"Rock, Paper, Scissors": A Bluetooth Multiplayer Game

This paper describes how to write a game in Symbian C++ for S60 3rd Edition and UIQ 3 smartphones. The project describes the issues to consider when creating a multiplayer game that uses Bluetooth wireless technology to allow two players to play together.

This paper should be read in conjunction with the accompanying game code (see below).

Comes with Code: File:RockPaperScissorsGameSourceCode S60.zip
File:RockPaperScissorsGameSourceCode UIQ.zip

Introduction

Bluetooth has the advantage that the connection between two or more players is free - it does not require the expense of a data plan. Furthermore, all S60 3rd Edition and UIQ 3 smartphones support Bluetooth, and most users are familiar with its use – although it is still important for the game to make it easy to set up a connection between the players. The game can also be played in single-player mode, but multiplayer functionality is a nice addition to a mobile casual game, because it can be fun to play against people in the same room. Chapter five of Games on Symbian OS - A Handbook for Mobile Development discusses the benefits of adding a multiplayer component to a mobile game in further detail. The chapter also describes some of the issues to consider when designing and implementing local and remote multiplayer games.

The game code is provided for both S60 3rd Edition and UIQ 3, and signed software installation (SIS) files are available for each platform. The game can be played cross-platform (that is, a player using an S60 3rd Edition smartphone can play against an opponent playing on a UIQ 3 smartphone). For completeness, the game has also been tested in single-player mode on the Windows® emulator found in the S60 3rd Edition FP1 SDK and UIQ 3.0 SDK. It is also possible to attach Bluetooth hardware to the emulator for testing on the PC, and further details of how to do this can be found in Symbian OS Communications Programming.

Game Overview

Rock Paper Scissors

The game used in this paper is "Rock, Paper, Scissors" (RPS) which is also sometimes known as `RoShamBo`. The game was chosen because it is a two-player game with simple rules: when playing it in person, each player shows, with their hand, the element they have selected. This can be either a rock (clenched fist), paper (flat hand) or pair of scissors (index and middle fingers extended) at the same time. The objective is to beat the opponent: rock beats scissors by blunting them; paper beats rock by wrapping it; and scissors beats paper by cutting it. More details can be found on Wikipedia (en.wikipedia.org/wiki/Rock,_Paper,_Scissors), and on the official RPS site at www.worldrps.com. You can even play RPS against a online bots available on many gambling and gaming web sites.

In the example code we discuss in this paper, in single-player mode, the game uses very simple artificial intelligence to select an element. The static `Math::Random()` function is used to select a random number, and the modulus operator (%) used to derive a value between 0 and 2, which is taken as the smartphone's "choice". In code:

```
enum TElement {EInvalid = -2, ENone = -1, ERock = 0, EPaper = 1, EScissors = 2};
TUint32 randomChoice = (Math::Random()) % 3;
TElement smartphonePlayer = (TElement)randomChoice;
```

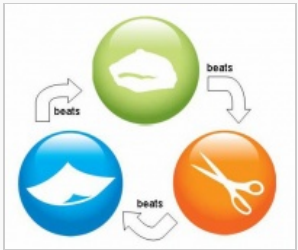

Figure 1 – The Rules of Rock Paper Scissors

That settles it: How to play rock, paper, scissors ... and win! New Scientist, December 2007. how-to-win-at-rock-paper-or-scissors describes the strategy one can adopt to improve the chance of winning at RPS. This, more complex, AI is used by online bots such as the `RoShamBot`, but was considered to be outside the scope of this particular example!

Symbian Platform Game Basics

The basics of writing a C++ game for Symbian smartphones are covered in chapter two of Games on Symbian OS - A Handbook for Mobile Development, which can be downloaded electronically, free of charge, from the book's website.

The RPS example game provided with this paper follows a very similar layout to the skeleton game loop described in that chapter, and uses a heartbeat timer to avoid running a tight synchronous loop. The `CRpsGameEngine` class contains the logic necessary to control the game loop, and to pause it when necessary, such as when the game loses focus or the user has been inactive for some time.

