. We repeated the testing 10 times manually and the mean was calculated. The results are shown in Table 6-3:

Table 6-3 The time a Robot needs to run for 30 cms

No. of Test	1	2	3	4	5	6	7	8	9	10
Seconds	1.64	1.43	1.82	1.82	1.70	1.67	1.85	1.78	1.75	1.87

From table 6-3, the average Robot speed is 17.3 centimetres per second, which is equivalent to 190 pixels per second.

Wiimote IR Tracking Update Rate

We ran a number of experiments to identify the Wiimote IR tracking update rate, and found that is extremely stable at approximately 98 updates per second, which is equivalent to a rate of 10.2 milliseconds per update. The experiments were repeatedly using a simple game on the physical RoboTable where the robot was tracked for about 10 minutes on 10 occasions. The number of Wiimote updates received and the duration in seconds was recorded for each experiment. The value

of the tracking rate for each experiment was calculated by dividing the number of Wiimote updates received for the duration.

Performance vs. Wiimote Update Rate

The average robot speed in pixels is about 190 pixels per second, which equates to 0.19 pixels per millisecond.

Because the Wiimote IR Update Rate is 98 updates per second (10.2 milliseconds per update), we conclude that the Robot can travel at most 1.94 pixels per Wiimote Update.

However, the tracking toolkit may generate too many position messages if the robot only moves 1.94 pixels per Wiimote Update. Position messages are used to communicate robot updates to remote tables and missing a few robot updates is acceptable as long as a game can still run smoothly. In order to find out how robot update performance can be affected by the transmission rate, we added a control to slow down the transmission rate by skipping some position messages. The control is a duration in seconds that the Network Toolkit will not send position updates to all tables. To do this, the Network Toolkit records the time the last position update was sent and checks the time of receiving an incoming position update. If the time of the latest position update received is greater than the sum of the duration and the last position update sent-time, the Network Toolkit sends the latest position update out to all tables and records the latest position update received time as the last position update sent time, otherwise it does nothing.

We have run a number of tests based on different time settings. Table 6-4 presents the results of the tests.

Table 6-4 Result of base line performance evaluation with time lag

Time Lag (ms)	**20**	**30**	**40**	**50**	**60**	**70**	**80**
Number of Position Messages Skipped	1	2	3	4	5	6	7
Position Updates/Second	50	33.3	25	20	16.7	14.3	12.5
Robot Update Performance	Smoothly	Smoothly	Smoothly	Less Smoothly	Less Smoothly	Marginally Acceptable	Marginally Acceptable

The robot update performance results in Table 6-4 were determined from observations of robot movement performance from a simple game. We altered the time between updates and observed the outcome on 20 occasions. The “Acceptable” term here is decided by the judgement of how smoothly the robot movement can be reproduced on remote tables.

From the results in Table 6-4, we determine that if the time between sending position updates is more than 60 milliseconds, the performance becomes unacceptable. A time between 20 to 60 milliseconds appears to be acceptable because the robot movement can be reproduced smoothly.

The best time between updates is 40 milliseconds because the update rate is 25 updates per second, which matches the Real-Time Performance requirement.

In summary, the maximum distance that the robot can travel per Wiimote update is about 0.17 cms, and the maximum number of position messages we can skip is 5 to maintain smooth update rates. The maximum number of position messages we can skip to maintain Real-Time performance is 3.

6.4.2 Multiplayer Scenario Evaluation

In order to test the multiplayer performance of the toolkits, we ran a number of experiments using the game we developed and discuss in the case study (Section 6.3). The experiments used 4 PCs running simultaneously. One PC was connected to the RoboTable; this is the master table. The other PCs were virtual tables. The virtual tables simulated a physical RoboTable by generating position messages with a time interval of 10 milliseconds per update, similar to the Wiimote IR tracking rate.

We have used the Land grabbing game, which we describe in section 6.3, to test all multiplayer scenarios, and each scenario takes about 5 minutes.

The scenarios that we used for the evaluations were as follows:

- One master table with one virtual table
- One master table with two virtual tables
- One master table with three virtual tables

The results for the experiment with maximum speed are shown in Figure 6-11.

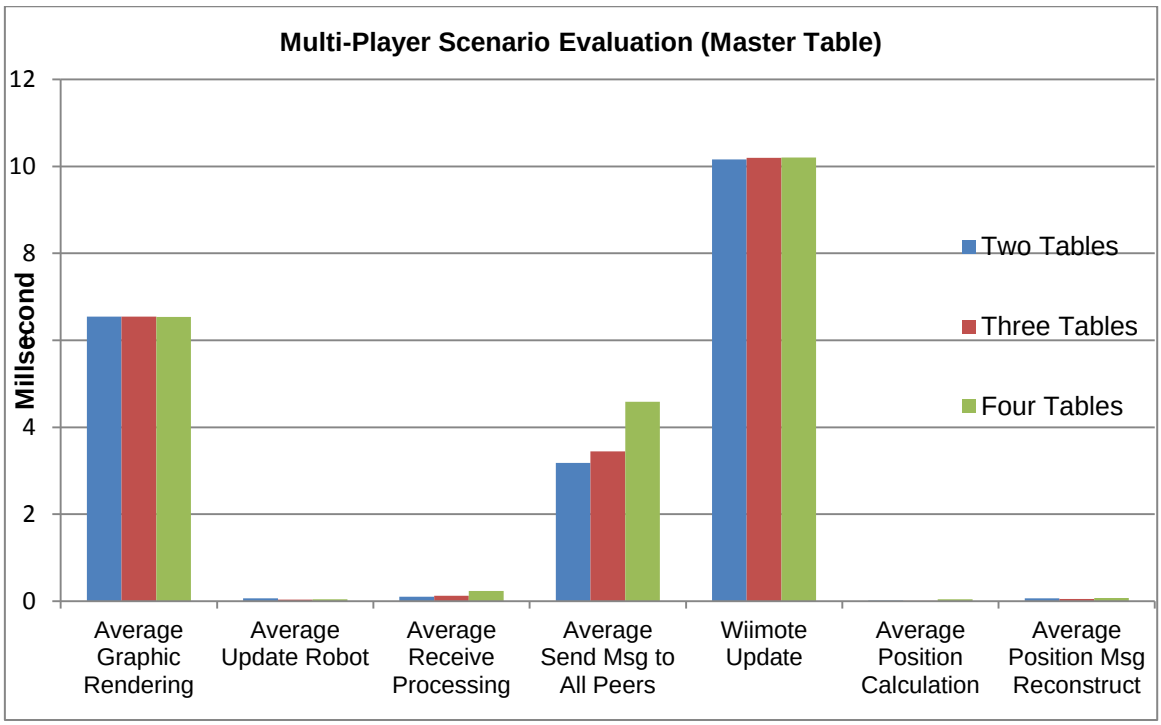


Figure 6-11 Performances with Two to Four RoboTables

The results of the experiment are shown in Figure 6-11 and they show that:

1. The Wiimote update rate remains stable at about 10.2 milliseconds.

2. The Send Message to All Peers is the only performance measure that is significantly affected by increasing the number of tables. The reason is that the Network toolkit uses a loop to send messages to all peers one by one. Therefore, the more tables that are connected, the more time is required to send messages.

The time to send robot update, which is the summation of “Position Calculation”, “Wiimote Update” and “Send Msg to All Peers”, for 2 to 4 tables, is shown in Figure 6-12. The time to display robot updates, which is the summation of “Graphic Rendering”, “Update Robot”, “Receive Processing” and “Position Msg Reconstruct”, for 2 to 4 tables, is shown in Figure 6-13. From the two figures, we can see that the total processing time (approximately 22 milliseconds per update) for sending and displaying robot updates for 4 tables is lower than the video frame rate, which is 24 updates per second (41.7 milliseconds/update). Therefore, the toolkits can process robot updates in real-time for multiple tables and more tables can join and play concurrently because adding an extra table adds only approximately 1 millisecond. Hence the toolkits match the requirement of Real-Time performance.

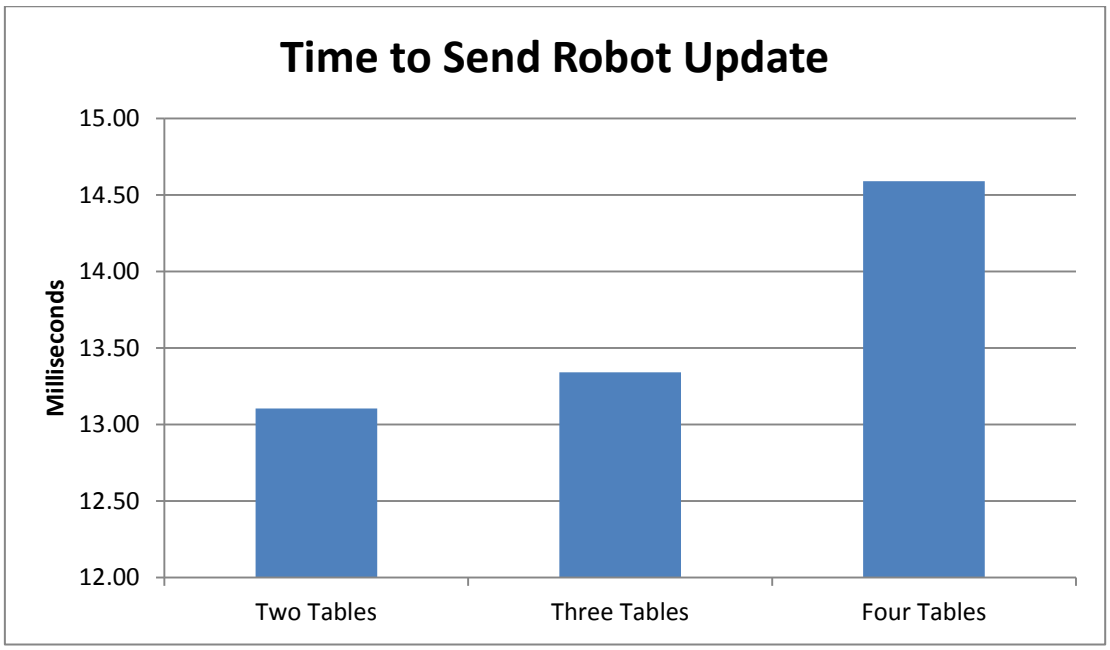


Figure 6-12 Time to Send Robot Update

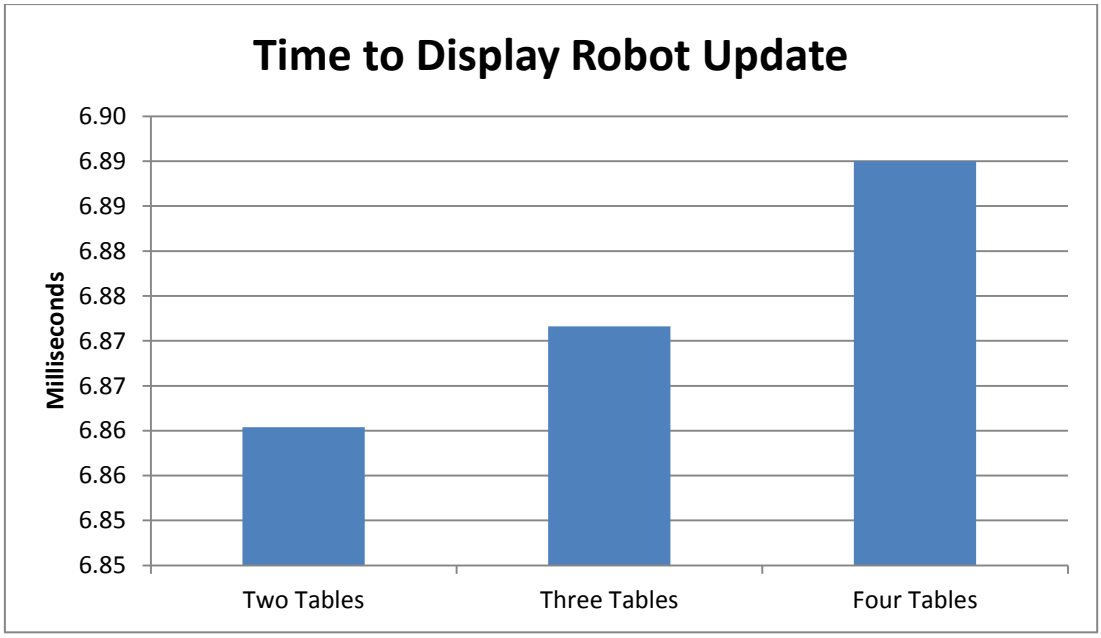


Figure 6-13 Time to Display Robot Update

Because the evaluation used 2 to 4 tables, the toolkits match all requirements outlined in section 6.1.

6.4.3 Communication Evaluation

The communication toolkit uses SkypeKit. Currently the toolkit allows a user to communicate with others using his or her Skype™ account. Video chat was not developed at this stage because it was not a requirement for this project. The toolkit can talk with other Skype™ users through Text or Audio communication. We evaluated the communication toolkit by setting up two Skype users on two different PCs. We conducted two tests as follows:

- Talking using the communication toolkit without playing game concurrently
- Talking using the communication toolkit and playing game concurrently

We discovered that if we run the games while we are talking via the communication toolkit, the performance of the game is still acceptable, but voice transmission may have some interruptions compared to talking using the communication toolkit without playing a game concurrently.

6.5 Summary

The game, which we used for the multiplayer scenario evaluation, is created from the case study which means that the processes followed in the case study successfully created a real-time multiplayer RoboTable game.

In section 6.1, we outlined the requirements for the toolkits and we consider them below to show how the toolkits have met the requirements.

- Using the toolkits should be straightforward

  From the case study, we see that the BaseGameRuleControl class is designed to help game developers to program their own game rules easily. The game rule control class in this case study inherits the BaseGameRuleControl class. From the discussion in section 6.3, we see that the base classes already implement the majority of functions for game developers. There were only two main parts that the game developers needed to develop namely UpdateGraphics() and CreateGameMapAndGoalObjects(). The main methods we use to interact with the RoboTable graphics are AddGraphicObject, ChangeGraphicObjectLocation, HideGraphicObject, ShowGraphicObject and ChangeGraphicObjectColour.

  The connection form is a simple form and very little code is required to make it work. To integrate it with the communication toolkit, we only need to design the game controls and pass them to the communication toolkit. Because the skeleton project has implemented them, the case study uses the two forms directly. As a result of developing the Land Grabbing game, game developers have a skeleton project that helps developers more than just having the toolkits. Hence the development of a new distributed multiplayer RoboTable game should be relatively straightforward using the toolkits.

- Real-Time performance

  The results of the multiplayer scenario evaluation have shown that the toolkits perform well in real-time with a rate of 45.5 updates per second (approximately 22 milliseconds per

update), which is about twice the Real-Time performance requirement of 24 updates per second. In addition, we can skip up to 5 messages while still maintain good performance.

- Distributed multiplayer performance

  The results have also shown that the toolkits perform well in real-time within a distributed computer environment of up to 4 tables. Hence, the toolkits have met this requirement.

- Tracking subsystem performance

  Finally, the multiplayer scenario has shown that the tracking subsystem performance is very stable at 10.2 milliseconds per update and is not affected by the robot speed. Hence the toolkits have met this requirement.

