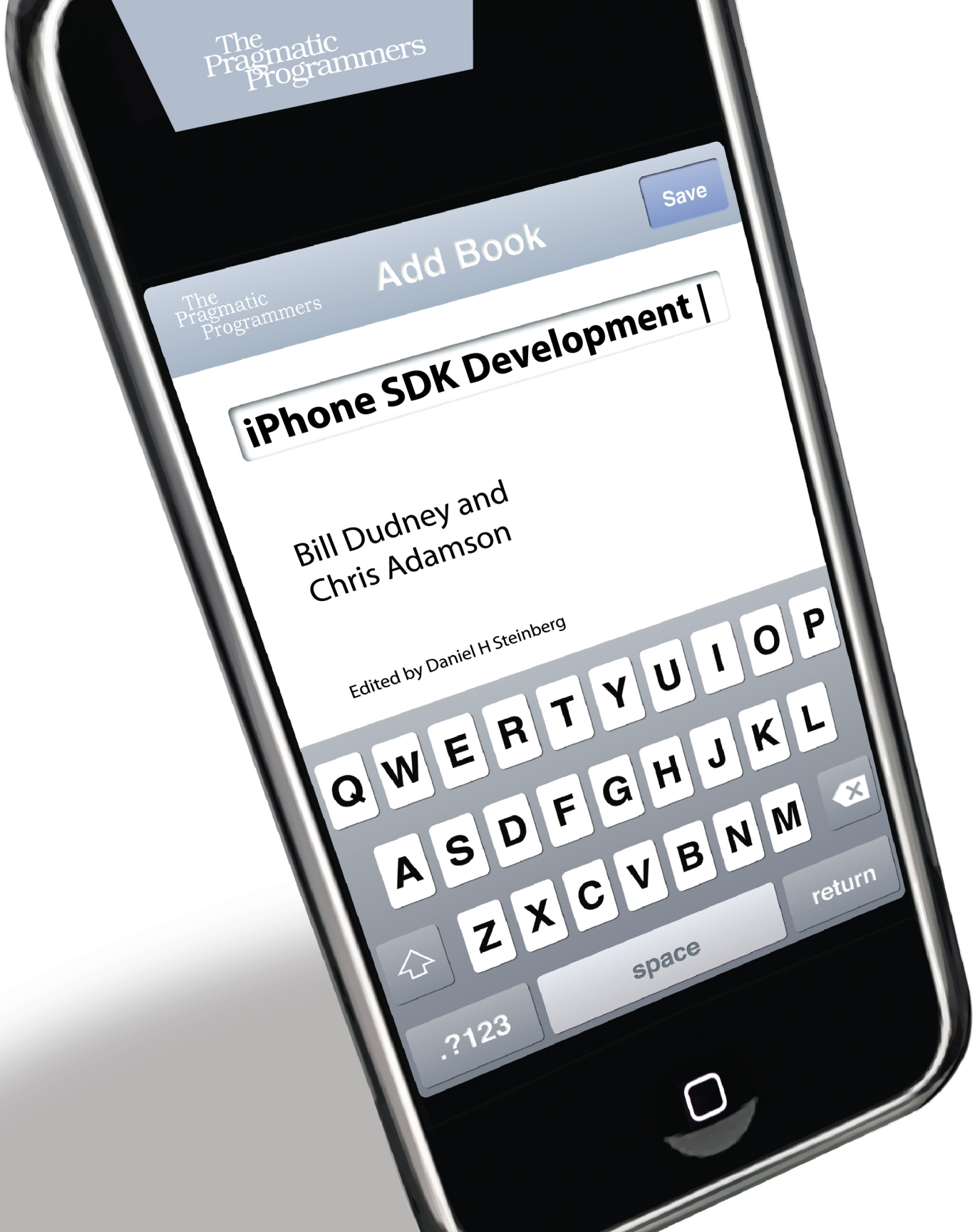
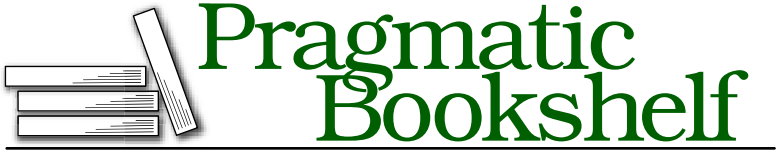
Extracted from:

iPhone SDK Development

Building iPhone Applications


The Pragmatic Programmers
Add Book
Save
iPhone SDK Development
Bill Dudney and
Chris Adamson
Edited by Daniel H Steinberg
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
space
return
.?123



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-25-5

ISBN-13: 978-1-934356-25-8

Printed on acid-free paper.