The `CRpsGameEngine` class, and much of the Bluetooth classes and code used in this example, are designed to be easy to re-use in your own game code. For example, the `CRpsGameEngine` class does not contain any logic to control the game screens, but defers all user input handling and control over what is drawn to the screen to the `CGameScreenManager` class. Figure 2 shows the class hierarchy for the RPS game in more detail – the Bluetooth classes are discussed in detail in Section 3.

Input Handling

The game has a very simple menu system and set of controls. It consists of a series of text screens that can be navigated using up and down keys (for example, those of a four way controller or navi-wheel) and selected using the standard selection key (for example, the centre key of a four way controller or by pressing down the navi-wheel). In addition, the '5' key on the phone keypad is handled in some game screens (for example, to continue after a connection drop) and on S60, the right soft key is used to exit the game. For simplicity, no touchscreen input is handled in the UIQ version of the game, although this could be added using the technique described in Games on Symbian OS - A Handbook for Mobile Development.

(C) Copyright Nokia 2013. All rights reserved.

Game Graphics

The graphics used in the game are also kept deliberately simple, relying mostly on text which is written to the screen. To account for the potential variations in screen size, resolution and orientation, the layout of each screen is not hard-coded. Instead, the dimensions of the screen are used to scale the layout as appropriate. Thus, for example, when writing two lines of text to the screen, the layout is as shown in Figure 3.

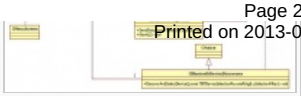

Figure 2 – RPS class hierarchy

To add some interest to the game, some very basic graphics are used - a bitmap is used for a timer-based splash screen, displayed on startup, and another bitmap displayed when the game is paused. The bitmaps are provided in MBM files, which are the most convenient graphics format on the Symbian platform, since they can be generated as part of a build. The MBM file can contain multiple BMP files, which may then be loaded synchronously into `CFbsBitmap` objects. In the RPS example, the bitmaps are loaded by calling `CEikonEnv::CreateBitmapL()` to return pointers to `CWsBitmap` objects which can then be very easily drawn to the screen by calling `CWindowGc::BitBlt()`. The `rps.mmp` file shows how to use the MMP file syntax to generate MBM files, while the `CSplashScreen` and `CPauseScreen` classes illustrate how to load the bitmaps and draw them to screen. Further information can also be found in the Symbian Developer Library found within each public SDK.

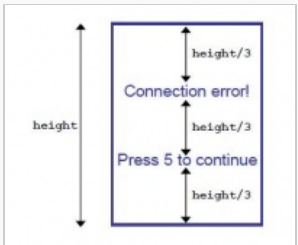

Figure 3 – Designing a scalable screen layout

Game Menu Screens

The text menu screens are provided by the abstract base class, `CMenuScreen`, which uses a doubly linked list (provided on the Symbian Platform by class `TDblQue`). The doubly linked list stores `TGameScreenItem` objects, each of which represents an item in a menu.

```
class TGameScreenItem
    {
public:
    TGameScreenItem(RPS::TMenuOption aOption, const TDesC& aText);
    TDblQueLink iDlink; // Double-linked list link
    RPS::TMenuOption iOption; // The menu item identifier
    TBool iHighlighted; // Indicates whether item is highlighted
    TInt iX; // X position on screen
    TInt iY; // Y position on screen
    TBufC<KMaxRpsTextElement> iText;
    };
```