Chapter 7
Summary and Future Work

7.1 Summary

The RoboTable is a mixed reality tabletop learning environment and provides a distributed learning platform that can allow students at remotely located tables to interact and compete on robotic projects. Mason (2005) developed a game for the RoboTable to demonstrate the potential of the platform. Despite this there are no general-purpose tools to support RoboTable game development or distributed game play. Hence a gap exists to develop a robust solution that will allow distributed multiplayer games to be created and played using RoboTable.

To address these issues, we have investigated the requirements of the RoboTable system and have designed, developed and implemented toolkits to help game developers develop multiplayer RoboTable games.

The set of toolkits comprise a Network Toolkit, a Robot Tracking Toolkit, a Game Management Toolkit and a Communication Toolkit. In addition, we have developed a skeleton project to help game developers start game development.

To evaluate the toolkits, we have used a number of assessment approaches. The first approach was a case study of the game development process using the toolkits. The second approach was to establish baseline performance benchmarks for the system. The third approach was to carry out experiments to evaluate their real-world performance and the scalability of the toolkits using the game created in the case study. The results from the experiments have revealed that the toolkits perform well within the Lincoln University environment and that the development of a new RoboTable game is relatively straightforward.

The toolkits satisfy the goals of the project. We conclude the design goal of the toolkits as follows:

- Flexible and extensible

  The toolkits deal with abstract classes and each toolkit is independent of each other, except for the Game Management Toolkit. Hence, the toolkits are flexible and extensible. If one toolkit is out of date, we can upgrade the toolkit using the same abstract class, and the system still works.

- Using the solution should be straightforward

  The case study has shown that the development of a new multiplayer RoboTable game is relatively straightforward. The toolkits not only provide a general interface to game

developers but also provide a skeleton game project to help game developers to develop their game easily.

- Good Performance

  The multiplayer scenario evaluation has shown that the toolkits have real-time performance in distributed environments. The toolkits are able to support 4 tables concurrently. Currently the robot update rate is 45.5 updates per second for 4 tables which is about twice the current video frame rate of 24 updates per second.

- Good Functionality

  The toolkits manage network communication, robot tracking and game states. The evaluation has shown that game developers do not need to worry about how to implement network communication and robot tracking facilities. Hence the toolkits provide good functionality to game developers. Game developers don't need to maintain game states as the toolkits can maintain all game states at the local table consistently.

However, in developing the toolkits we have identified some limitations both in their implementation and evaluation. Possible future improvements are discussed in section 7.2.

7.2 Future Work

7.2.1 Base Game Rule Control Class

We have demonstrated how to use the base game rule control class to create game rules for the Land Grabbing game. One problem with the current base game rule control class is that it may have some problems in matching the final results across all tables. We suggest reviewing the routine for reconciliation and it may require all tables to confirm the final results. One way to do this is to send a final message to ask all tables to confirm the final status of a goal object and all tables will need to send a message to confirm or protest. If a claiming goal object is confirmed by all tables, then the goal object definitely belongs to the claimant.

The base game rule control class contains methods for goal message sending and receiving, goal claiming, goal reconciling, goal results output and game finishing controls. However, the base game rule control class might not be suitable for other games that do not need goal reconciliation. The reason for this is that the base game rule control class is derived from the development of the sample game i.e. Land Grabbing Game. Hence the base game rule control class may not be general enough for all game types and may need to be redesigned in future. A fully generalisable base game rule control class implementation is outside the scope of this project.

7.2.2 Network Communication

We have implemented a robust Network communication toolkit, however it has some limitations. One limitation is that we do not encode any message contents, and this may cause some problems. For example, if a user enters a string “xxx</Body>” and sends the string as a text message to all tables, the string will break the system because we have used “</Body>” as the key value for the Network Stream. To solve these problems, we suggest adding a string coder to encode the message contents into numbers and decode the numbers into a string when the messages have been sent to the final receiver. This is an important issue as game developers do not want their game players to break their game by simply entering a special string like “xxx</Body>”.

Another limitation is that there is no security to check if a packet is from another RoboTable or if it is a fraud. Also the toolkit does not check if it has received all messages from the other tables. Hence, if game developers want to use UDP to send messages, they must implement their own mechanisms to check the integrity of received messages. TCP guarantees delivery network packets in the correct order. However, the toolkits are aimed to provide flexible facilities to allow different network protocols to be implemented rather than using only one network protocol.

As we have discussed in Chapter 5, the network messages are sent or received using the class NetStream. The class NetStream is a key class for converting messages into network stream format and vice versa. We can add security validation in the class NetStream to defeat malware attacks.

Regarding message integrity, we can add buffers to store operational messages. These messages will be numbered in succession. If an operational message is received and its number does not continue on from the previous operational message received, the system should send a ReSend request to the sender of the message for the missing message. The sender should resend the message from its sending buffers when the ReSend request is received.

Checking the integrity of operational messages is suggested because tactical messages are less significant than operational messages. The tactical message is the position message which is used to display the robot update. Missing a few position messages does not impact the performance of the toolkits, as discussed in section 6.4.1.

7.2.3 Animation

The design and implementation of RoboTable graphics rendering provides opportunities for animation and 3-Dimensional rendering.

We suggest adding animations to make the game more entertaining, especially if the game is designed for school children. “Kids enjoy animation” (ABCYA.com, 2012). A possible animation could

start when a robot reaches a goal object, when the goal object would be magnified quickly before fading away.

To animate output, we need only to add a timer, or using the ThreadPool to perform continual transformations of a graphics drawing object.

7.2.4 Communication

The communication toolkit assumes that game players are members of Skype™ already. The contacts for a Skype™ user must be added through the Skype™ application or website. In future, we suggest adding a tool for managing the Skype™ user profile in the communication toolkit. A Skype™ user profile management tool would allow a RoboTable user to register as a new Skype™ user and would enable the RoboTable user to manage their Skype™ account directly through the communication toolkit. The profile management system would also allow a user to add new contacts through the communication toolkit.

The current communication toolkit provides limited features for text and audio communications. However, we suggest enabling Video communication in the future so that the RoboTable game can use the communication toolkit for video chat. Audio and video communication may have some impact on the performance of the toolkits. This would have to be evaluated.

Apart from the SkypeKit, the Network Toolkit also provides flexible facilities for constructing text and audio communication features. Currently the toolkits allow text messages to be sent directly to all peers or a single peer. In future, we could implement voice messages to enable an audio conference to be held using the toolkit directly and using the same structure as other messages, rather than using third party components such as Skype™ facilities. An advantage of this would be that game players would not need to register as the Skype user before they can use the communication toolkit.

7.2.5 Independent Qualitative Evaluation

The evaluations that we have conducted focus on quantitative performance measures for the toolkits and a self-assessment of the ease of use of the toolkits for developers. Therefore, there is no independent qualitative evaluation to ensure acceptable performance and game play. We suggest performing an evaluation of the toolkits in an educational environment using the Land Grabbing Game in order to assess its true performance in a real-world independent environment. We also suggest having different game developers do user trials on developing a distributed multiplayer RoboTable game using the toolkits so that we can investigate functionality issues about the toolkits and understand how easy the toolkits are to use to develop a game.

7.2.6 Extensibility

The current Robot Tracking Toolkit cannot track multiple robots on one table because the Wiimote technology, adapted by the toolkit, has a limit to tracking one robot's position only per table. An efficient robust tracking system that does not have this limitation would allow the tracking of multiple robots' positions on one table. This could be implemented by an upgrade without the need to rewrite the whole system.

Currently a robot image moves to each new location via a single discrete step. To date this has been found to offer movement that is smooth enough, however an interpolation scheme (eg. splines) may offer smoother transitions.

In addition it may be possible to increase the time between updates (see Table 6-4) using a predictor-corrector approach. We can extend the Robot class, which is inherited from the BaseRobot class, to use a predictor-corrector method to automatically calculate the next position value if a Robot object has not received a new position update in a short time frame.

7.3 Final Remarks

This study investigates how to support the implementation of distributed multiplayer games for a RoboTable environment so that game developers do not have to implement multiplayer functionality from scratch for each new game. The solution of this study is a multi-toolkit approach. The approach comprises a Network Toolkit, a Robot Tracking Toolkit, a Game Management Toolkit and a Communication Toolkit. The Network Toolkit and Robot Tracking Toolkit are independent of other toolkits. In order to evaluate the approach, we conducted a case study in which we developed a multiplayer RoboTable game. We evaluated the toolkits using the game within multiplayer scenarios. The results show that the toolkits have real-time performance and can support up to 4 tables concurrently. Developing a new game using the toolkits is relatively straightforward. As a result of developing the game, game developers are able to use a skeleton project that helps developers more than just having the toolkits.

This study has made a contribution to the research in multiplayer RoboTable game development. Before this study, there were no general-purpose tools to support RoboTable game development or distributed game play. Now a set of toolkits has been designed and implemented to support multiplayer RoboTable game development.

Appendix A

Interview Question List

A.1 Goal of this Interview

1. To understand client's initial requirements

2. Find out the client's usability, user group, how they are going to use this toolkit

3. Find out user's respective or objective, client's objective

4. Find out Technique requirements of this project

5. Find out the limitation of this project (scope of this project)

A.2 Question sections:

Usability

1. Who are going to play with this RoboTable game?

2. What is the main objective of playing with this RoboTable game?

3. Explain how a person uses the table to play a game at the moment.

4. Explain how a person interacts with a RoboTable.

5. Explain how game developer develops a RoboTable game.

6. How would you like that to change (if at all)?)

7. Are user allowed to chat with their opponent while playing a match?

Interface

What is user's expectation of interface for RoboTable game developing?

Data Structure

1. What sort of the main data generated by current RoboTable game

2. Explain what other data should be included into a future RoboTable game?

Techniques

1. What programming language is going to use for developing the toolkit

2. Explain how to security network communication of RoboTable game?

3. From user's point of view, please explain the network requirements for developing RoboTable game.

4. From user's point of view, explain how to manage game states?

5. If a player dropped during the normal playing session, how can the system detect and inform all parties?

6. How game developer deal with network protocol?

7. Are multiplatform required? Explain

8. Explain how you see someone developing a new multiplayer game for the RoboTable?

Limitation of this Project

1. What kind of limits of developing a RoboTable game?

2. What is the maximum number of players?

Testing and Evaluating

1. Explain the requirement of designing a simple game for evaluation

2. Is tester expected to code some sample program to test with the toolkit? If it is yes please explain how they are going to test.

Appendix B

Game Developer Guide

B.1 Overview

This document is a self-contained developer guide to help you to develop a distributed multiplayer RoboTable game using the toolkits. In addition, Appendix C provides a manual for the game management toolkit, BaseGameRuleControl class and BaseGoalObject class, and Appendix D provides debugging support for game development.

The distributed multiplayer RoboTable game toolkits incorporate a Network Toolkit, a Robot Tracking Toolkit, a Game Management Toolkit and a Communication Toolkit. The Game Management Toolkit integrates the Network Toolkit, the Robot Tracking Toolkit and provides generic interfaces for game developers to conveniently build distributed multiplayer RoboTable games. The Communication Toolkit is an independent toolkit, which provides text and audio communication facilities via the SkypeKit interface. Game developers can integrate the Communication Toolkit into their game application. The interconnection between toolkits is shown in Figure B.1.

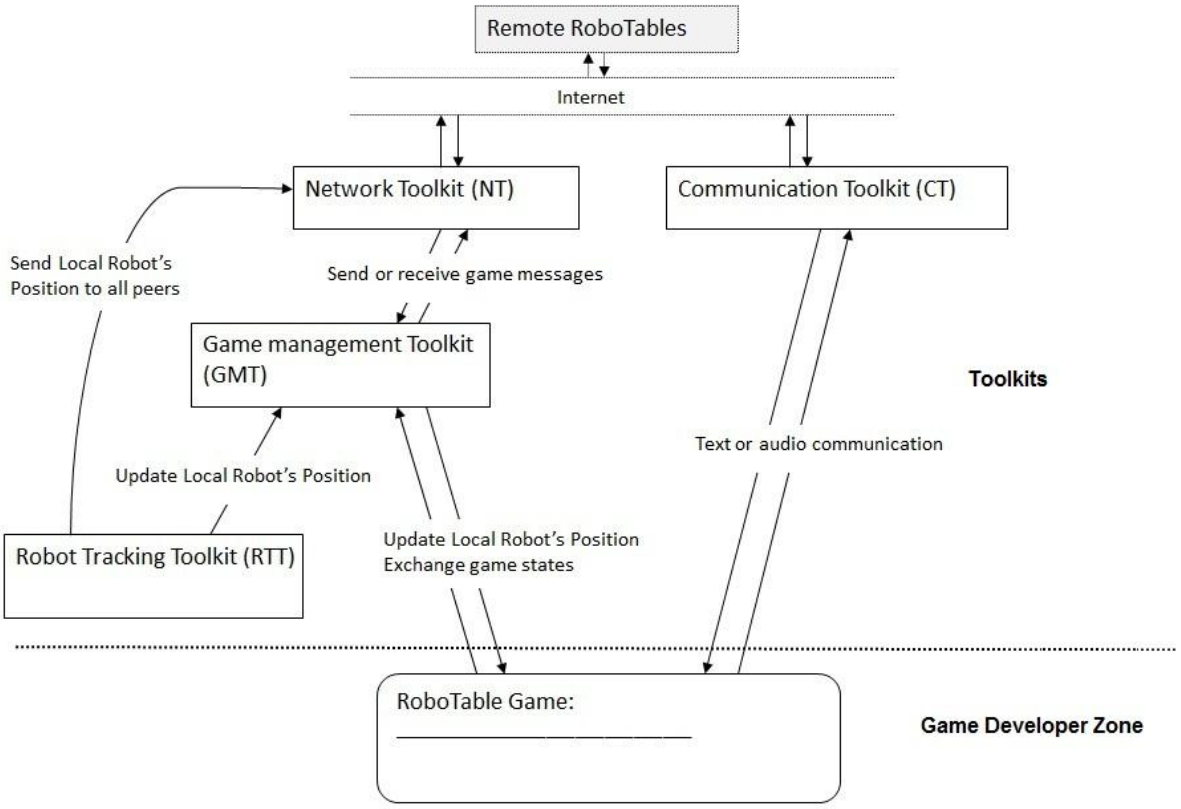


Figure B.1 Interconnection of the toolkits

This guide will show you how to develop a game using a skeleton project provided with the toolkits.

This guide is for a .NET developer with some knowledge of the RoboTable. However, developing a distributed multiplayer game can be straightforward and here is an overview of the process is presented below:

- Design Game
- Skeleton Project
- Create Game Map
- Develop Game Rules
- Add Game Rules to Game Loop
- Testing and Debugging

To help illustrate the process we will use the example of a Land Grabbing game. The Land Grabbing game is a multiplayer RoboTable game we developed for this case study. It contains a number of rectangular land objects as shown in Figure B.2. The robots on different tables run concurrently and try to grab as many land objects as possible in order to win the game. Each robot has its own colour and when a robot grabs a land object, the colour of the land object will be changed to the colour of the robot. When the game play time exceeds 5 minutes, or there are no more land objects available for capturing, the game will finish and the final result reporting the number of objects grabbed by each robot, will be displayed.

The game map, which is shown in Figure B.2, is a uniform grid of rectangles of 50 (5 Rows and 10 Columns). The background colour of each square at the start is white, and the frame colour of each square is blue. The rectangle is the land object that robots will try to grab.

