Jamie Gotch's Thoughts on the Technology behind Fieldrunners

My interest in video games dates way back to the Commodore 64 era, with its 4-bit color and 64 kilobytes of RAM. I was never lucky enough to own a Nintendo Entertainment System during its heyday like most of my friends, but that may have been for the best. Having a computer opened up many more possibilities. As my interest in video games began to grow, I started to develop my own games. The Commodore 64 allowed me to do just that. Not long after I began to learn to program, I started to create my own games—and the rest is history. My entire academic career was focused on learning the art of programming and developing my math skills so that one day I could create something to call my own—Fieldrunners.

Taking the Plunge into Xcode and Objective-C

When I first started working on Fieldrunners, everything about developing for iPhone—Xcode, Objective-C, and even using OS X—was new to me. At the time, the majority of my development experience was writing C++ code on Windows. I had dabbled in various other languages and platforms before, but I never had the need, nor the desire, to write code on or for the Macintosh platform. That is, until the iPhone SDK was announced.

At first I was hesitant. I knew that moving over to such a platform would require a lot of my time and energy before I would become fluent enough in Objective-C to be productive. However, I also knew that the iPhone was a perfect fit for what I had in mind for Fieldrunners, a fast-paced tower defense game.

So I took the plunge and started learning everything I needed to create an iPhone game. I was quickly surprised by how easy everything was to use, from the operating system to Xcode, and even Objective-C. Of course, there were some challenges that I had to overcome, which led me to write this chapter to help you avoid the same mistakes I have made.

Where to Begin?

Creating a game from scratch can be pretty overwhelming. A lot of the decisions you make early on can drastically affect your end product. It is therefore very important to create a rough battle plan before you dive straight into writing code. To create this plan, you need to know what kind of game you are creating and determine your needs.

Determining Your Needs

Games come in all shapes and sizes. Casual games, in particular, have a much broader range of needs. For example, card games generally do not need to draw the screen at a high frame rate nor do they need several dozen sound effects to play simultaneously. Knowing what your application needs up front can save you lots of time and headaches in the long run. Here are some questions that can help determine the specific needs for your game:

- Will your game depend heavily on 3D graphics? Will it be sprite-based or use simple 2D shapes?
- Will your game need many constantly moving graphic elements on the screen at the same time? Or will it only have one or two moving graphics, with the rest of the graphic elements staying put most of the time?
- Will your game use many sound effects at the same time (i.e., explosions, gunfire, screams)? Or will you play sound only every few seconds?

  Your specific needs dictate which technology to use for your graphics and sound programming.

OpenGL ES vs. Quartz

One of the questions that I am asked the most by individuals starting to work on their own games is whether to use OpenGL or Quartz.

- Quartz is easier to use, because most of its concepts are learned in high school geometry. OpenGL programming, on the other hand, uses concepts, such as matrices and vectors, which are more complex.
- Quartz is great for applications that do not need to redraw the screen at a high frame rate and for applications that only need 2D graphics. OpenGL is designed explicitly with 3D graphics in mind. The iPhone GPU is capable of high-speed graphics rivaling the Sony PS2 and PSP.
- Quartz uses less battery power than OpenGL because OpenGL makes heavier use of the iPhone GPU.

OpenAL vs. AVFoundation vs. CoreAudio

Likewise, when taking stock of your sound and music playback, you have several choices. These are OpenAL, AVFoundation, and the CoreAudio frameworks.

- AVFoundation is simpler to use. You load your sound file and tell the player to play, pause, or stop your sound. You can create several AVAudioPlayer instances, load a different sound on each instance, and instruct each instance to play. On the other hand, this framework does not guarantee that it will play your sound with any degree of synchronization.

- CoreAudio gives you more control and lets you synchronize sound playback timing to a very fine degree, but it has a complex API.
- OpenAL requires specifying sound position in 3D space using matrices and vectors, similar to specifying the position and composition of your graphics with OpenGL. This makes it the more complicated API of the group.