The following code is used to construct the main menu screen for the RPS game:

```
void CMainScreen::ConstructL()
    {
    _LIT(KSingle, "Single Player");
    _LIT(KTwo, "Two Players");
    _LIT(KAbout, "About");

    TGameScreenItem* item = new (ELeave) TGameScreenItem(RPS::ESinglePlayer, KSingle);
    item->iX = KRPSScreensHAlignment;
    item->iY = gScreenHeight/3;
    item->iHighlighted = ETrue;
    iItems.AddFirst(*item); // iItems is the doubly linked list of CMenuScreen
    iIterator.SetToFirst(); // iIterator is used to iterate through iItems

    item = new (ELeave) TGameScreenItem(RPS::ETwoPlayers, KTwo);
    item->iX = KRPSScreensHAlignment;
    item->iY = gScreenHeight/2;
    TGameScreenItem* current = iIterator;
    ASSERT(current);
    item->iDlink.Enque(&current->iDlink);
    iIterator.Set(*item);

    item = new (ELeave) TGameScreenItem(RPS::EAbout, KAbout);
    item->iX = KRPSScreensHAlignment;
    item->iY = 2*gScreenHeight/3;
    current = iIterator;
    ASSERT(current);
    item->iDlink.Enque(&current->iDlink);
    }
```

You can find the full code for `CMenuScreen` and its derived classes in `rpsGameScreens.cpp`.

Local Multiplayer Games

When connecting over Bluetooth, devices must be within about 10 meters of each other to establish a connection. Once connected, data transfer rates are high compared to a cellular network, with low latency, of the order of 20-50 milliseconds, and the relative throughput of a Bluetooth connection is approximately 700 kbps.

In a Bluetooth piconet there is one master device and up to seven slaves. The master device is, by default, the one that initiates the connection and invites other devices. Slaves cannot connect directly to other slaves on the piconet, and all communications must go through the master device. In RPS, which is a two-player game, there is one master device and one slave device. The master device holds the state of the game and tells the slave device of any changes. If the slave device needs to make a change to the state of the game, it does so by making a request to the master device.

> **Tip:** In a similar game that allowed for more than two players, on receiving a request to update the state of the game, the master would then notify any other slave devices of the change.

`CBluetoothManager` contains the logic to support both the master role and the slave role. The decision to include both roles in the same class was acceptable because of the simplicity of the RPS game; there are only two players – one of which is the master and the other the slave – and neither device requires multiplexing (that is, managing many connections at one time). However, if your multiplayer game involves a Bluetooth piconet of more then two slaves, then the logic to handle the master and slave roles should ideally be separated into two different classes, with another class used to manage the communication between them.

The RPS's Bluetooth stack uses the RFCOMM protocol, which is a stream-based communications API that is implemented on top of the packet-based L2CAP protocol. Both RFCOMM and L2CAP are transparent to the player, but the choice could affect the performance of your multiplayer application. In some Bluetooth multiplayer games, L2CAP is chosen instead of RFCOMM to speed up the bidirectional communication by reducing the latency and avoiding the

(c) Copyright Nokia 2013. All rights reserved.

overheads involved in establishing a stream connection. However, the drawback is that L2CAP is a lower layer protocol than RFCOMM, which means that you need to provide your own support to use it. In the case of RPS, latency wasn't a big issue, so we decided to use RFCOMM in preference for this game.

CBluetooth Manager

The `CBluetoothManager` class is responsible for managing the Bluetooth piconet. When playing the game in two-player mode, the user is asked to choose between controlling the game (the master role) and waiting for a connection (the slave role). The game screen is shown in Figure 4.

Figure 4 - RPS in Two-Player Mode

The connection between the master and a slave is represented by the `CBluetoothConnector` class, and one `CBluetoothConnector` is created for each master/slave connection. The slave is represented by the `CBluetoothResponder` class. More details can be found below in the sections that discuss the `CBluetoothConnector` and `CBluetoothResponder` classes.

`CBluetoothConnector` and `CBluetoothResponder` share a common virtual base class, `CBluetoothConnectionBase` and provide implementations of `CBluetoothConnectionBase`'s two pure virtual functions `SendData()` and `StartL()`.

`CBluetoothConnectionBase` also provides an API to query if a `CBluetoothConnectionBase` pointer is to a master or slave object.