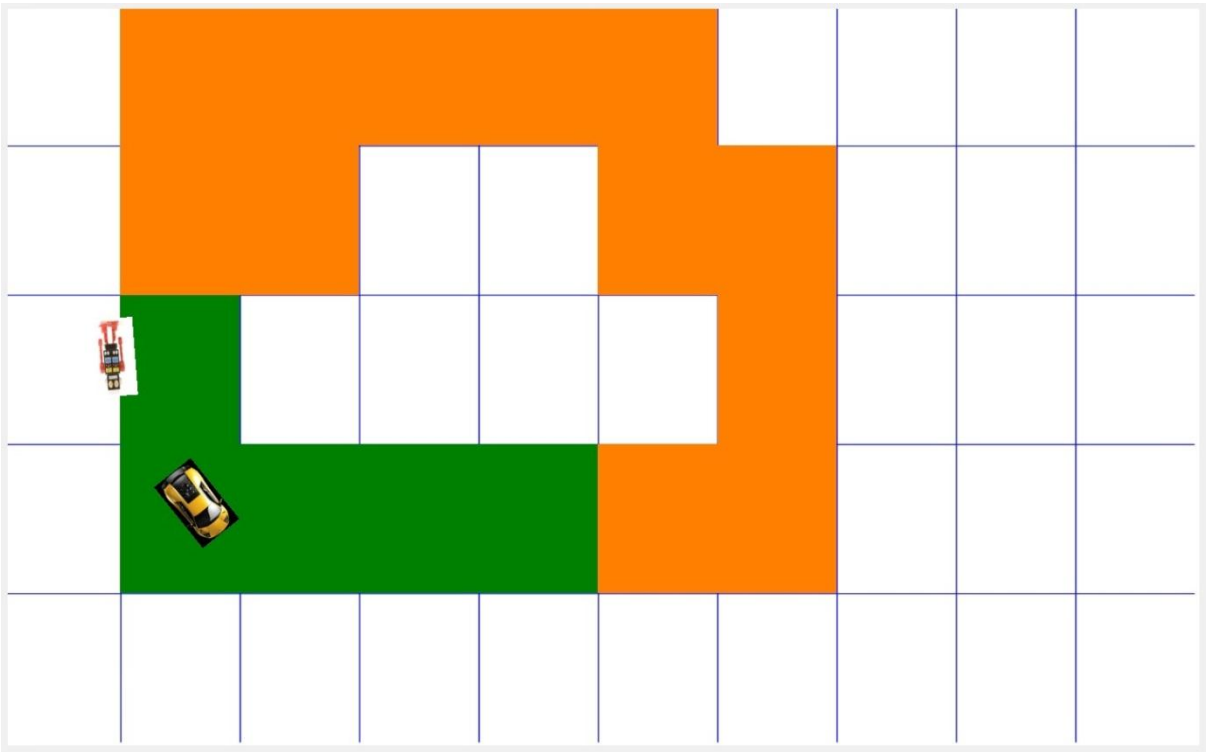


Figure B.2 The Land Grabbing Game Map

B.2 Design Game

To begin creating a game, the design of the game needs to be determined. This involves determining the objective and rules of the game and what the game map looks like. In our Land Grabbing game, the objective is to get as many lands as possible by being the first to enter a land area and therefore acquiring it.

From the objective we can develop game rules for our Land Grabbing game as follows:

- If a robot enters a Land then it acquires it if it is the first robot to enter the land.
- The winner of the game is the robot with the most lands when either.
    a. All Lands have been acquired
    b. 5 minutes of game play have expired
    c. Game play has been aborted.

In determining the objectives and rules it is also helpful to think about the game map, the playing board for the game. In our Land grab game we use a grid layout but a map can be as simple or complex as required. Remember however it will have to be created as an image with the type of JPEG. The size of the Game map will be 1024 * 768 (RoboTable Screen size). The size is determined by the resolution of the Wiimote controller.

B.3 Skeleton Project

The skeleton project provides a complete and implemented game structure. What you are going to do is:

1. Load the skeleton project to your current project
2. Design the game map
3. Develop the game rules
4. Add the game rules to the skeleton project
5. Test the game

The files in the skeleton project are shown in Figure B.3, you need to load the project file "SkeletonProject.csproj" into your project solution. Your Skeleton project should contain all files and folders that are shown in Figure B.3.

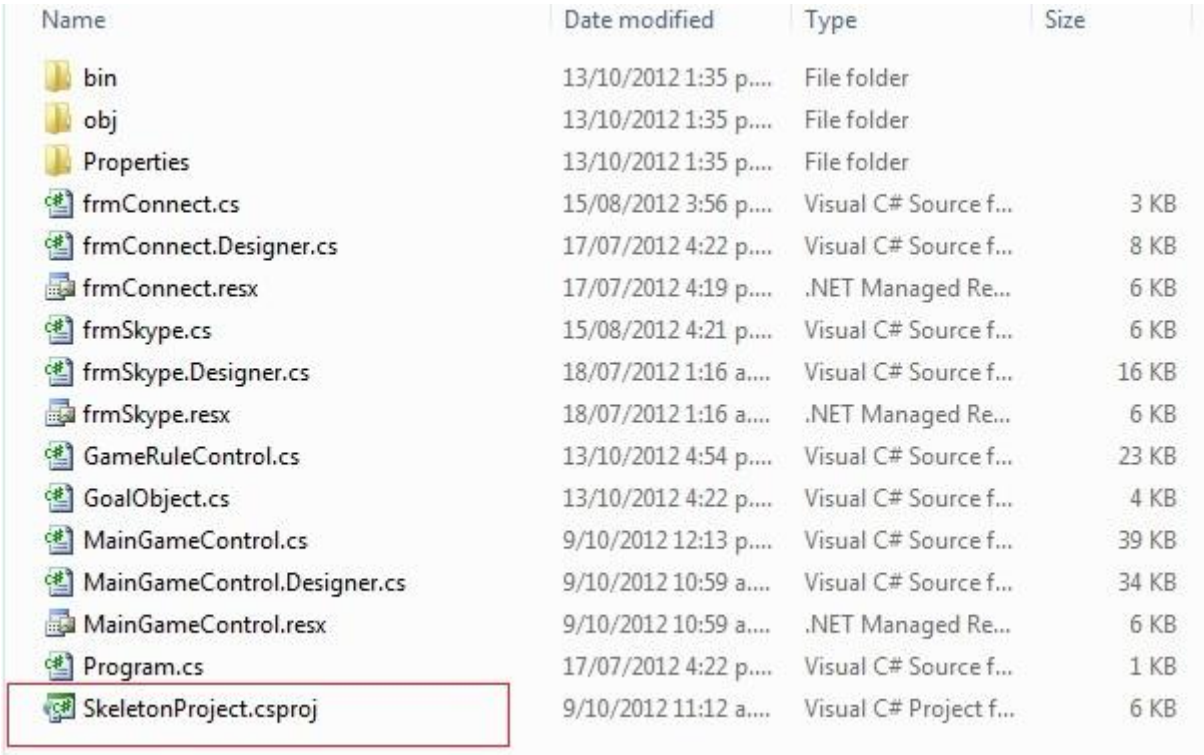


Figure B.3 Skeleton Project Files

The skeleton project includes three form classes and two classes for game rules (Figure B.4). The three form classes are: MainGameControl class, frmSkype class and frmConnect class. The two classes for game rules are GameRuleControl class and GoalObject class.

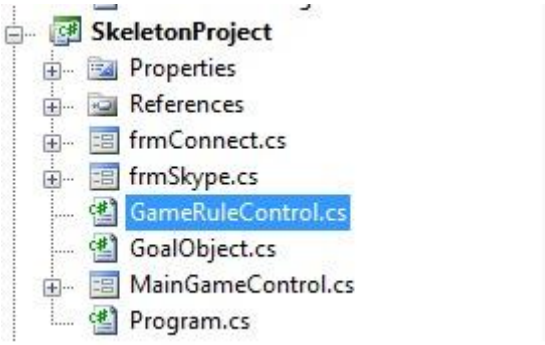


Figure B.4 Skeleton Project

Please ensure the skeleton project contains all the necessary DLLs (Microsoft.DirectX, Microsoft.Direct3D, Microsoft.Direct3DX, System.Drawing, MRGT.Base, MRGT.CommunicationToolkit and MRGT.GameManagementToolkit). You should add them into your project's References folder as shown in Figure B.5.

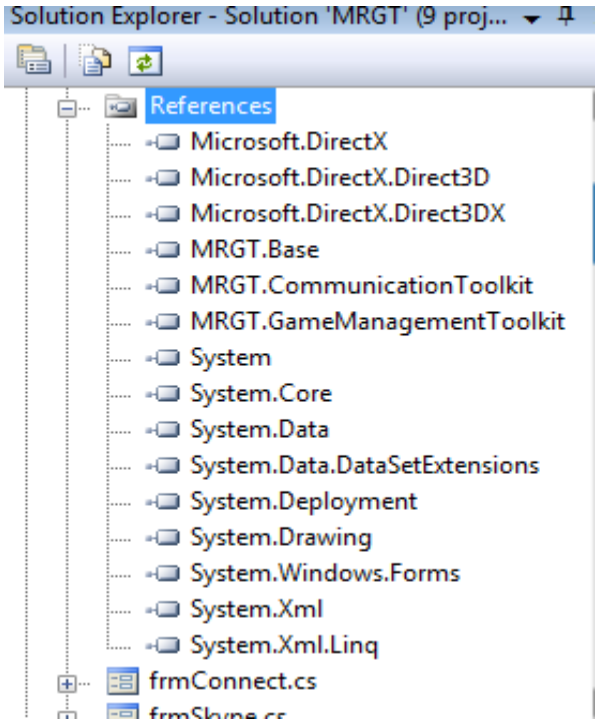
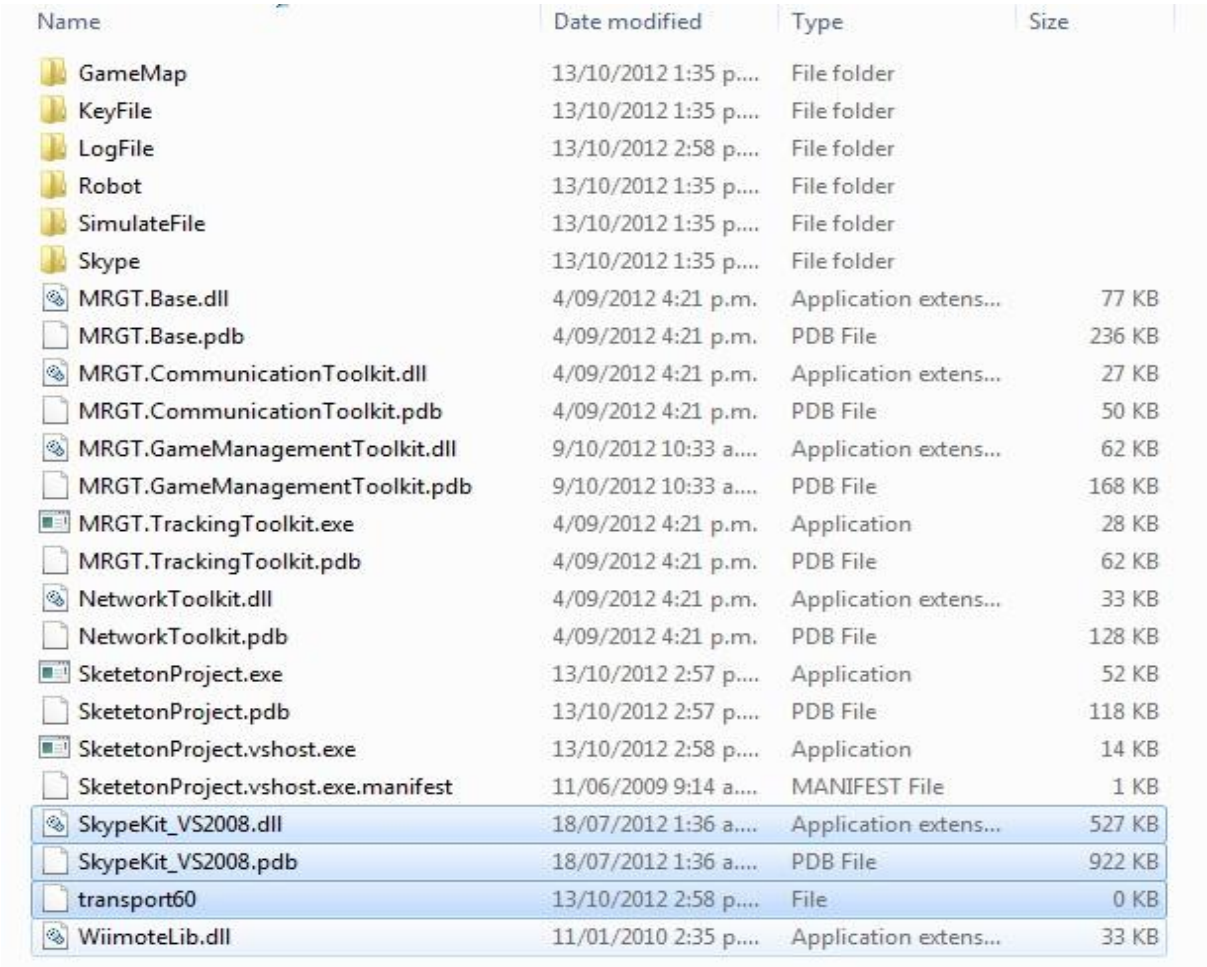


Figure B.5 The Project References

In the main control form, you need to add the following namespaces to the head of the form class.

```
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using MRGT.Base;
using MRGT.GameManagementToolkit;
using System.Threading;
```

In your project, please make sure the following folders and DLLs are in your bin folder (either debug or release) as shown in Figure B.6. The GameMap folder contains game map images (your custom created game map should be placed there). The KeyFile contains all key data files for Skype (windows-x86-skypekit, Skypekit developer key), Calibration.dat and NTPServer.txt. Calibration.dat is a file that stores all calibration data when you run a calibration of the table with WiiMote. NTPServer.txt contains a list of NTP servers for the toolkits to use. Windows-x86-skypekit and SkypeKit developer keys are required when you use Skype toolkit. The current SkypeKit developer Key is specific to this research project and you should apply for your own SkypeKit account and get your own keys and make sure your key names are the same as “robotable1.0.pem” and “robotable1.0.pfx”. The extra DLLs you need are SkypeKit_VS2008.dll and WiimoteLib.dll. These extra DLLs may automatically be created by the other toolkits. The Skeleton project contains all of these DLLs, folders and files. The items shown in Figure B.6 are under Skeleton Project/Bin/Debug. You need to copy them to your release folder when you are going to release them. The toolkits are locating these folders and files based on its current project‘s working directory (/bin) and hence there is no problem for you to move them to the Release folder.