P2.0 printing, December 2009

Version: 2010-2-3

13.8 Communicating via the GKSession

Having mapped out a strategy for sending game data across Bluetooth, we can now implement our protocol with Game Kit's communication methods. We'll want to be able to handle state changes from peers (i.e., when the opponent connects or disconnects), send data to the opponent, and receive data from the opponent.

Sending Data

We need to send data to a peer every time the tap view is tapped, so let's go ahead and implement our handleTapViewTapped event handler:

Download NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(IBAction) handleTapViewTapped {
        playerTapCount++;
        [self updateTapCountLabels];
        // did we just win?
        BOOL playerWins = playerTapCount >= WINNING_TAP_COUNT;
        // send tap count to peer
        NSMutableData *message = [[NSMutableData alloc] init];
        NSKeyedArchiver *archiver =
                [[NSKeyedArchiver alloc] initForWritingWithMutableData:message];
        [archiver encodeInt:playerTapCount forKey: TAP_COUNT_KEY];
        if (playerWins)
                [archiver encodeBool:YES forKey:END_GAME_KEY];
        [archiver finishEncoding];
        GKSendDataMode sendMode =
                playerWins ? GKSendDataReliable : GKSendDataUnreliable;
        [gkSession sendDataToAllPeers: message withDataMode:sendMode error:NULL];
        [archiver release];
        [message release];
        // also end game locally
        if (playerWins)
                [self endGame];
}
```

This obviously calls a few internal game methods that we haven't written yet, starting with the call to update the score locally with updateTapCountLabels. The critical part of the method is after this, however: an NSKeyedArchiver is created to pack an NSMutableData with key-value pairs for our message. The updated tap count is added to the message, and if it equals the tap count needed to win the game, the END_GAME_KEY is added as well. We then call GKSession's sendDataToAllPeers:withDataMode:error: method in reliable mode if it includes the END_GAME_KEY, unreliably otherwise. Finally, there's a little more local logic to end the

game locally if necessary, with yet-to-be-written endHostedGame and endJoinedGame methods.

That takes care of the sending, but there's clearly quite a bit we haven't accounted for, including the receipt of messages and the game startup. These tasks aren't initiated by our application but are instead performed by the delegate methods, which handle asynchronous events from the session.

Handling State Changes

Let's start with session:didReceiveConnectionRequestFromPeer:, which is called when one party receives a request from another to connect. When the GKSession is connected via the peer picker, this callback is received only by the player who was asked to join the game, not by the one who chose the opponent in the picker. This gives us a chance to make the requesting player the *host*, a designation we use so that only one party actually starts the game.

Download NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
- (void)session:(GKSession *)session
            didReceiveConnectionRequestFromPeer:(NSString *)peerID {
        actingAsHost = NO;
}
```

Assuming that this player accepts the request, each side's delegates will get a callback to session:peer:didChangeState:, with the state GKPeerStateConnected. A number of other states can be reported this way, but for now, let's just implement some logic to set up the game when a peer connects:

Download NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
- (void)session:(GKSession *)session peer:(NSString *)peerID
        didChangeState:(GKPeerConnectionState)state {
    switch (state)
    {
        case GKPeerStateConnected:
                        [session setDataReceiveHandler: self withContext: nil];
                        opponentID = peerID;
                        actingAsHost ? [self hostGame] : [self joinGame];
                        break;
    }
}
```

When a connection is received, the first thing this method does is to call setDataReceiveHandler:withContext: on the GKSession. This is critical, because it gives the session an object that is capable of receiving data

over the network. The handler object is not specified with a formal protocol, but it has to implement a callback method with the following signature:

```
- (void) receiveData:(NSData *)data fromPeer:(NSString *)peer
               inSession: (GKSession *)session context:(void *)context;
```

setDataReceiveHandler:context: also takes a context that is passed back to the receiveData:fromPeer:inSession:context method. As a **void***, this context reference can be any kind of pointer, including all Objective-C objects. We don't need a context object for this game, so we set it to **nil**.

Next, our state-change handler remembers the peer ID of the opponent as the instance variable opponentID and either starts or joins the game based on whether this player is the host. Both of these methods need to update the local state and GUIs, but only the host needs to send a "start game" message over the connection. Here are the hostGame and joinGame methods, along with the initGame and updateTapCountLabels convenience methods they both call:

Download NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(void) updateTapCountLabels {
        playerTapCountLabel.text =
                [NSString stringWithFormat:@"%d", playerTapCount];
        opponentTapCountLabel.text =
                [NSString stringWithFormat:@"%d", opponentTapCount];
}

-(void) initGame {
        playerTapCount = 0;
        opponentTapCount = 0;
}

-(void) hostGame {
        [self initGame];
        NSMutableData *message = [[NSMutableData alloc] init];
        NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                initForWritingWithMutableData:message];
        [archiver encodeBool:YES forKey:START_GAME_KEY];
        [archiver finishEncoding];
        NSError *sendErr = nil;
        [gkSession sendDataToAllPeers: message
                        withDataMode:GKSendDataReliable error:&sendErr];
        if (sendErr)
                NSLog (@"send greeting failed: %@", sendErr);
        // change state of startQuitButton
        startQuitButton.title = @"Quit";
        [message release];
```

```
        [archiver release];
        [self updateTapCountLabels];
}

-(void) joinGame {
        [self initGame];
        startQuitButton.title = @"Quit";
        [self updateTapCountLabels];
}
```

In startGame, you can again see how we use an NSKeyedArchiver to build a message in an NSMutableData, which as a subclass of NSData is appropriate for use with the GKSession's sendDataToAllPeers:withDataMode:error: method.

Receiving Data

Now that we've handled state changes from opponents,[3] the last remaining task is to deal with the data we receive from a peer. We created the outgoing data with an NSKeyedArchiver, so to unpack it on the receiving end, we'll use an NSKeyedUnarchiver.

Download NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
- (void) receiveData: (NSData*) data fromPeer: (NSString*) peerID
                  inSession: (GKSession*) session context: (void*) context {
        NSKeyedUnarchiver *unarchiver =
                [[NSKeyedUnarchiver alloc] initForReadingWithData:data];
        if ([unarchiver containsValueForKey:TAP_COUNT_KEY]) {
                opponentTapCount = [unarchiver decodeIntForKey:TAP_COUNT_KEY];
                [self updateTapCountLabels];
        }
        if ([unarchiver containsValueForKey:END_GAME_KEY]) {
                [self endGame];
        }
        if ([unarchiver containsValueForKey:START_GAME_KEY]) {
                [self joinGame];
        }
        [unarchiver release];
}
```

As you can see, the unarchiver gets the data received by the GKSession and looks for some of the known keys. If it sees TAP_COUNT_KEY, it unpacks the value and updates the score display, whereas if END_GAME_KEY appears, it calls a method to end the game, cleans up the local

3. Actually, a fully robust app would want to handle some of the other state changes, such as gracefully dealing with a peer that has disconnected.

state, disconnects all peers from the GKSession, and calls a convenience method to show a victory or defeat alert, both of which are shown in Figure 13.6, on the following page.

Download NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(void) showEndGameAlert {
        BOOL playerWins = playerTapCount > opponentTapCount;
        UIAlertView *endGameAlert = [[UIAlertView alloc]
                initWithTitle: playerWins ? @"Victory!" : @"Defeat!"
                message: playerWins ? @"Your thumbs have emerged supreme!":
                        @"Your thumbs have been laid low"
                delegate:nil
                cancelButtonTitle:@"OK"
                otherButtonTitles:nil];
        [endGameAlert show];
        [endGameAlert release];
}

-(void) endGame {
        opponentID = nil;
        startQuitButton.title = @"Find";
        [gkSession disconnectFromAllPeers];
        [self showEndGameAlert];
}
```

That's everything you need to build and deploy this peer-to-peer Bluetooth game. To review, we used a GKPeerPickerController to present the user with a GUI to select an opponent. We provided the picker with a GKSession to handle the local Bluetooth networking and added delegate methods so this session could pass along asynchronous events like peers connecting. On the GKPeerStateConnected event, we set up the game, using the session to send data to the peer and providing the session with a "data receive handler" that could process incoming messages from the peer.

13.9 Voice Chat

Along with Bluetooth local networking, the other feature provided by Game Kit is peer-to-peer chat. As mentioned earlier, these two features are completely independent: you can use the voice chat with the Bluetooth network we set up in the previous sections or over a wifi connection that you've set up. Let's look in general terms at how voice chat works.

Voice chat uses just two classes. The GKVoiceChatService represents a single, shared access point to voice chat functionality. You get a refer-

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

iPhone SDK Development's Home Page
http://pragprog.com/titles/amiphd
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/amiphd.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)