The RPS game implements a set of callbacks, such as `ConnectionErr()`, `DataReceived()` and `SendDataComplete()`, that take a `CBluetoothConnector/CBluetoothResponder` handle as a parameter, to allow `CBluetoothManager` to identify the caller. In RPS, there is only ever a single connection between two players, so this connection handle parameter is unused. However, the parameter is passed into the callbacks to make the code scalable for a game where more than two players are involved. For example, a game could display the presence of all players, and if a player abandons the game (for example by moving out of Bluetooth range and breaking the connection) then the `CBluetoothManager` needs to know which `CBluetoothConnector` has returned the error, in order to update the game's UI (for example, by greying out the player that left the game). We have left some commented code inside the `BluetoothManager.h` header file, to illustrate how to scale up the code for more than two players.

Device Discovery

`CBluetoothDeviceDiscoverer` is the class responsible for displaying to the master player the available Bluetooth devices in range, from which an opponent can be selected. The class hierarchy is shown in Figure 5.

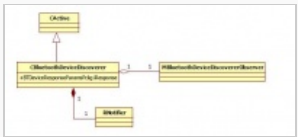
Figure 5 – Device Discovery

`CBluetoothManager` needs to implement the interface `MBluetoothDeviceDiscovererObserver` to receive callbacks from `CBluetoothDeviceDiscoverer`.

`CBluetoothDeviceDiscoverer` will notify `CBluetoothManager` of the result of the discovery, either by calling `OnDeviceDiscoveryErr()` to report any error that occurred during device discovery (for example, the player cancels the discovery dialog) or by calling `OnDeviceDiscoveryComplete(const TBTDeviceResponseParamsPckg& aResponse)` if the player successfully selected a remote Bluetooth device. In this case, aResponse will contain the remote Bluetooth device's address.

Device Discovery on S60 3rd Edition

On S60 3rd Edition, `CBluetoothDeviceDiscoverer` is implemented using the `RNotifier` dialog that is provided by the Symbian platform notifier framework.

If you decide to use customised Bluetooth discovery, to retain a look and feel consistent with the game rather than the standard S60 themes, you can instead use the `RHostResolver` class for discovery. The following steps are then necessary:

1. Initialise the name resolution service using `RHostResolver::Open(RSocketServ &aSocketServer, TUint anAddrFamily, TUint aProtocol)`.
2. Get the name of the host from the address using `RHostResolver::GetByAddress(const TSockAddr &anAddr, TNameEntry &aResult, TRequestStatus &aStatus)`.
3. Store the Bluetooth address in the `RunL()` and call `RHostResolver::Next()` to get the next available Bluetooth device.
4. When the Bluetooth discovery is finished, pass the array of Bluetooth device addresses to `CBluetoothManager` (using a callback) and then add the code to display them using a custom game UI.

The final steps are to allow the master player to select the Bluetooth devices to connect to, and create one `CBluetoothConnector` per Bluetooth device selected by the master player. This step is necessary regardless of whether the RNotifier approach is used, or if custom discovery is implemented.

Device Discovery on UIQ 3

In the RPS game, `CBluetoothDeviceDiscoverer` is implemented using the `CQBTUISelectDialog` dialog. The selection of the Bluetooth devices in range can be controlled by choosing one of the flags defined in `TQBTUISelectDialogFlags` (in `QBTselectdlg.h`). For multiple selections use `KQBTUISelectDlgFlagMultipleSelect`.

It is possible to implement a custom UI for device discovery on UIQ, in much the same way as for S60.

CBluetoothConnector

`CBluetoothConnector` is an active object. It has been implemented as a state machine that encapsulates all the Bluetooth stack parts that are responsible for managing the connection between a master and a slave. The class hierarchy is shown in Figure 6. The `CBluetoothConnector`'s state machine includes the following operations:

1. Searching for the slave's available services (`CBluetoothServiceSearcher`).
2. Connecting to the slave (`CBluetoothSockConnector`).
3. Both sending data to and receiving data from the remote device (slave) (`CSocketReader/ CSocketWriter`).
4. Callbacks to report to the `CBluetoothManager` any state machine error and for completion of events such as connection completed, report data, sending data completed (`MBluetoothObserver`).