Name	Date modified	Type	Size
GameMap	13/10/2012 1:35 p....	File folder	
KeyFile	13/10/2012 1:35 p....	File folder	
LogFile	13/10/2012 2:58 p....	File folder	
Robot	13/10/2012 1:35 p....	File folder	
SimulateFile	13/10/2012 1:35 p....	File folder	
Skype	13/10/2012 1:35 p....	File folder	
MRGT.Base.dll	4/09/2012 4:21 p.m.	Application extens...	77 KB
MRGT.Base.pdb	4/09/2012 4:21 p.m.	PDB File	236 KB
MRGT.CommunicationToolkit.dll	4/09/2012 4:21 p.m.	Application extens...	27 KB
MRGT.CommunicationToolkit.pdb	4/09/2012 4:21 p.m.	PDB File	50 KB
MRGT.GameManagementToolkit.dll	9/10/2012 10:33 a....	Application extens...	62 KB
MRGT.GameManagementToolkit.pdb	9/10/2012 10:33 a....	PDB File	168 KB
MRGT.TrackingToolkit.exe	4/09/2012 4:21 p.m.	Application	28 KB
MRGT.TrackingToolkit.pdb	4/09/2012 4:21 p.m.	PDB File	62 KB
NetworkToolkit.dll	4/09/2012 4:21 p.m.	Application extens...	33 KB
NetworkToolkit.pdb	4/09/2012 4:21 p.m.	PDB File	128 KB
SketetonProject.exe	13/10/2012 2:57 p....	Application	52 KB
SketetonProject.pdb	13/10/2012 2:57 p....	PDB File	118 KB
SketetonProject.vshost.exe	13/10/2012 2:58 p....	Application	14 KB
SketetonProject.vshost.exe.manifest	11/06/2009 9:14 a....	MANIFEST File	1 KB
SkypeKit_VS2008.dll	18/07/2012 1:36 a....	Application extens...	527 KB
SkypeKit_VS2008.pdb	18/07/2012 1:36 a....	PDB File	922 KB
transport60	13/10/2012 2:58 p....	File	0 KB
WiimoteLib.dll	11/01/2010 2:35 p....	Application extens...	33 KB

Figure B.6 Skeleton Project Bin folder

We provide a brief overview of the forms (MainControlForm, frmConnect and frmSkype), and game developers do not need to make significant changes to these.

The Main Control Form

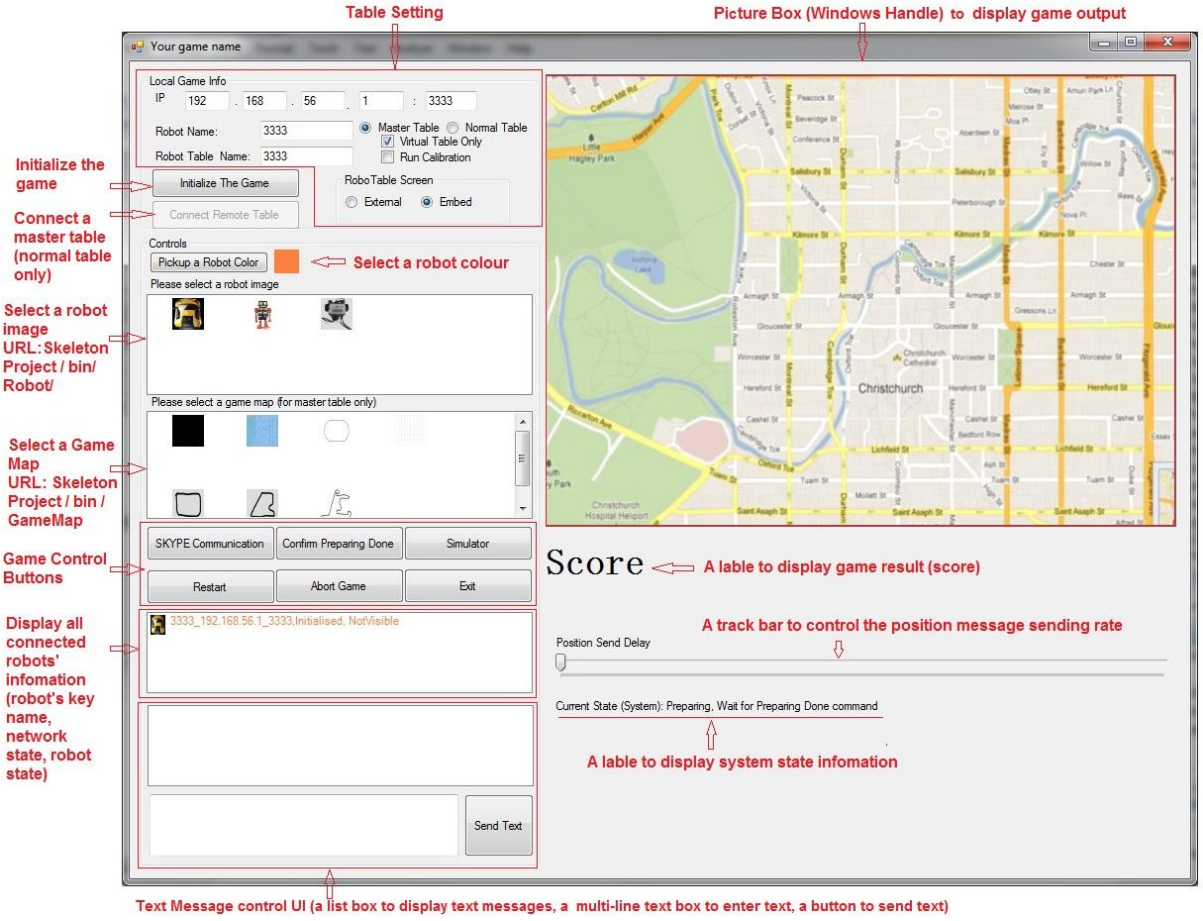


Figure B.7 The main control form for the Land Grabbing Game

The MainGameControl form is shown in Figure B.7. It has five small text boxes for the local IP address and port number, and two text boxes for the Robot name and RoboTable name. One button is used to pick a colour for the local robot, and two radio buttons are used to determine if the game is a master table. If it is the master table, no remote connection is required. The checkbox “virtual table only” indicates if the RoboTable is a virtual or physical table. The check box “Run Calibration” is used for the physical table only when a new calibration is required. Another two radio buttons “External” and “Embed” indicate if you want the game output to be displayed on an external monitor (physical RoboTable) or on the current Picture box. The MainGameControl form provides some extra implemented controls such as the Robot Colour picker and game map selector, for game developers to work with and game developers can remove any of these controls that are not related to their games.

The list box under the radio button is a list of Robot Images available for game users to choose. It also contains a number of button controls. Game players can select a colour representing the local

robot. The robot selector works only when all tables contain the same robot images with the same names (jpeg / bmp) in the folder: your project/bin/Debug/Robot.

The list box under the robot image selector is the Game map selector, which is only available for the master table. When a game map is selected, all tables will be updated with the selected game map. The game map selector works only when all tables contain the same game map image with same name (jpeg / bmp) in the folder: your project/bin/Debug/GameMap.

The Connection Form

The connection form, which does not need to be changed by game developers, requests users to enter the IP address and port number of the remote master table. Figure B.8 shows this form. This form is shown automatically to users when the table has been set as normal table and has not been connected with a remote master table.

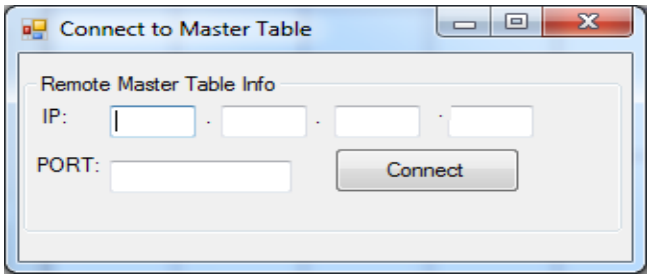


Figure B.8 The Connect Form for connecting the remote master table

The Communication Form

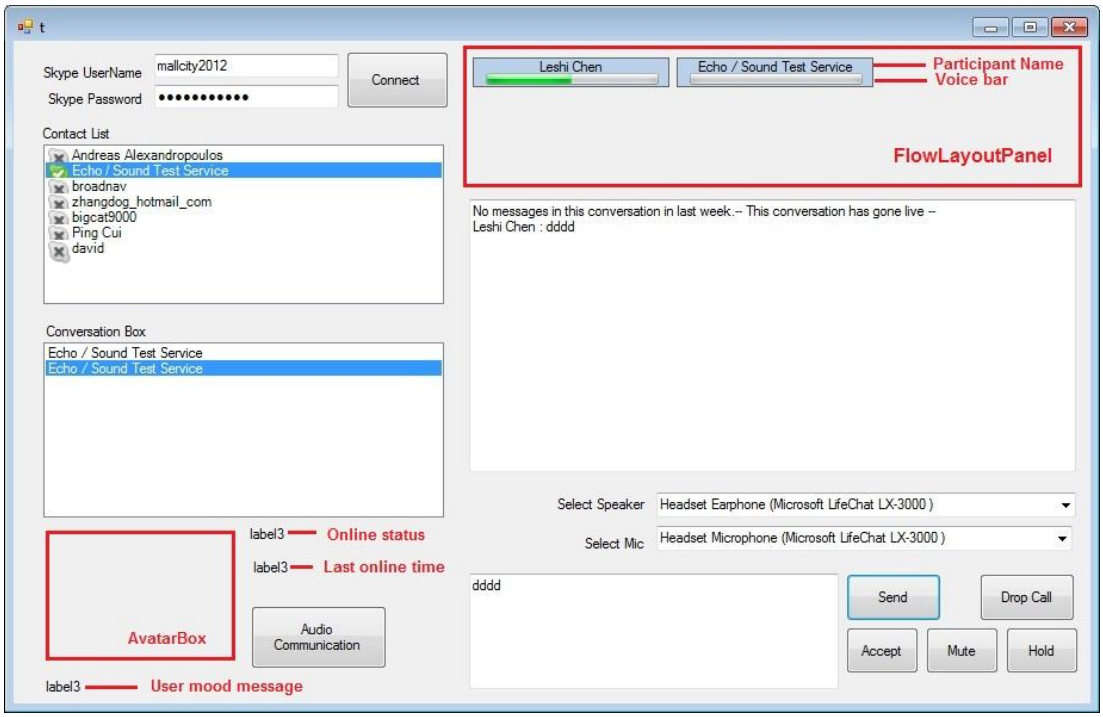


Figure B.9 The communication form

The communication form, which does not need to be changed by game developers, is shown in Figure B.9. This form is used with the communication toolkit. It contains two single line text boxes for the username and password, and two list boxes for contacts and conversation members. One button is used for connecting to Skype™, and there is one picture box for the avatar icon. Three labels are used for displaying the online status, the time of last online and the user mood message. The control at the top right hand side of the form is a FlowLayoutPanel, which is used for displaying the name of the participants and the voice bar in a conversation. A multiple line text box is used to display the messages for log and text communication. There are two combo selectors and they are used for selecting a speaker and microphone. Another multiple line text box is used as input for text communication. There are five buttons for communication control namely Send for sending text messages, Call for making an audio call, Accept for accepting an audio call, Hold for holding the current call and Mute for muting the microphone. AvatarBox is a picture box for showing a user image.

B.4 Game Map

The game map is the playing surface for a game. As part of developing a new game, a game map needs to be created. The game map must be a 1024 × 768 pixel image and this can be a JPEG. The size of the game map relates to the resolution of the Wiimote & projectors used with the RoboTable. There are two main ways to add your game map to the system as follows:

The most straightforward way is to add the game map through the **TableSetting** object and use the method UpdateGameMap to cause the system to load your game map from the TableSetting Object. When your table has been connected, you can use the method SendGameMapLocation to send the game map location to all other tables. The sample code is shown below. You should put your game map image in the folder of “GameMap”.

```
GameManager. TableSetting. GameMapImageLocation = “your project/bin/debug/GameMap”+ “\\” + “your game map name.jpg”
GameManager.UpdateGameMap();
GameManager.SendGameMapLocation (string Location);
```

A game map can also be drawn programmatically through the graphics methods provided by the GameManager class. The sample code is shown below.

```
GameManager.AddGraphicObject(DefaultRoboTableDrawingLayers.GameMap, mGameMap, "Background");
```

Note: the ”Background” is a key name for your game map which allows you to modify your game map with it. mGameMap is your game map object and the type of it is Bitmap object. A sample code to create game map dynamically is shown in Figure B.10.

```
//Create an empty game with the same size as the RoboTable screen.
//You can get the RoboTable screen size from the Table Setting of the game manager
Bitmap landmap = new Bitmap(GameManager.GameSetting.RoboTableScreenSize.Width,
GameManager.GameSetting.RoboTableScreenSize.Height);
//create a graphics object
Graphics g = Graphics.FromImage(landmap);
//initialise the background to white
g.Clear(Color.White);
//get the number of pixels for each row
_RowPixelSize = (float)(GameManager.GameSetting.RoboTableScreenSize.Height /
NumberOfGoalObjectOnARow);
//get the number of pixels for each collum
_CollumPixelSize = (float)(GameManager.GameSetting.RoboTableScreenSize.Width /
NumberOfGoalObjectOnAColum);
//the left top point
PointF ltop;
//the right bottom point
PointF rbot;
//A rectangel object
RectangleF rec;
//clear the initial goal object collection
this.GoalObjects.Clear();
//the number of goal object on a row and the number of goal object on a colum
this.TotalGoalObjects = NumberOfGoalObjectOnARow * NumberOfGoalObjectOnAColum;
bool ts = false;
//create all goal objects and drawing the game map as the same time
for (int y = 0; y < NumberOfGoalObjectOnARow; ++y)
 {
    for (int x = 0; x < NumberOfGoalObjectOnAColum; ++x)
      {
        ltop = new PointF(x * _CollumPixelSize, y * _RowPixelSize);
        rbot = new PointF((x + 1) * _CollumPixelSize, (y + 1) * _RowPixelSize);
        rec = new RectangleF(ltop, new SizeF(rbot.X - ltop.X, rbot.Y - ltop.Y));
        //create a new goal object
        GoalObject newGoalObject = new GoalObject(rec, 1, Color.Brown, Color.White);
        string goalobjectkey = ltop.X + "," + ltop.Y;
        newGoalObject.KEY = goalobjectkey;
        //create the game map on the graphic object g
        SolidBrush b = new SolidBrush(newGoalObject.CurrentColor);
        Pen p = new Pen(Color.Blue, 1);
        g.Clip = new Region(newGoalObject.GetLand);
        g.FillRectangle(b, newGoalObject.GetLand);
        g.DrawRectangle(p, newGoalObject.GetLand.X, newGoalObject.GetLand.Y,
newGoalObject.GetLand.Width, newGoalObject.GetLand.Height);
        //create the goal object's self image
        newGoalObject.UpdateGraphic();
        //add the new goal object to the game unility layer of the RoboTable graphics for
displaying when it has been discovered

       this.GameManager.AddGraphicObject(DefaultRoboTableDrawingLayers.GameUtility,
newGoalObject.GoalObjectBitmap, newGoalObject.KEY);
        //move the new goal object to the right location i.e. place them onto the right
cell of the grid
       this.GameManager.ChangeGraphicObjectLocation(DefaultRoboTableDrawingLayers.GameUtil
ity, newGoalObject.KEY, newGoalObject.GetLand.Location);
      //don't show them up yet until someone finds them and claims to own them
       this.GameManager.HideGraphicObject(DefaultRoboTableDrawingLayers.GameUtility,
newGoalObject.KEY);
        //add the new goal object to the initial goal object collection
        this.GoalObjects.Add(newGoalObject.KEY, newGoalObject);
      }
 }
 //dispose the graphic object
 g.Dispose();
```

Figure B.10 Sample code to create Game Map dynamically

The sample code in Figure B.10 is provided by the Skeleton project to create a game map for the Land Grabbing game in the game rule control class (section 4.3). Creating a game map dynamically requires some level of graphics drawing knowledge.

B.5 Develop Game Rules

After implementing the game map, the game rules for your game need to be developed. Game rules are the rules that game players need to follow and complete in order to win the game. The toolkits are independent of any game rules. Game rules need to use the methods provided by the toolkits to interact with the toolkits. The skeleton project uses BaseGameRuleControl class and BaseGoalObject class to implement sample game rules. The two base classes may not be suitable for other games. If the two base classes are not suitable for your game, you should create your own game rule control class and goal object class using the manual in Appendix C.