Learning Objective-C

At first, I thought learning Objective-C would be very difficult. I looked at code examples and the strange syntax would send my mind spinning in circles. I could not make heads or tails out of it, even though I had substantial previous programming experience. Add to this the complexities of the Cocoa Touch API and its design patterns, and I found myself up to my neck with a daunting task and no end in sight.

I don't know what made things click, but I simply did not accept defeat and tried Cocoa tutorial after Cocoa tutorial, then one Mac OS X programming book after another (there were no iPhone development books back then). Eventually, it all clicked and gelled into my current understanding of the subject.

These days, there are several iPhone books, such as *Beginning iPhone Development: Exploring the iPhone SDK* by Dave Mark and Jeff LaMarche (Apress, 2009), developer blogs, and screencasts available to people interested in learning iPhone development. You even have access to university lectures about iPhone programming, given by experienced Apple engineers, for free! What I would give to have had these sources of information 18 months ago! No matter what your learning style is, there is something for everyone.

My only advice is not to give up and to keep looking for tutorials, books, and screencasts. There's no telling what things you'll learn from one source that you didn't learn before. I'd also recommend that you consider more than just iPhone resources—look for Mac OS X programming resources (such as the Apress book *Learning Objective-C on the Mac* by Mark Dalrymple and Scott Knaster, 2009), and the Cocoadevcentral.com web site. The programming environment is the same, Xcode, and uses the same language and design patterns. One benefit of Mac OS X programming resources is that the Mac OS X programming community is a decade old, and there are many more sources of tried-and-true tips and tricks to learn from.

As you read about Objective-C programming, the Cocoa frameworks, and Xcode from the several resources out there, you will also come across the importance of memory management methodologies on Objective-C. Let's examine that next.

Observing Memory Management Methodologies

The importance of memory management on the iPhone cannot be stressed enough. At the time of this writing, the iPhone and iPod Touch come with 128 MB of RAM. But most of this is taken up by the operating system and background processing, like checking for email and synchronizing your calendar with Exchange. On the iPhone, the background processes also include notification of SMS messages and incoming calls. At best, this leaves 70 MB of RAM for your game. If you use your iPhone heavily for browsing and email, the available RAM could be much less.

Many developers coming to iPhone and Objective-C programming in the last year had experience with Java and C# or with dynamic languages, such as Ruby or PHP. These environments have garbage collection built-in, hiding the ugly details of memory management from the developer. This is generally a good thing.

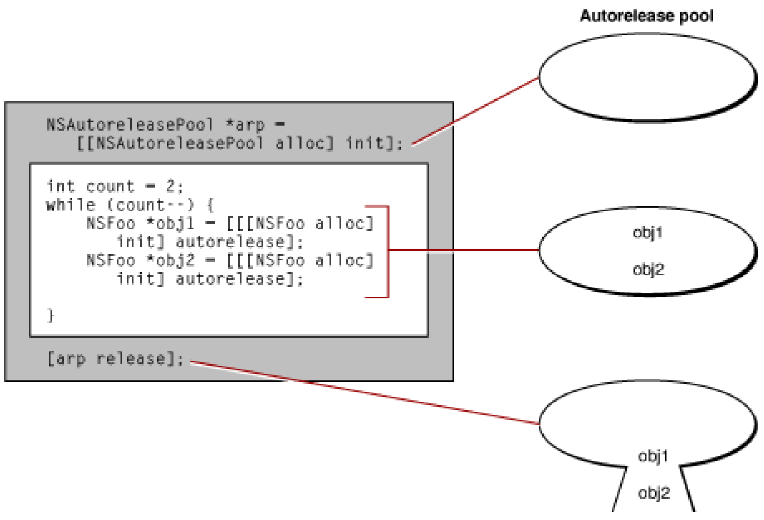
Objective-C, on the other hand, still has the developer handle memory management manually, like in C and C++ programming. Memory management on the iPhone is one of the most common sources of problems developers encounter. Even seasoned C and C++ programmers can be confused by the way Objective-C does memory management.