In RPS, we have decided to advertise and filter our service discovery by service UUID only. If your application needs to make more complex filtering, you need to pass the UUID of the attribute you are interested in to `AddL(const TUUID& aUUID);`.

(c) Copyright Nokia 2013. All rights reserved.

CBluetoothConnector doesn't queue the data to send to the slave device but if your multiplayer game needs to do this, you can change CBluetoothConnector's implementation to support queuing of events.

CBluetoothServiceSearcher

CBluetoothServiceSearcher is responsible for searching for the presence of a specific Bluetooth service on a remote Bluetooth device. The class hierarchy is shown in Figure 7.

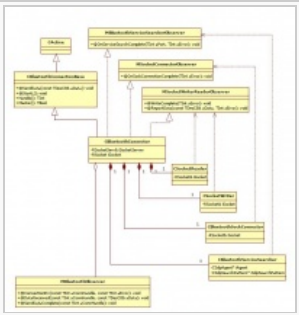

Figure 6 – CBluetoothConnector class hierarchy

The Bluetooth service class records are stored in the SDP database. In order to access/get SDP's records you need to use the CSdpAgent class provided by the Symbian platform. CSdpAgent can only be used if the address of a remote Bluetooth device has been identified. CBluetoothServiceSearcher must implement the MSdpAgentNotifier interface in order to handle SDP's responses.

In order to retrieve only the SDP record you are interested in, SDP provides the filter CSdpSearchPattern. CSdpSearchPattern is an array of TUUIDs (Bluetooth Universally Unique Identifier) that uniquely identifies a service class. A service class is added to the filter by repeatedly calling CSdpSearchPattern::AddL(). In RPS we are interested only in seeing if the remote device provides the RPS game service, but you could ask the SDP server to return all Bluetooth services that support the RFCOMM protocol, for example. After adding all the UUID you are interested in into CSdpSearchPattern, call CSdpAgent::SetRecordFilterL() to pass the array into CSdpAgent.

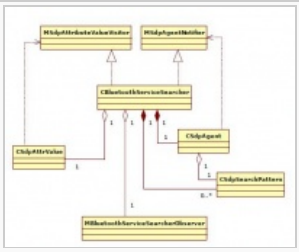

Figure 7 – CBluetoothServiceSearcher

In order to get the handle of the record on the remote device that matches the service class set previously in CSdpSearchPattern, use CSdpAgent::NextRecordRequestL(). This method is an asynchronous function which, when it completes, will call the callback NextRecordRequestComplete() from the MSdpAgentNotifier interface passed in the CBluetoothServiceSearcher::NewL().

> **Note:** Note that, as mentioned above, CSdpSearchPattern could contain more then one service class TUUID and you can get the next service record (if available) by repeatedly calling NextRecordRequestL().

If there are SDP records containing our class service, we need to browse the SDP record in order to retrieve the port number of the service. The port is then used by the master, with the remote Bluetooth address, to connect to the slave. CSdpAgent::AttributeRequestL() is an asynchronous call. When the attributes have been retrieved from the remote device, it calls AttributeRequestResult() from the MSdpAgentNotifier interface.

CBluetoothServiceSearcher uses the MBluetoothServiceSearcherObserver interface to report back to the caller (CBluetoothConnector) either the RFCOMM port or any error.

CBluetoothSockConnector

CBluetoothSockConnector is an active object responsible for connecting to a remote Bluetooth device on a given Bluetooth device address and on a given RFCOMM's port using the Bluetooth socket layer. The class hierarchy is shown in Figure 8.