B.5.1 Implementation Overview

The following describes the general steps for game developers to implement their own game rules.

1. Create a number of goal objects. A goal object has a score or action associated with it. Goal objects, which relate to a location on the game map, should use the location as their key in a collection and game developers need to specify the location for their goal objects. Because the current toolkits only support graphics rendering in 2-dimensional space, the location should be pixels related.

2. Read the latest position information of a local robot and check if the local robot can achieve a goal object.

3. If the local robot can achieve a goal object, the goal object should set the local robot as the temporary owner. If there is an issue such as two robots trying to claim the same goal object, a protest goal message should be sent to all tables to ask for the correction of the temporary owner.

4. If the reconcile time is up and there are no more issues, the temporary owner of the goal will be changed to the permanent owner of the goal.

5. The final result is calculated based on the permanently owned goal objects.

For example, there are four apples located in a position in pixel of (1,1), (5,5), (10,10) and (20,20). Hence an apple with the key as its position is a goal object. If a robot runs over the location (10, 10), the robot will win the apple. If two robots claim the same apple on (10, 10), they might need to do

goal reconciling by checking their time stamp. The first robot that runs over the position of (10, 10) should win the apple.

Steps 3 and 4 employ a goal reconciling technique. If the game does not require reconciling, game developers can skip these steps and mark the claimed goal object to be the permanently owned goal object.

B.5.2 BaseGameRuleControl Class and BaseGoalObject Class

The toolkits provide some basic classes to help game developers develop their game rules. Section 4.3 provides a sample game rule control class that inherits from the BaseGameRuleControl class and a sample goal object class that inherits from the BaseGoalObject.

Figure B.11 the base structure of Game Rules

Figure B.11 presents a simple structure for game rules. The BaseGameRuleControl and BaseGoalObject are two abstract classes that the game developers can use. BaseGameRuleControl and BaseGoalObject classes provide basic structures to help developers build their own game rules and goal objects. A goal object is something that the players are going to interact with. For example, in the Land Grabbing game, a piece of land is a goal object. The BaseGameRuleControl class provides general interfaces and utility functions for game developers to create their own game rules.

The BaseGameRuleControl class provides three main methods for the game developer to implement.

- PostLocalRobot(Robot robot): add a received robot object into the game rule object in order to send goal claim message.
- PostGoalMessage(GoalMessage mGoalMessage): add a received goal message into the game rule object for reconciling of game results. This is a general method to ask local table to update claiming goals from other tables and it is specially designed for multiplayer game environment. For example, a robot has just got 1 mark, and then the robot needs to tell other tables it has got 1 mark. Hence, the robot needs to send the claiming goal via a GoalMessage to all tables. The method is mainly used to track goals activities for all tables, and it is important to let the local table to update all goal activities. Otherwise, there is no way to know which table will win the game.
- UpdateRobotList(): add the current Robot list into the Game rule object. The robot list contains all robot objects, and the game rule can access all robot information easily.

To implement goal objects and game rules, you need to create a sub-class of BaseGoalObject for each type of goal in your game and a sub-class of BaseGameRuleControl for controlling your game rules.

B.5.3 Sample Game rules

The game rules of the Land Grabbing game are outlined as below:

1. If a robot is inside a square, the game will try to claim the robot as the new owner of the square.
    a. If the square that a robot is trying to claim is already owned by another robot, the game makes no changes.
    b. If there are two robots trying to claim the ownership of a square simultaneously, the game will check the timestamp of the claim message. The first claimant becomes the owner of the square.
2. If the total play time exceeds 5 minutes or all squares are claimed, the game will finish.
3. The final results will be displayed on all tables.
4. The user can abort the game, drop the game or terminate the game at any time.

We use the game rule control class and goal object class of Land Grabbing game to explain how to develop game rules. The game rules for the Land Grabbing Game are in the LandGrabRule object, which is inherited from the BaseGameRuleControl class. The LandGrabRule class has the following main properties and methods:

- The number of land columns and land rows, which gives the total number of land objects we need.
- The reconcile time, which is the minimum waiting time in seconds until a land object is claimed permanently.
- A method to dynamically create a game map based on the number of land columns and land rows. The goal objects are also created in this method.
- A method to update the graphics objects in the graphics Utility layer so that a land that has been changed colour can be displayed.
- A method to enforce game rules.

The structure of the game rules is shown in Figure B.12.

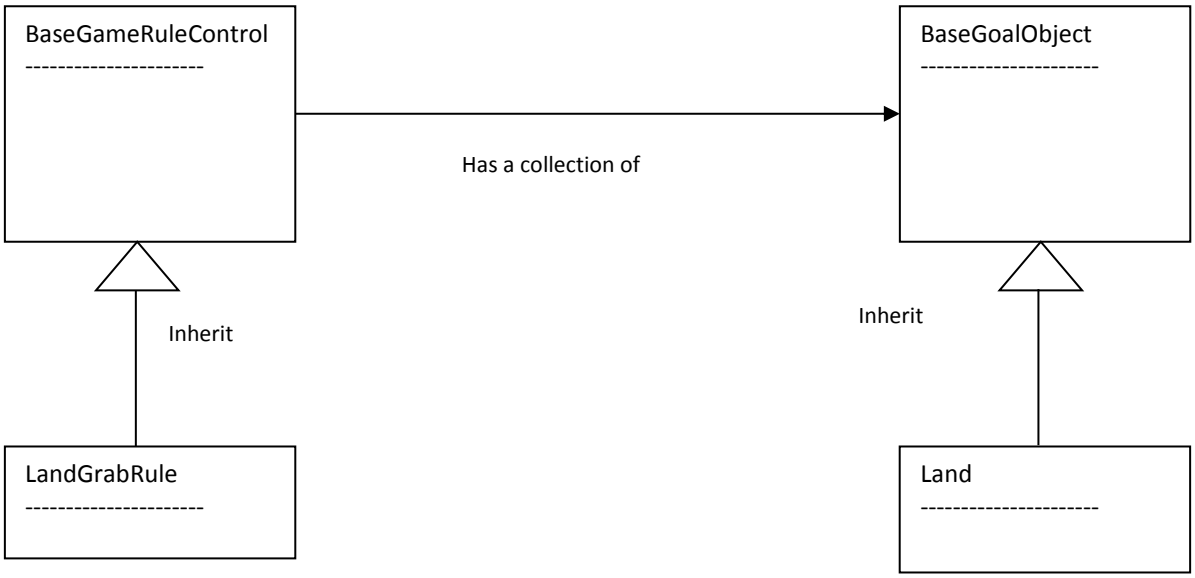


Figure B.12 Land Grab Rule structure

To implement your goals in your sub-class of BaseGoalObject you need to provide an overriding implementation of SetOccupied. If you game doesn't involve occupying anything, you can just provide an overriding implementation of this method with an empty code block such as {}.

The constructor of the BaseGoalObject is *BaseGoalObject(double mGoalScore)*. The mGoalScore is the score for your goal object.

The SetOccupied method is called if that goal object is occupied or achieved. It should update internal properties and SetOwnership to the robot passed in. An implementation of the Goal class "Land" from the Land Grabbing game is shown in Table B.1. The method CreateLandImage() is a method that is locally implemented for the Land class to create a Bitmap image for the Land i.e. to draw an image based on the passed RectangleF object with initial passed colour. The image object will be passed to the Game Utility layer (Appendix C) when the method CreateGameMapAndGoalObjects of the LandGrabRule object has been called (we will discuss this in Table B.2). CreateLandImage() provides a similar idea to create your own image for your goal object i.e. if it is an apple, you need to draw an apple or load an apple image.

Table B.1 The Land Class

<table>
<tr><td colspan="3">Class Land inherited from BaseGoalObject</td></tr>
<tr><td colspan="3">Initialising parameters: RectangleF mLand, double mGoalScore, Color TempOccupiedColor, Color UnoccupiedColor</td></tr>
<tr><td>Own Properties</td><td>Type</td><td>Description</td></tr>
<tr><td>LandBitMap</td><td>Bitmap</td><td>Hold a reference to the bitmap for this land</td></tr>
<tr><td>GetLand</td><td>RectangleF</td><td>Hold a rectangle representing the land</td></tr>
<tr><td>CurrentColor</td><td>Colour</td><td>Colour of the land</td></tr>
<tr><td>UnoccupiedColor</td><td>Colour</td><td>The default colour</td></tr>
<tr><td>OccupiedColor</td><td>Colour</td><td>The occupied Colour</td></tr>
<tr><td>TempOccupiedColor</td><td>Colour</td><td>Temp Occupied Colour</td></tr>
<tr><td colspan="3">Constructor</td></tr>
<tr><td colspan="3">public GoalObject(RectangleF mLand, double mGoalScore, Color UnoccupiedColor): base(mGoalScore)

The constructor requires a RectangleF object as a Land object, the goal value as the initial Land colour as the initial parameters for the goal object.</td></tr>
<tr><td colspan="3">Methods</td></tr>
<tr><td>SetOccupied (Robot robot, DateTime CurrentTime)
the method is overridden from the inherited based class</td><td colspan="2">try{
base.SetOwnerShipTo(robot.RobotID, CurrentTime);
if (robot.RobotColor != null)
_OccupiedColor = robot.RobotColor;
_CurrentColor = _OccupiedColor;
return true;
}catch(Exception e)
{
Utility.MTLogger.Error(LogType.Debug, e, "Land - SetOccupied");
return false;
}</td></tr>
<tr><td>CreateLandImage()
the method is the local method of the Land class. This method is used to create your own image for this goal object. i.e. if it is an apple, you need to draw an apple or load an apple image.</td><td colspan="2">Used to draw the land's bitmap.

if (_LandBitmap != null) _LandBitmap.Dispose();
_LandBitmap = null;
_LandBitmap = new Bitmap((int)this.GetLand.Width, (int)this.GetLand.Height);
Graphics Tempg = Graphics.FromImage(_LandBitmap);
SolidBrush b = new SolidBrush(this.CurrentColor);
Tempg.Clear(Color.White);
Tempg.Clip = new Region(this.GetLand);
Tempg.FillRectangle(b, this.GetLand);
Tempg.Dispose();
b.Dispose();</td></tr>
</table>

Once you have implemented the goal objects you need to implement the sub-class of BaseGameRuleControl. To explain this we show the implementation of "Land Grab Rule" in Table B.2, which is the game rule control class for the Land Grabbing Game.

The methods of DoActions, PostLocalRobot, CreateGameMapAndGoalObjects, AddToAGoalObjectCollection and UpdateGraphics are abstract methods of the BaseGameRuleControl class, and hence these methods must be implemented, other local methods such as getLandInfo and getCurrentLand are also created in our examples. The AddToAGoalObjectCollection method is important for game developers to implement because game developers need to implement a way to put their goal objects to the permanent goal object collection or the temporary goal object collection. For example, for the Line Following game, the goal object generated based on time will go to the permanent goal object collection directly. The final result is concluded based on the permanent goal object collection.

The sample codes, which is shown in the methods UpdateGraphics() and CreateGameMapAndGoalObjects (Table B.2), may look complicated. However, the algorithms for these two methods are straightforward.

The UpdateGraphics() method firstly merges the temporary goal list and the permanent goal list into a new temporary list and then changes the colour of the object in the temporary list according to the colour of the owner. When an object has been successfully painted, a flag is updated to its original object in the permanent goal list or original temporary list. The following sample code is used to make a hidden graphic object visible and change the colour of the graphic object.

```
this.GameManager.ShowGraphicObject(DefaultRoboTableDrawingLayers.GameUtility, pair.Key);

this.GameManager.ChangeGraphicObjectColor(DefaultRoboTableDrawingLayers.GameUtility, pair.Key,
((Land)pair.Value).CurrentColor);
```

The basic idea of the CreateGameMapAndGoalObjects() method is to create the game map and land goal objects, and save the goal objects in the initial collection i.e. GoalObjects. The following sample code is used to add a new graphics object i.e. the newly created Land object, to the game Utility layer, and then translate the new created graphic object to its actual location. Initially, all new graphics objects will be temporarily hidden.

```
this.GameManager.AddGraphicObject(DefaultRoboTableDrawingLayers.GameUtility, newL.LandBitmap,
newL.KEY);

this.GameManager.ChangeGraphicObjectLocation(DefaultRoboTableDrawingLayers.GameUtility, newL.KEY,
newL.GetLand.Location);

this.GameManager.HideGraphicObject(DefaultRoboTableDrawingLayers.GameUtility, newL.KEY);
```

The code UpdateGameMap((Bitmap)landmap.Clone()) is used to update the game map using the method provided by the base class.

The game result is controlled by the base game rule control class. The method for displaying the final game result, is DisplayFinalResult(), which is called by the DoActions() method (Table 12) when the finishing condition is matched. The constructor of the BaseGameRuleControl class is *BaseGameRuleControl(GameManager gamemanager)*. The parameter is a reference to your GameManager object that is created in your main game control form. The reason for this is that your game rule control class will need the methods provided by the GameManager class to interact your game rules with the toolkits.