Objective-C uses a memory management methodology called *retain counts*. When you first create an object, there is a property called the retain count, which is initially set to one. During the lifetime of that object, every time you assign that object to a data structure, such as a collection, the retain count is incremented by one. There are other instances when you should increment the retain count yourself, such as when passing the object around to other parts of your code.

When you are done with an object you have retained, you have to remember to release it, or you will have what is known as a *memory leak*. Objective-C won't reclaim the memory used by that object until the retain count is set to zero. If you don't manage your retain counts properly, your game will use up the available memory on the iPhone, and the OS will quit your game at the most inopportune time.

Understanding Autorelease

Autorelease pools make it a lot easier to manage memory in Objective-C. By using an autorelease pool, objects you have marked as autorelease are added to the pool. When the pool is released at a later time, all objects in the autorelease pool are released as well. Without using an autorelease pool, it is the responsibility of the developer to manually release each object when it is no longer needed. Refer to Figure 1 for a visual representation.



```
NSAutoreleasePool *arp =
    [[NSAutoreleasePool alloc] init];

int count = 2;
while (count--) {
    NSFoo *obj1 = [[[NSFoo alloc]
        init] autorelease];
    NSFoo *obj2 = [[[NSFoo alloc]
        init] autorelease];

}

[arp release];
```

Figure 1. *Autorelease pool in action*

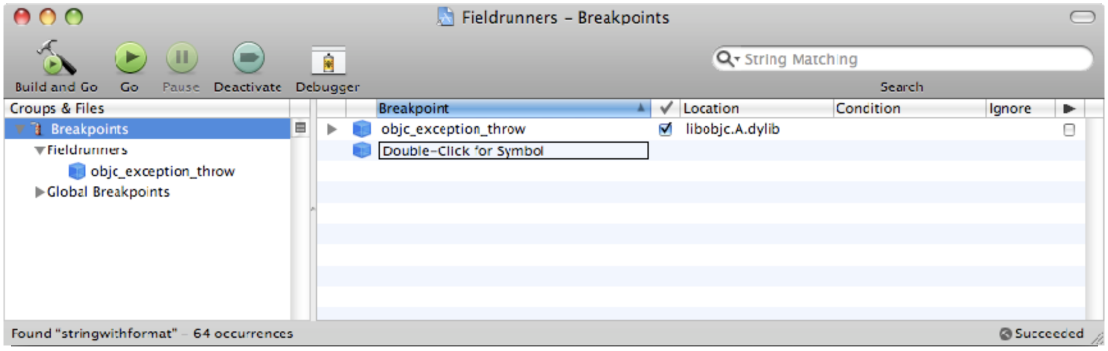
Note Because on iPhone OS an application executes in a memory-constrained environment, the use of autorelease pools is discouraged in methods or blocks of code where many objects are created. You should explicitly release objects whenever possible. Also, you do not need to set up an autorelease pool when developing applications using the Application Kit, as the Application Kit automatically sets up an autorelease pool that is scoped to the application's event cycle.

Besides the difficulties of learning Objective-C and memory management on iPhone, exceptions also caused me confusion. I found a handy trick that helps immensely with debugging exceptions on Objective-C code.

Breaking on Exceptions

This particularly handy trick is setting a breakpoint to catch exceptions. Objective-C throws exceptions by calling an Objective-C runtime function called obj_exception_throw. Setting a breakpoint on this function allows you to track down crashes by examining the call stack at the time of the crash. This is a great aid! No more head scratching and guessing at the reason for that EXC_BAD_ACCESS error. The debugger will show you the whole stack of calls down to the moment of the exception. You can jump right to the offending line of code that is responsible for the crash.

To set this breakpoint, open up the Breakpoints editor and create a new breakpoint called objc_exception_throw, as seen in Figure 2.



Breakpoint	✓	Location	Condition	Ignore
objc_exception_throw	✓	libobjc.A.dylib		
Double-Click for Symbol				

Found "stringwithformat" – 64 occurrences

Succeeded

Figure 2. *The Breakpoints editor in the Xcode debugger*

A* Algorithm

The A* algorithm is very powerful and versatile. It is primarily used to navigate characters around a map filled with obstacles, but it also can be used for various other purposes. A* was used in the implementation of Fieldrunners.