> **Note:** Note that CBluetoothSockConnector uses the CBluetoothConnector's RSocket. This allows the connected socket to be passed to CSocketWriter/CSocketReader, if the connection is successful.

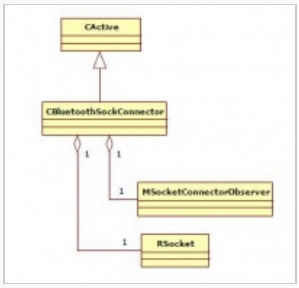

Figure 8 – CBluetoothSockConnector

CBluetoothSockConnector uses MSocketConnectionObserver to notify the caller (CBluetoothConnector) of the status of the connection.

CBluetoothResponder

CBluetoothResponder has been implemented as a state machine that encapsulates all the parts of the Bluetooth stack that are responsible for advertising the Bluetooth service using the SDP server, setting Bluetooth security, listening for an incoming connection and transferring data to/from a connected remote device (master). CBluetoothResponder acts as the slave part in a multiplayer game. See Figure 9 for details of the class hierarchy.

The CBluetoothResponder's state machine includes the following operations:

1. Finding an available RFCOMM port
2. Binding the Bluetooth socket to the RFCOMM port
3. Advertising the RPS server using the SDP server (see CBluetoothServiceAdvertiser)
4. Setting Bluetooth security
5. Listening for an incoming connection (master connection)
6. Both sending data to and receiving data from the master. See both CSocketReader and CSocketWriter.
7. Callbacks to report to CBluetoothManager the state machine error and completion of events such as listening for connection, connection completed, report data, sending data completed. (See MBluetoothObserver)

Like CBluetoothConnector, CBluetoothResponder returns the CBluetoothResponder's handle to when making the callbacks.

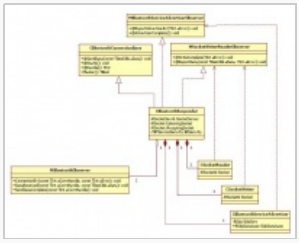

Figure 9 – CBluetoothResponder

CBluetoothResponder includes Bluetooth security support but for simplicity the RPS game does not require any authentication or encryption. CBluetoothResponder doesn't queue the data to send to the remote Bluetooth device but if your multiplayer game needs to do so then you can change CSocketWriter's implementation to support queuing of writes.

CBluetoothServiceAdvertiser

CBluetoothServiceAdvertiser has been implemented as a state machine that encapsulates the part of the Bluetooth stack responsible for advertising the Bluetooth service using the SDP server. The CBluetoothServiceAdvertiser's state machine includes the following operations:

1. Connecting to the SDP server and opening a SDP's RSdpDatabase subsession

(c) Copyright Nokia 2013. All rights reserved.

1. Connecting to the SDP server and opening a SDP's RSdpDatabase subsession.
2. Building the RPS's service record.
3. Notify the observer `MBluetoothServiceAdvertiserObserver` when the service advertising is completed successfully or if an error occurred.

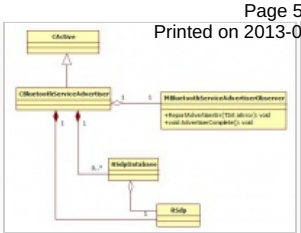

Figure 10 – CBluetoothServiceAdvertiser

CSocketReader/CSocketWriter

`CBluetoothConnector` and `CBluetoothResponder` both use the `CSocketReader` and `CSocketWriter` classes in order to listen for and send data. They are shown in Figure 11.

> **Note:** Note that `CSocketReader` and `CSocketWriter` make use of the `CBluetoothConnector` object's `RSocket`.