Table B.2 The LandGrabRule class

<table>
<tr><td colspan="3">Class LandGrabRule inherited from BaseGameRule</td></tr>
<tr><td colspan="3">Initialising parameters: the Game Manager reference</td></tr>
<tr><td>Own Properties</td><td>Type</td><td>Description</td></tr>
<tr><td>Goal Score</td><td>Integer</td><td>The score for the current game</td></tr>
<tr><td>NumberOfLandRows</td><td>Integer</td><td>The value for the number of rows</td></tr>
<tr><td>NumberOfLandColumns</td><td>Integer</td><td>The value for the number of columns</td></tr>
<tr><td colspan="3">Constructor</td></tr>
<tr><td colspan="3">public GameRule(GameManager gamemanager)
: base(gamemanager)
{ //code your game rule constractor in here}

The constructor of this class uses the base constructor. The base constructor requires a reference to the GameManager object as the only parameter for the game rule control class.</td></tr>
<tr><td colspan="3">Abstract Methods from base class (the game developers need to implement these)</td></tr>
<tr><td>DoActions</td><td colspan="2">Called from the main thread, game developers need to implement this so that their game rules can update the graphics and output game results. The following is the sample code.
UpdateGraphics();
if (this.TotalGoalObjects == this.FinalOccupiedGoalObjects.Count)
{
Thread.Sleep(5000);//the final need to wait 5 seconds until all settle down
DisplayFinalResult();
}</td></tr>
<tr><td>UpdateGaphics()</td><td colspan="2">Used to update the utility graphic layer in the RoboTable graphic object. The game developer needs to implement this to interact with the graphic functions provided by the toolkits. The following is the sample code to update the graphics.</td></tr>
</table>

	if (this.GetCurrentGameMap != null)
	{
	lock (this.FinalOccupiedGoalObjects)
	{
	lock (TempOccupiedGoalObjects)
	{
	temp3 = Utility.MergeLeft(this.FinalOccupiedGoalObjects, TempOccupiedGoalObjects);
	}
	}
	if (temp3 == null) return;
	foreach (KeyValuePair<string, BaseGoalObject> pair in temp3)
	{
	if (((Land)pair.Value).IsPainted == false)
	{
	this.GameManager.ShowGraphicObject(DefaultRoboTableDrawingLayers.GameUtility, pair.Key);
	this.GameManager.ChangeGraphicObjectColor(DefaultRoboTableDrawingLayers.GameUtility, pair.Key, ((Land)pair.Value).CurrentColor);
	if (this.FinalOccupiedGoalObjects.ContainsKey(pair.Key))
	((Land)this.FinalOccupiedGoalObjects[pair.Key]).IsPainted = true;
	else ((Land)TempOccupiedGoalObjects[pair.Key]).IsPainted = true;
	}
	}
	temp3 = null;
	}

ReconcileProcess()	The game developers may need to implement this to do reconciling. This method runs in the second thread of the BaseGameRuleControl class. Game developers can use the reconcile method provided by the BaseGameRuleControl class or implement their own one by overriding it. The following sample is just calling its base reconcile method. You don't have to override this method. This method here is a sample. base.ReconcileProcess();
AddToAGoalObjectCollection	It is used to add a game goal object to the temporary box. Land mLand = (Land)GameObject; this.AddToAGoalObjectCollectionWithKey(mLand.GetLand.X + "," + mLand.GetLand.Y, mLand);

CreateGameMapAndGoalObjects	The game developers need to implement this to create the game map and the goal objects when the game is initialising.
	if (IsGameMapCreated == false)
	{
	Bitmap landmap = new Bitmap(GameManager.TableSetting.RoboTableScreenSize.Width, GameManager.TableSetting.RoboTableScreenSize.Height);
	Graphics g = Graphics.FromImage(landmap);
	g.Clear(Color.White);
	_RowPixelSize = (float)(GameManager.TableSetting.RoboTableScreenSize.Height / NumberOfLandRows);
	_CollumPixelSize = (float)(GameManager.TableSetting.RoboTableScreenSize.Width /

	NumberOfLandColums);
	//Rectangle rec = new Rectangle();
	PointF ltop;
	PointF rbot;
	RectangleF rec;
	this.GoalObjects.Clear();
	this.TotalGoalObjects = NumberOfLandRows * NumberOfLandColums;
	bool ts = false;
	for (int y = 0; y < NumberOfLandRows; ++y)
	{
	for (int x = 0; x < NumberOfLandColums; ++x)
	{
	ltop = new PointF(x * _CollumPixelSize, y * _RowPixelSize);
	rbot = new PointF((x + 1) * _CollumPixelSize, (y + 1) * _RowPixelSize);
	rec = new RectangleF(ltop, new SizeF(rbot.X - ltop.X, rbot.Y - ltop.Y));
	Land newL = new Land(rec, 1, Color.Brown, Color.White);
	string landkey = ltop.X + "," + ltop.Y;
	newL.KEY = landkey;
	SolidBrush b = new SolidBrush(newL.CurrentColor);
	Pen p = new Pen(Color.Blue, 1);
	g.Clip = new Region(newL.GetLand);
	g.FillRectangle(b, newL.GetLand);
	g.DrawRectangle(p, newL.GetLand.X, newL.GetLand.Y, newL.GetLand.Width, newL.GetLand.Height);
	newL.CreateLandImage();
	this.GameManager.AddGraphicObject(DefaultRoboTableDrawingLayers.GameUtility, newL.LandBitmap, newL.KEY);
	
	this.GameManager.ChangeGraphicObjectLocation(DefaultRoboTableDrawingLayers.GameUtility, newL.KEY, newL.GetLand.Location);
	
	this.GameManager.HideGraphicObject(DefaultRoboTableDrawingLayers.GameUtility, newL.KEY);
	this.GoalObjects.Add(newL.KEY, newL);
	}
	}
	g.Dispose();
	UpdateGameMap((Bitmap)landmap.Clone());
	landmap.Dispose();
	landmap = null;
	}
	IsGameMapCreated = true;

PostLocalRobot	The game developer needs to implement this to add a local robot reference to the game rule and check if it can occupy a goal object or not. The following is the sample code and the last line uses the base method PostRobot to finish the posting. This method firstly calculates the best matched Land goal object's key based on column and row pixel size. Secondly, the method tries to claim if the robot can get the best matched land goal object.
	
	PointF mltop = new PointF();
	mltop.X = (float)Math.Floor((mRobot.X / _ColumPixelSize)) * _ColumPixelSize;
	mltop.Y = (float)Math.Floor((mRobot.Y / _RowPixelSize)) * _RowPixelSize;
	string goalkey = mltop.X + "," + mltop.Y;
	
	this.PostRobot(mRobot, goalkey);

Abort()	The game developers may need to override this to tell the game rules how to abort the game. In this case study, we only need to call the

	method provided by the base class. You don't have to override this method, we put this method here as a sample. base.Abort();
GameFinish()	The game developers may need to implement this to tell the game rules how to finish the game. You don't have to override this method, we put this method here as a sample. Base.GameFinish();
Own Methods	
GetLandInfo(string GoalUniqueID)	Get a land object based on the goal unique id. mland = (Land)this.GetGameObjectByKey(GoalUniqueID);

GetCurrentLand(PointF location)	Get a land object based on the location (x,y). ColumnPixelSize holds a value of how many pixels for a column i.e. the width of a column. RowPixelSize holds a value of how many pixels for a row i.e. the height of a row.
	
	Land result = null;
	if (this.IsGameMapCreated == false)
	CreateGameMapAndGoalObjects();
	PointF mltop = new PointF();
	mltop.X = (float)Math.Floor((location.X / _ColumnPixelSize)) * _ColumnPixelSize;
	mltop.Y = (float)Math.Floor((location.Y / _RowPixelSize)) * _RowPixelSize;
	string goalkey = mltop.X + "," + mltop.Y;
	
	result = (Land)GetGameObjectByKey(goalkey);
	return result;

B.5.4 Thread Safety

Because the GameRuleControl (derived from the BaseGameRuleControl) class object is independent of the main control form and uses multithreading, we need to be careful to avoid issues such as the main control form and the game rule control class accessing an object concurrently. Hence, thread safety is an important issue. There are three objects that could be accessed by multiple threads when used the BaseGameRuleControl class and they are: GoalObject collection, TempOccupiedGoalObjects collection and FinalOccupiedGoalObjects collection.

The BaseGameRuleControl class has two independent threads, one is the main thread to update the graphics rendering on Game Utility layer via the Game Management Toolkit and check the game finish condition. The Game Management Toolkit manages the graphics rendering loop and provides a generic graphics interface to game developers. Hence, the main thread uses the graphics interface to update image objects' properties such location, rotation, colour and size. The other thread is used to send goal claiming messages and to do goal reconciliation. The reason we implemented two threads for the BaseGameRuleControl class is because the graphics updating may require more time while

the reconciling process has a strict time requirement. Hence, having two independent threads to process graphics updating and goal reconciling is a good idea for the BaseGameRuleControl class.

Implementation of the game rule control classes should bear thread safety in mind and make sure one thread only can access the above collections at a time. Using *lock(){…}* statements is a good way to control this.

B.6 Add Game Rules to Game Loop

To control the game flow, you only need to start the game engine (GameManager.Begin()), initialise the game engine(GameManager.Initialise()), wait for all tables to have finished preparing which includes connecting with the master table and offline activities such as building robots. The game engine will start the system state management routine when you call Begin() method. The Initialise method is required to tell the engine you have input some information to the game engine and need the game to transit to the next game state i.e. Game Preparing State.

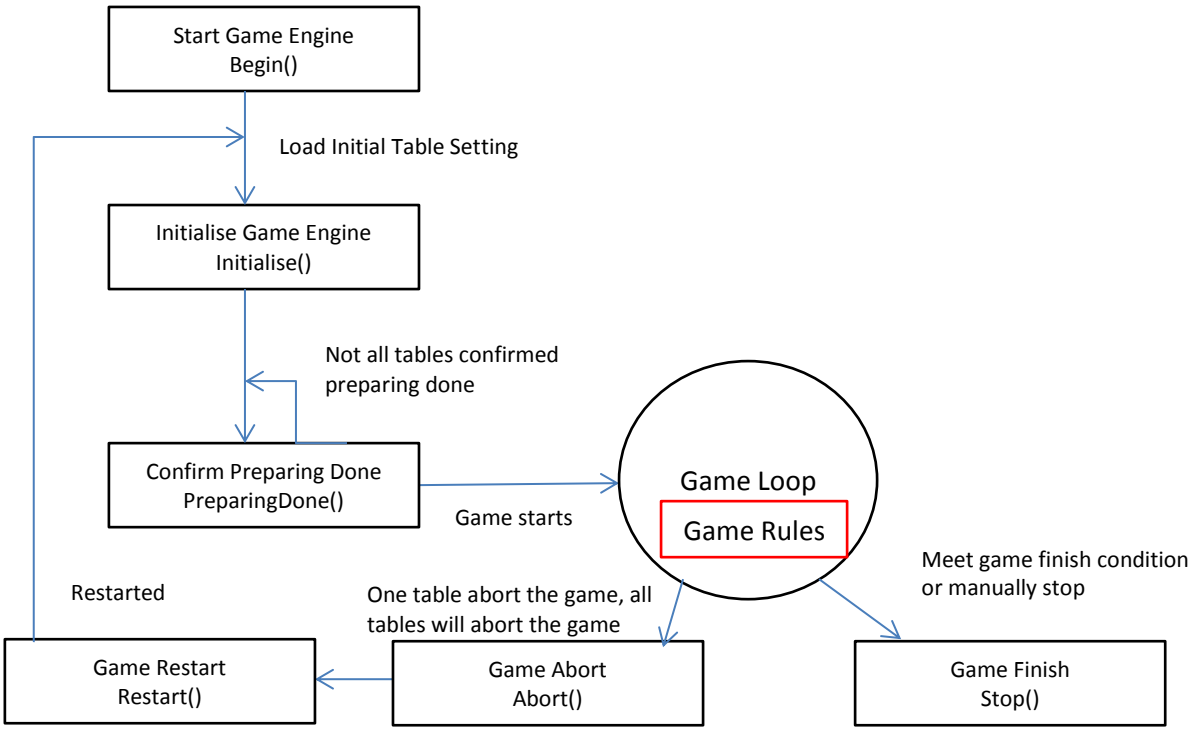


Figure B.13 Basic game flow

The game flow is illustrated in Figure B.13. To integrate the game rules and game goals with the skeleton game, the **LoadTableSetting** method needs to be updated along with some other methods. The code samples below show an example implementation of this.

To add your game rules to your game control form, you will need to update the LoadTableSetting method on the main control form provided by the skeleton project. You need to define your game rule class on your main control form as a variable.

```
private LandGrabRule _Landrule;

private void LoadTableSetting()

{

      ......

      _Landrule = null; //your game rule control class object

      //Create a new game rule control class object
      _Landrule = new LandGrabRule(_RoboTableGMT);

      //add an event to receive goal score update
      _Landrule.RobotScoreUpdateEvent += new
      UpdateGoalScoreEventHandle(RobotScoreUpdate);

      //set the game stop time in minutes
      _Landrule.GameStopTime = 5;

      //set the unit of goal score
      _Landrule.GoalScore = 1;

      //set the game rule details
      _Landrule.NumberOfLandColums = 10;
      _Landrule.NumberOfLandRows = 5;
      TotoScore = _Landrule.NumberOfLandColums * _Landrule.NumberOfLandRows;

      //set the minimum reconcile time in seconds
      _Landrule.ReconcileTime = 4;

      //create game map and goal object, you need call this method once
      _Landrule.CreateGameMapAndGoalObjects();
      ......
}
```

The next thing is to add robot update to your game rule control class object via the events provided by the GameManager class. The following sample code adds robot list update to the game rule control class object.

```
private void GameListUpdate(Dictionary<string, Robot> robotlist)
{
      //post received game list to your game rule control class
      _Landrule.UpdateRobotList(robotlist);

}
```

The following sample code adds robot update to the game rule control class object.

```
private void RobotUpdate(Robot robot)
{
      //post received robot update (from all tables) to your game rule control class
      _Landrule.PostLocalRobot(robot);

}
```

The following sample code adds goal message update to the game rule control class object.

```
private void UpdateGoalMessage(GoalMessage gmsg)
{
    //post received goal messages to your game rule control class here
    _Landrule.PostGoalMessage(gmsg);
}
```

This completes the implementation of the game however we encourage you to test the game before releasing it to ensure that it behaves in line with your expectations and requirements.

B.7 Testing and Debugging

In order to help locate any problems during testing and aid in debugging, the toolkits provide a log system MTlogger to log exceptions. An example of the use of the logger is shown in the code below.

```
catch (Exception re)
{
    Utility.MTLogger.Error(LogType.Debug, re, "fromControls - lvRobot_SelectedIndexChanged");
}
```

More information about the MTlogger can be found in Appendix D. The log file currently is located in “your project/bin/debug/LogFile/”and includes information on the type of error and what triggered it.

B.8 Summary

This guide has been designed to allow you with the help of a skeleton project to develop a distributed multiplayer game. The areas requiring most developer effort are on the implementation of Game Rules and Goals. To develop your own game from scratch is entirely possible. To do this we recommend looking through the code of the skeleton project and reading through the Appendices to this guide which have more in-depth information on the toolkit methods.

.

Appendix C

MRGT Developer Manual (Game Developer Guide)

C.1 Overview

The distributed multiplayer RoboTable game toolkits incorporate the Network Toolkit, the Robot Tracking Toolkit, the Communication Toolkit and the Game Management Toolkit. The Game Management Toolkit integrates the Network Toolkit, the Robot Tracking Toolkit and provides generic interfaces for game developers to develop games. The Communication Toolkit is an independent toolkit, which provides text and audio functions.

The toolkits contain a number of classes, but the game developers are required to work with MRGT.Base, MRGT.GameManagementToolkit and MRGT.CommunicationToolkit namespaces only. Table C.1 outlines the main classes of the MRGT toolkits.

Table C.1 MRGT toolkits, namespace and their main classes

Toolkit Name	Name Space	Main Class
Game Management Toolkit	MRGT.GameManagementToolkit	*GameManager* *GameStates* *BaseGameRuleControl* *BaseGoalObject* *TableSetting*
Network Toolkit	MRGT.Network	*TCPCommunicators* *ControlMessage* *NetworkToolkit* *NTMessages* *PeersMessage* *PositionMessage* *StateMessage* *GoalMessage* *TextMessage*
Tracking Toolkit	MRGT.TrackingToolkit	*WiimoteTracking* *Warper*
Communication Toolkit	MRGT.CommunicationToolkit	*SkypeToolkit*

C.2 System Requirements

The toolkits are developed using the .NET framework 3.5 and Microsoft Windows 7. We recommend that the game developers should run their RoboTable game on a multicore CPU. The final system requirements of their game will be depend on its complexity.

DirectX11 is required and hence the namespace of System.Drawing, Microsoft.DirectX and Microsoft.DirectX.Direct3D are required to be imported into a RoboTable game project.

C.3 Game management Toolkit

The Game Management Toolkit has overall responsibility for managing the state of the RoboTable games and is also the integration point for the Network Toolkit and Robot Tracking Toolkit. The toolkit contains three main classes, namely TableSetting , GameManager and GameState.

C.3.1 Table Setting

The TableSetting class is an attribute class and contains only properties. The class is designed for setting up a RoboTable game and holding all game relevant information. Table C.2 outlines the main properties of a Table Setting object.

Table C.2 The Properties of Table Setting

Property Name	Type	Required	Description
RoboTableName	string	Optional	The name of RoboTable
RobotName	string	Optional	The name of Robot
LocalIP	string	Yes	The local IP of the table
LocalPort	int	Yes	The local Port of the table
RemoteMasterIP	string	Optional	The remote master table IP
RemoteMasterPort	string	Optional	The remote master table port
IsMasterTable	bool	Yes	Set as a master table
IsVirtualTableOnly	bool	Yes	Set as a virtual table for demonstrating
GameMapImageLocation	string	Optional	Game basic map location
RobotImageLocation	string	Optional	Virtual Robot image location
CountDownSeconds	int	Optional	The seconds for count down such as 10 seconds. The default value is 11 seconds
CountDownStartDelaySeconds	int	Optional	The number of seconds is used as waiting time before start to count down.

WindowsHandle	IntPtr	Yes	For display control handle such as picture box, form etc.
WindowsHandleSize	Size	Yes	The size of the control handle for example the size of the picture box for displaying game scene
RobotSize	Size	Yes	The size of virtual robot
RoboTableScreenSize	Size	Yes	The size of final screen used for displaying game scene on RoboTable. Currently its default value is 1024 * 768. Also referred to as the size of game map
TurnOffPerformenceLog	bool	Optional	Used to turn off the logging of performance
CalibrationDataFile	string	Optional	Used to specify the location of a calibration data file. The default location is KeyFile//
RunCalibration	bool	Optional	Used to force the game to run calibration. True: run calibration, False: do not run the calibration and use the calibration data file instead

C.3.2 Game Manager

The Game manager is the main class that the game developers need to work with it. The game manager works as a game engine and the game application needs to control the game engine.

Commands

Table C.3 outlines the main commands for controlling the game manager.

Table C.3 the main commands for controlling the game manager

Method name	**Description**
Begin()	Start the game engine. It will start the main thread to manage the system states.
Initialise()	Initialise the game and load the initial Table Setting.
PreparingDone()	When this command is executed, the game will go to countdown state, which means that game play starts
Abort()	Abort the game during game playing session
GameStop()	Stop the game and display game result
Restart()	Restart the game and go back to the preparing state
StopLooping()	Stop the game looping
GameFinish	Finish the game and output the final result
InfoStored	Tell the system the current game info has been stored
Stop()	Stop the whole game immediately and reset the states to the

	beginning
GetCurrentSystemState()	Get current system state in a string format of state name + " and the current step is " + state step description

Default graphics layers

Currently the game manager provides six default graphics layers for game developers to interact with the game graphic rendering. The lowest layer is reserved for initial game information such as displaying the name of the game and the name of the game developer. The second layer is reserved for the game map, and the third layer is reserved for game utility such as a new map that can be changed with different land colours. The game info layer can be used to display the state information. The Robot layer is the principal layer that the game engine needs to manage the movements of the robots. Figure C.1 outlines the structure of the default game graphics layers. The layers marked with the game engine should leave for the game engine to accomplish.

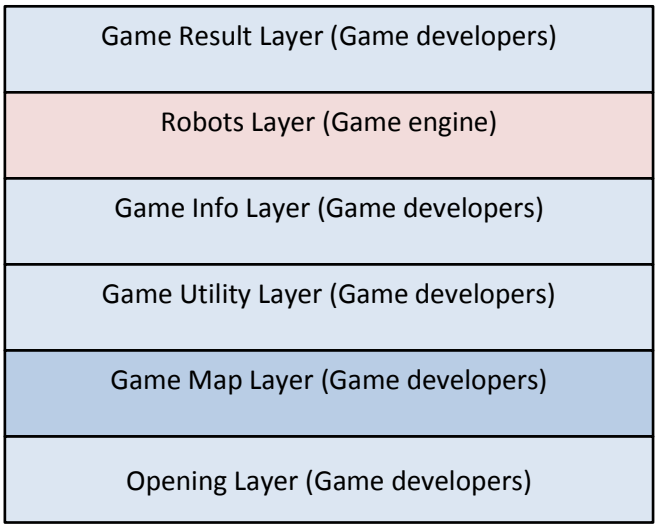


Figure C.1 The Default Layers for RoboTable Scene

Graphics Object

A graphic object is an object that will display on a RoboTable scene. A graphics object can be rotated, translated and scaled. The graphics object content is usually an image. Currently the toolkits only provide one graphics object class for a Bitmap type of image. However, the game developer can implement their own graphics object classes by inheriting the base graphic drawing class.

A graphics object has a key name, and the game developers can use the key name to control the graphic object. To control a graphics object in the RoboTable scene, the game developer needs to identify in which layer the graphics object is located and the key name. Table C.4 outlines the main methods for graphics management.

Graphic Methods

Table C.4 The main methods for graphic management

Method name	Description
AddGraphicObject	Add a bitmap or the location of a bitmap as a graphics object to the game engine
IsGraphicObjectExist	Check if a graphics object exist or not
RemoveGraphicObject	Remove a graphics object
HideGraphicLayer	Hide a graphics layer
ShowGraphicLayer	Show a hidden graphics layer
ChangeGraphicObjectLocation	Change the location of a graphics object
ChangeGraphicObjectColor	Change the colour of a graphics object
HideGraphicObject	Hide a graphics object
ShowGraphicObject	Show a hidden graphics object
RotateGraphicObject	Rotate a graphics object
ScaleGraphicObject	Scale a graphics object
GetGraphicObject	Get a graphics object
UpdateGameMap	To reload a game map from the GameMapImageLocation of the Table Setting

Game Messages

Table C.5 outlines the main methods provided by the game manager for sending game messages to all tables.

Table C.5 The Main Methods for Sending Messages

Method name	Parameters	Description
UpdateTimeLagOfPositionMsgSendInMs	Int TimeLagInMs	Used to skip a few position messages and reduce the update rate
SendGameMapLocation	string GameMapUrl	Send the game map image location
SendLocalRobotImageLocation	string LocalRobotUrl	Send the local Robot image location
SendLocalRobotColor	color RobotColor	Send the robot colour that is picked by users

SendMessageToTables	BaseMessage Msg	Send custom messages to all tables
SendMessageToSingleTable	BaseMessage Msg string TableIP, int TablePort	Send custom messages to a single table
SendClaimGoalMessage	int mGoalUniqueID Datetime ClaimTimeStamp	Send a message to claim a goal (status = 0)
SendProtestGoalMessage	GoalMessage original Datetime PosterTimeStamp	For reconciling only, a protester claims his/she should have the goal, not the original goal claimant (Status=1)
SendConformedGoalMessage	int mGoalUniqueID int ConfirmedRobotID Datetime ConfirmedTimeStamp	For reconciling only, used to confirm the goal ownership (Status =2). This is not implemented in the current base game rule control class.
SendTextMessage	string TextMsg	Send a text message
SendPositionMessage	Int X, int Y, double Orientation	Send a position message (used for evaluation purpose)

C.3.3 The Development of Game Rules

A game must have game rules. Developing game rules is the responsibility of the game developers, and this is going to be the most challenging part. However, the toolkits provide some basic classes to help the game developers to develop their game rules more easily.

Base game rule control class and Base goal object Class

The base game rule control class and the base goal object class are the two abstract classes that game developers can use. The classes of base game rule control and base goal object provide a basic structure to help the game developers to build their own game rules and game goal object straightforwardly. A goal object is something that the players are wishing to achieve. For example, in the land grab game, a piece of land is a goal object. The base game rule control class provides general interfaces and utility functions for game developers to create their own game rules. There are three main methods for game developers to implement and use. One is *PostLocalRobot(Robot robot)* for posting a robot object into the object of a game rule control class in order to send a goal claim message. The second is *PostGoalMessage(GoalMessage mGoalMessage)* for posting a goal claiming message into the object of the game rule control class and the third is *UpdateRobotList(Dictionary<string, Robot> RobotList)* for updating the latest robot list. PostGoalMessage method is used to post a received goal message into the object of a game rule control class for reconciling. The BaseGoalObject class is outlined in Table C.6 and the BaseGameRule class is outlined in Table C.7.

Table C.6 BaseGoalObject Class Overview

Abstract BaseGoalObject		
Initialising parameters: double mGoalScore		
Properties	**Type**	**Description**
GoalScore	Double	Hold a score number
IsPainted	Boolean	Check if the goal object has been painted or not
KEY	String	Hold a key string of this goal object. This is used to identify the goal object. Normally the KEY is associated with the goal object location.
TimeStamp	DateTime	The owner timestamp of this goal object
OwnerID	String	The ID of this goal object's owner
IsOccupied	Boolean	Check if it is occupied
Methods	**Type**	**Description**
SetOwnerShipTo(Robot robot, DateTime CurrentTime)	Boolean	A method that is used to set the ownership of this goal object to a robot. It can be overridden by the sub classes.
SetOccupied(Robot robot, DateTime CurrentTime)	Boolean	An abstract method to set the ownership to a particular robot and the owner time stamp.

Table C.7 BaseGameRule Class Overview

Abstract BaseGameRule Class		
Initialising parameters: the Game Manager reference		
Properties	**Type**	**Description**
TotalGoalObjects (Base class)	Integer	The value indicates the number of goal objects for a game
IsStarted (Base class)	bool	Indicate whether the game starts or not
IsGameMapCreated(Base class)	bool	Indicate whether the game has created game map

Protected GoalObjects	Dictionanry<string,BaseGoalObject>	A collection of unoccupied base goal objects
Protected TempOccupiedGoalObjects	Dictionanry<string,BaseGoalObject>	A collection of temporarily occupied base goal objects
Protected FinalOccupiedGoalObject	Dictionanry<string,BaseGoalObject>	A collection of permanently occupied base goal objects
GetCurrentGameMap	Bitmap	Get the current created game map.
ThreadSleep	Integer	Indicates the main thread sleeping time
GameManager	GameManager	A reference of current the game manager created by the game.
Robots	Dictionanry<string,Robot>	A collection of current Robots in the game
ReconcileTime	Integer	Indicate when the game rule should move the goal object to the permanent collection.
GameStopTime	Integer	Get or set a time for game stop.

Methods(implemented by the base class)	
Start()	Start the threads of the game rules, so that one thread can keep updating the graphics and outputting the result and the other thread performs reconciling.
Stop()	Stop all threads
PostGoalMessage(GoalMessage mGoalMessage)	Post a goal message to the game rule control class. Because the method is unique, game developers are not required to implement their own version.
GetGameObjectByKey (string KEY)	Get a goal object (BaseGoalObject) by its key
UpdateRobotList (Dictionary<string, Robot> RobotList)	Post the current robot list into the game rule object

PostRobot (Robot mRobot, string GoalUniqueID)	Post a robot with a goal unique id to check if it can occupy the object. If yes the method will send a goal claiming message
DisplayFinalResult()	This method is used to display the final result.
Reconcile()	Does reconciling work
Abstract Methods (methods the game developers need to implement)	
DoActions	Called from the main thread. Game developers need to implement this so that their game rules can update the graphics and output game results.
UpdateGaphics()	Used to update the utility graphics layer in the RoboTable graphic object. Game developers need to implement this to interact with the graphic functions provided by the toolkits.
ReconcileProcess()	Runs in the second thread and game developers can use the reconcile method provided by the base game rule control class or implement their own one. The game developers may need to override this.
AddToAGoalObjectCollection	It is used to add a game goal object to the temporary box. Land mLand = (Land)GameObject; this.AddToAGoalObjectCollectionWithKey(mLand.GetLand.X + "," + mLand.GetLand.Y, mLand);
CreateGameMapAndGoalObjects	Game developers need to implement this to create the game map and the goal objects when the game is initialising.
PostLocalRobot	The game developers need to implement this to post a local robot to the game rule and check if it can occupy a goal object or not.
Abort()	The game developers may need to override this to tell the game rules how to abort the game.
GameFinish()	An abstract method that the game developers may need to override this to tell the game rules how to stop the game.

C.4 Utility Class

The utility class contains some static functions that may be useful for game developers. Table C.8 outlines the static functions of Utility class.

Table C.8 Utility static functions or methods

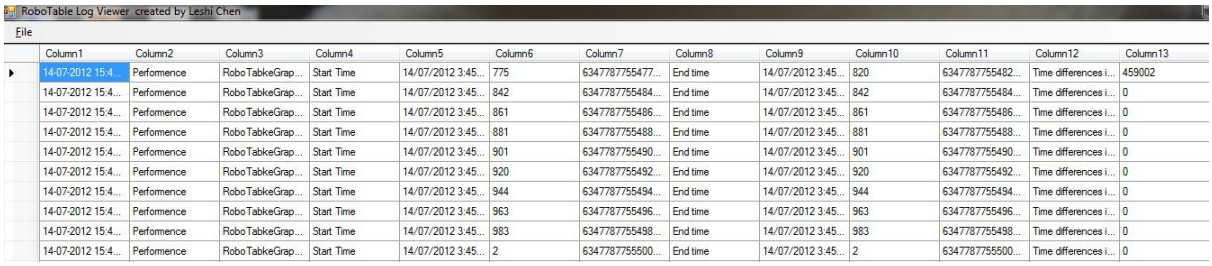
Method name	Return Type	Arguments (type name)	Description
IsIPv4	string	string IPv4	Check if it is IPv4
Ping	bool	string ip int timeout	Ping remote ip
IsTcpPortAvailable	bool	Int port	Check is a tcp port available
RndPort	int	int MinPort int MaxPort	Random to get an available port
GetFilePathsOnCurrentDirectory	String[]	string FolderName, string[] ListofExtension	Get file paths on current directory
CheckIfAFormIsOpen	bool	Type FormType	Check if a form is opened or closed
GetLocalIP	string		Get local IP
ClossAlllForms()	void		A method to close all form objects
IPAddressValidCheck	RoboTableException	string RemoteIP, int RemotePort	A function to check if a ip address is valid
SubstringNE	string	this string str, int index, int length	A function to get a substring from a string
MergeLeft<T,K,V>	T	this T me, params IDictionary<K,V>[] others	This funciton is used to merge two dictionaries
GetAllPublicProperties.	Dictionary<string, object>	object aObj	A method to get all public properties from an object
MTLogger	MultiLogger		A static logging object
DegreeToRadian	double	double angle	Convert a degree to radian
RadianToDegree	double	double angle	Convert a radian to degree

Appendix D

MRGT Debugging Support (Game Developer Guide)

Debugging a distributed multiplayer application using compiled toolkits is difficult without detailed information. In order to help the game developers to digest the problems, we have developed a logging system (we called it MultiLogger) to log every exception that has been generated during the game running. The logging system uses System.Diagnostics.Trace class to write a log in the local hardware. Because the logging system will output to a local file only when the game closes, the logging system does not affect the game toolkits performance at all.

The logging system has five types of log, and they are Error, Warning, Info, Performance and Result. Each type can have seven output files, and they have been named as Debug log, Network log, Graphic log, Tracking log, Management log, Communication log and Result log. All log files have been added to the listeners of the System.Diagnostics.Trace object during the MultiLogger object initialising. The game developers can use the logging system to log their applications in order to digest the problems in the process of the game development. Figure D.1 presents a sample log from MTLogger. The log Result is used to output the game result when the game is going to finish.