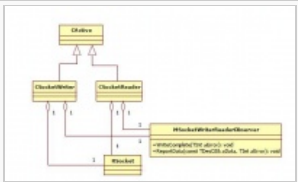

Figure 11 – CSocketReader/CSocketWriter

RPS game with more than two players

Whilst RPS has been designed for simplicity as a two player game, every effort has been made to design and implement the Bluetooth support using classes that could be reused/extended/modified in order to suit other Bluetooth multiplayer games for two players or more. If you would like to extend the RPS game by including more than two players, here are some tips on how to do so:

- You need to consider the following question: how is your game going to be played with more than two players? For example, we can have the master playing against all the slaves simultaneously and the results being displayed only when the master has received all the opponent choices. The master, as in the two player mode, is the one that starts the game. Another possibility is to fix the maximum number of players and accommodate all the players on the screen. In this scenario you could play simultaneously against any one of the opponents by selecting the part of the screen that player is displayed on.
- For S60 devices you need to implement the `CBluetoothDeviceDiscoverer` using the `RHostResolver` and display to the player the Bluetooth device in range in a customised multi choices control. On UIQ devices you need to use `CQBTUISelectDialog` in multiselection mode (see section 3.2.2 Device Discovery on UIQ 3). All the Bluetooth devices addresses that the player chooses to connect to must be cached in an array.
- The `CBluetoothManager` must create one `CBluetoothConnector` per opponent (slave) and store the base pointer in an array of `CBluetoothConnectionBase` pointers. The handle passed to the callback methods is used to identify players in order to handle disconnections and other errors.
- Error handling needs to be modified in order to finish the game either if the master player decides to or when the last slave player abandons the game.

Omissions

Versioning

The RPS game doesn't take in consideration different versions of the game because the data that is exchanged is only few bytes long. If your multiplayer game needs to exchange more complex data, you should consider adding an extra byte to include the version of the game.

Game Enhancements

We have deliberately kept the fundamentals of this example game very simple, and have focussed on the code needed for a robust Bluetooth multiplayer solution. Thus we've used text-based menus, basic bitmaps and very simple game play.

Some game enhancements that we've omitted, but may be added as an exercise for the reader include:

- Touch screen support for UIQ phones that support it.
- Attract mode, which is displayed after a period of user inactivity, before the game's pause screen is displayed.
- Support for dynamic change in screen orientation, in those phones that support a landscape to portrait switch (or vice versa).
- Additional graphics enhancements, for example, to add bitmaps corresponding to rock, paper or scissors for the player to select, and a dynamic graphic to indicate the amount of time the player has left to make the choice.
- The use of audio to provide sound effects and background music.
- The use of the vibra motor to add haptic feedback.
- The use of an accelerometer API (where available) as a novel way to provide input.
- Localization support to allow the game to be translated easily to support different languages and locales.

Games on Symbian OS - A Handbook for Mobile Development contains further information about how to add each of these enhancements.

Further Information

- Category:Books
- **Games Over Bluetooth: Recommendations to Game Developers v1.0.** This document provides recommendations for the use of Bluetooth in the development of multiplayer mobile games and can be found at www.Forum.Nokia.com.
- **S60 Platform: Bluetooth API Developers' Guide v2.0.** This document provides information on how to develop Bluetooth applications in C++. In the Symbian platfrom, the Bluetooth API consists of various components, and there are also some additional S60 APIs. The document describes how to perform typical Bluetooth tasks such as discoverability and service advertising, device and service discovery, and communication using different protocols. Security and platform security, configurations, and changes in the Bluetooth API v2 architecture (introduced in S60 2nd Edition, Feature Pack 2) are also described. The document provides several code snippets from separately published code examples, and can be found at www.developer.nokia.com.
- **S60 Platform: Bluetooth Point-to-Multipoint Example v2.0.** This C++ example demonstrates the use of Bluetooth technology: device and service discovery, connection establishment to one or more devices, communication between devices, and disconnection. The Bluetooth RFCOMM protocol (RS-232 serial port emulation) is used as a transport protocol. The code can be downloaded from www.developer.nokia.com.

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(c) Copyright Nokia 2013. All rights reserved.

(c) Copyright Nokia 2013. All rights reserved.