RoboTable Log Viewer created by Leshi Chen

File

Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11	Column12	Column13
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	775	6347787755477...	End time	14/07/2012 3:45...	820	6347787755482...	Time differences i...	459002
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	842	6347787755484...	End time	14/07/2012 3:45...	842	6347787755484...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	861	6347787755486...	End time	14/07/2012 3:45...	861	6347787755486...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	881	6347787755488...	End time	14/07/2012 3:45...	881	6347787755488...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	901	6347787755490...	End time	14/07/2012 3:45...	901	6347787755490...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	920	6347787755492...	End time	14/07/2012 3:45...	920	6347787755492...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	944	6347787755494...	End time	14/07/2012 3:45...	944	6347787755494...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	963	6347787755496...	End time	14/07/2012 3:45...	963	6347787755496...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	983	6347787755498...	End time	14/07/2012 3:45...	983	6347787755498...	Time differences i...	0
14-07-2012 15:4...	Performence	RoboTabkeGrap...	Start Time	14/07/2012 3:45...	2	6347787755500...	End time	14/07/2012 3:45...	2	6347787755500...	Time differences i...	0

Figure D.1 Sample Log From MultiLogger

To catch the error exception and output to the log file, we can use the follow sample code.

```
catch (Exception re)
{
Utility.MTLogger.Error(LogType.Debug, re, "fromControls - lvRobot_SelectedIndexChanged");
}
```

The implementation of the Logging system is straightforward using System.Trace within System.Diagnostics namespace.

To read the log without a good tool may be difficult for game developers to digest the problems or performance issues. In order to help the game developers, we implemented a tool called LogViewer to read the log and sort the log based on a field. The LogViewer can convert the log into a csv file. Figure D.2 presents the user interface of the LogViewer.

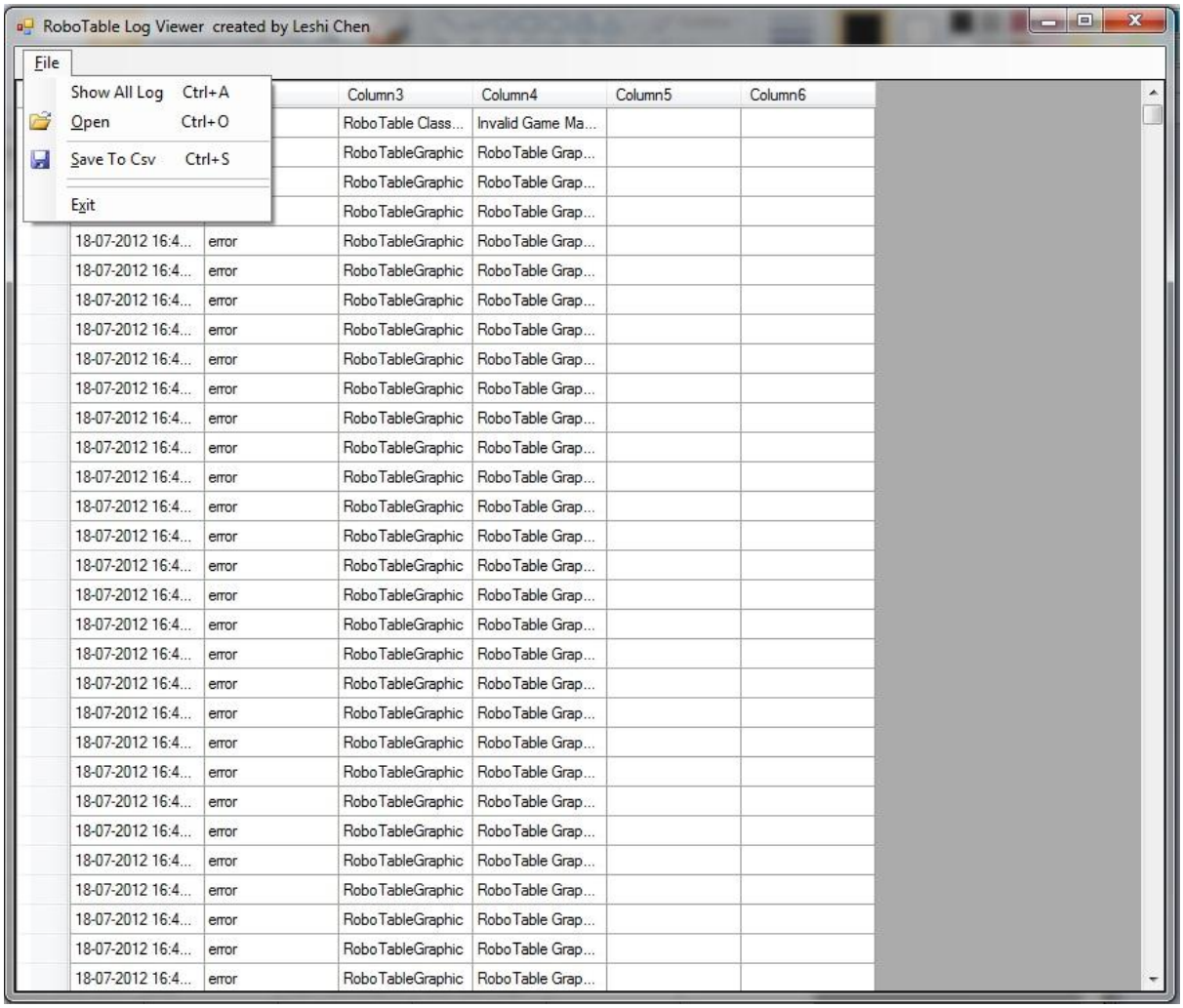


Figure D.2 LogViewer

In order to understand the format of the log, we explain the sample log shown in Figure D.3.

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11	Column12	Column13
2	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	175	6.35E+17	End time	5/07/2012 13:20	175	6.35E+17	Time differences in ticks	0
3	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	613	6.35E+17	End time	5/07/2012 13:20	613	6.35E+17	Time differences in ticks	0
4	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	34	6.35E+17	End time	5/07/2012 13:20	456	6.35E+17	Time differences in ticks	4218831
5	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	456	6.35E+17	End time	5/07/2012 13:20	456	6.35E+17	Time differences in ticks	0
6	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	472	6.35E+17	End time	5/07/2012 13:20	472	6.35E+17	Time differences in ticks	0
7	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	19	6.35E+17	End time	5/07/2012 13:20	19	6.35E+17	Time differences in ticks	0
8	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	144	6.35E+17	End time	5/07/2012 13:20	144	6.35E+17	Time differences in ticks	0
9	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	19	6.35E+17	End time	5/07/2012 13:20	19	6.35E+17	Time differences in ticks	0
10	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	159	6.35E+17	End time	5/07/2012 13:20	159	6.35E+17	Time differences in ticks	0
11	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	19	6.35E+17	End time	5/07/2012 13:20	19	6.35E+17	Time differences in ticks	0
12	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	175	6.35E+17	End time	5/07/2012 13:20	175	6.35E+17	Time differences in ticks	0
13	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	19	6.35E+17	End time	5/07/2012 13:20	19	6.35E+17	Time differences in ticks	0
14	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	175	6.35E+17	End time	5/07/2012 13:20	175	6.35E+17	Time differences in ticks	0
15	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	19	6.35E+17	End time	5/07/2012 13:20	19	6.35E+17	Time differences in ticks	0
16	5/07/2012 13:20	Performence	GameManager - ReceiveOperationalMsg	Start Time	5/07/2012 13:20	191	6.35E+17	End time	5/07/2012 13:20	191	6.35E+17	Time differences in ticks	0

Figure D.3 Sample Log

The values of the log are varied based on the log type and measure type. Generally speaking, the format of the log always contains the log time in column 1, the log type in column 2, the measure type in column 3 and the detail values across other columns. Normally, the column followed by the measure type is the “Start time”, start time value, start time value in milliseconds and start time value in ticks (column 4 to column 7). From Column 8 to column 11, there are “End time”, end time value, end time value in milliseconds, end time value in ticks. The last two columns are “Time

difference in ticks” and its value (the end time in ticks minus the start time in ticks). Each log value has its name on its pre-column so that we can understand them easily.

References

ABCYA.com. (2012). *Animation for Kids - Create an Animation Online*. Retrieved from http://www.abcya.com/animate.htm

Alwis, B. d. (2009). GT: A Groupware Toolkit for C#. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.184.1020

Alwis, B. d., Cutwin, C., & Greenberg, S. (2009). GT/SD: Performance and Simplicity in a Groupware Toolkit. *Proceedings of the 1st ACM SIGCHI symposium on Engineering Interactive Computing System, EICS 2009, Pittsburgh, PA, USA,*

Bharambe, A., Douceur, J. R., Lorch, J. R., Moscibroda, T., Pang, J., Seshan, S., et al. (2008). Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games. *SIGCOMM'08, August 17-22, 2008, Seattle, Washington, USA,*

Brickman, N., & Joshi, N. (2010). *HTN Planning and Game State Management in Warcraft II*: University of California at Santa Cruz. Retrieved from http://users.soe.ucsc.edu/~nishant/CS244.pdf

Burton, G. (2011). *Opening Messages*. Retrieved from http://www.information-age.com/channels/comms-and-networking/features/1674773/opening-messages.thtml

Chen, L.-y., & Li, C.-q. (2007). Design and Implementation of the Network Server Based on SIP Communication Protocol. *World Academy of Science, Engineering and Technology, 31 2007,*

Cowell, A. J., May, R., & Cramer, N. (2004). The human-information workspace (HI-Space): Ambient table top entertainment, Entertainment. *Entertainment Computing - Icec 2004, 3166*, 101-107.

Das, P. (2009). *AMQP (Advanced Message Queueing Protocol).* Paper presented at the 13th International Workshop on High Performance Transaction Systems, Oct 15- 28, 2009. Retrieved from www.hpts.ws

Facer, K. (2003). *Computer games and learning*. Retrieved Dec, 2011, from http://www2.futurelab.org.uk/resources/publications-reports-articles/discussion-papers/Discussion-Paper261

Festing, J. (2005). *The Development of Robotable Infrared Correction Techniques to Enhance Marker Detection.* Unpublished Bachelor Honours Thesis, Lincoln University, Christchurch.

Gerardo, P.-C., & Angelo, C. (2007). Analysis of the Advanced Message Queuing Protocol (AMQP) and comparision with the Real-Time Publish Subscribe Protocol (DDS-RTPS Interoperability Protocol): Real-Time Innovations, Inc.

http://www.omg.org/news/meetings/workshops/RT-2007/04-3_Pardo-Castellote-revised.pdf

HITLab. (2003). Human Interface Technology Laborator: HI-SPACE. Retrieved from http://www.hitl.washington.edu/projects/hispace/

Isbister, K., Flanagan, M., & Hash, C. (2010). Designing Games for Learning: Insights from Conversations with Designers. *Proceedings of the 28th international conference on Human factors in computing systems, April 10-15, 2010, Atlanta, Georgia, USA*. doi:10.1145/1753326.1753637

Klopfer, E., Osterweil, S., & Salen, K. (2009). Moving Learning Games Forward: Obstacles, Opportunities and Openness. *An Education Arcade paper, Massachusetts Institute of Technology*. Retrieved from http://www.educationarcade.org/

LaMothe, A. (2003). Tricks of the 3D Game programming Gurus. *Indianapolis:Sams Publishing,*

Linton, M. A., Calder, P. R., & Vlissides, J. M. (1988). Interviews: A C++ Graphical Interface Toolkit. *Technical Report CLS-TR-88-358, Stanford University,*

Makofske, D. B., Donahoo, M. J., & Calvert, K. L. (2004). *TCP/IP sockets in C#: Practical Guide for Programmers*: Elsevier Inc.

Mason, P. S. (2005). *The Development of Robotable A Hands-On Tabletop Environment to Support Engineering Education.* Unpublished Masters Thesis, Tufts University, June, 2005.

MSDN. (2012). *IntPtr Structure*. Retrieved 14/07/2012, 2012, from http://msdn.microsoft.com/en-us/library/system.intptr%28VS.71%29.aspx

Nagel, C., Evjen, B., Glynn, J., Watson, K., & Skinner, M. (2008). *Professional C# 2008*. Indianapolis: Wiley Publishing, Inc.

Nei, P. (2006). An open standard for instant messaging: eXtensible Messaging and Presence Protocol (XMPP). Retrieved from http://www.tml.tkk.fi/Publications/C/21/nie_ready.pdf

NTP.org. (2011, June 03, 2011). *NTP: The Network Time Protocol*. Retrieved from http://www.ntp.org/

Oblinger, D. (2004). The Next Generation of Educational Engagement. *Journal of Interactive Media in Education, 2004(8)*, 1-18.

Oblinger, D. (2006, January 1, 2006). *Simulations, Games, and Learning*. Retrieved from http://www.educause.edu/library/resources/simulations-games-and-learning

OpenVOSBlog. (2012). *IT Myths – TCP guarantees delivery of your data*. Retrieved from http://www.stratus.com/blog/openvos/?p=749

Papastergiou, M. (2009). Digital Game-Based Learning in high school Computer Science education: Impact on educational effectiveness and student motivation. *Computers and Education, 52*(1), 1-12.

Park, E.-Y., & Park, Y.-H. (2010). A Hierarchical Interface Design of a Puzzle Game for Elementary Education. *International Journal of u- and e- Service, Science and Technology, 3*, 43-50.

Pattie, C. (2004). *Optical Tracking for the Robotable Project.* Unpublished Bachelor Honours Thesis, Lincoln University, Lincoln, Canterbury, New Zealand.

Philip, L. (2003). *ARToolKit*. Retrieved from http://www.hitl.washington.edu/artoolkit/

Prensky, M. (2001). Fun, play and games: What makes games engaging? *In Digital Game-Based Learning, Chapter 5 of McGraw-Hill, New York, NY, USA.,*

Rodney, P. (1997). *Software Unit Testing*. Retrieved from http://www.slideshare.net/Softwarecentral/software-unit-testing

Roseman, M. (1993). *Design of a Real-Time Groupware Toolkit.* Unpublished Masters Thesis, The University of Calgary, Calgary, Alberta.

Saint-Andre, P. (2004). XMPP Design Guidelines. *the XMPP Standards Foundation, jabber.org,*

Schmidt, D. C. (1995). *Object-Oriented Network Programming with C++*. Retrieved from http://www.dre.vanderbilt.edu/~schmidt/PDF/acceptor-patterns4.pdf

Schmidt, D. C., Harrison, T., & Al-Shaer, E. (1995). Object-Oriented Components for High-speed Network Programming. *Proc.the USENIX Conference on Object-Oriented Technologies, Monterey, CA, June 1995,*

Schuller, D. (2005). *Game States*. Retrieved March, 2012, from http://www.godpatterns.com/2005/02/game-states.html

Singh, K., & Schulzrinne, H. (2005). Peer-to-peer Internet telephony using SIP. *in Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), Skamania,Washington, June 2005.,*

sourcemaking.com. (2012). *Design Patterns*. Retrieved from http://sourcemaking.com/design_patterns

Stansbury, M. (2009). *Can gaming change education? New research on gaming design and brain plasticity offers more perspectives on educational gaming*. Retrieved December 08, 2012, from http://www.eschoolnews.com/2009/12/09/can-gaming-change-education/

Tanenbaum, A. S. (2003). *Computer Networks(Fourth Edition ed.)*: Prentice Hall.

Vinoski, S. (2006). Advanced Message Queuing Protocol. *IEEE INTERNET COMPUTING* Retrieved from www.computer.org/internet

Young, R. M., Riedl, M. O., Branly, M., Jhala, A., Martin, R. J., & Saretto, C. J. (2004). An Architecture for Integrating Plan-Based Behavior Generation with Interactive Game Environments. *Journal of Game Development, 1*, 51-70.

Zhao, K. (2008). *An Architecture for Distributed Multiplayer RoboTable Games.* Unpublished Bachelor Honours Thesis, Lincoln University, Cantebury, New Zealand.