Many books have been written about the intricacies of the A* algorithm, which I will not get into in this chapter. This chapter introduces you to the algorithm and helps you understand when and where it can be used.

A Look at A* from 1000 Feet

The A* algorithm finds the shortest path between two nodes while avoiding obstacles. Although the A* algorithm does not need the nodes in a particular arrangement for it to work, it is easiest to learn how it works by laying the nodes out in a grid pattern. Refer to Figure 3 for an example. The white squares represent traversable blocks, while the colored blocks represent obstacles of different types.

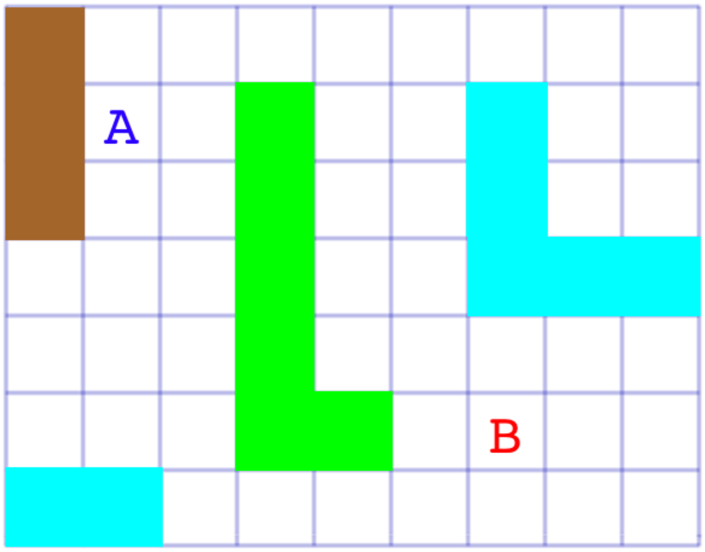


Figure 3. *The A* algorithm can find the shortest path from point A to point B.*

Some of the nodes on the map grid represent locations where game units can move. They could be roads, sidewalks, or alleys in a driving game. Other nodes are obstacles or simply parts of the map you cannot reach at this moment in the game. In a driving game, these other nodes could be buildings, fences around a park, or a river.

The map grid is just a representation of the graphics the user sees on his screen. It is not a part of the graphics on the screen. These map grid nodes can be high castle walls, steep cliffs, or water too deep for your game unit to swim. Or maybe they are just scenery, things on the map created "just for show" or to fill in the map. But in the game grid, these things are stored as objects in Objective-C or C++, as integers representing an object type in a game made in C, or if the developer is really twisted, as an index into a lookup table of function pointers to C functions that know how to draw the proper graphic on screen. In Model-View-Controller design pattern terms, the game grid is the game's model, while the graphics on the screen are the view.

How you determine which nodes the game unit cannot traverse is up to what kind of game you are developing. For example, in a war simulation, they are traversable by some units but not by other kinds of units (i.e., helicopters can move anywhere, but tanks can't cross deep lakes or climb steep cliffs). You create the map grid while designing your levels and refine it while integrating the level data into your game code.

A* Technical Implementation

The A* algorithm operates on the game grid, where each block on the grid represents a discrete location in space. In A* discussion, grid blocks are often referred to as *nodes*. A* manages two separate lists to track the nodes that have been examined in looking for a path to the goal and the nodes yet to be examined. These are referred to as the Closed list and Open list, respectively.

There is also an optional Moves list in some implementations, where the algorithm stores the nodes that lead from the starting node to the goal node. This Moves list is typical of games where game units move in turns. In games where game units move in "real time," such as the opponent AI in a driving game, the game AI only looks ahead a few steps. That kind of AI runs the A* in short cycles, moving to favorable locations as it finds them. In cases like these, the Moves list is not necessary.

When the algorithm is first executed, it empties the Closed list and moves through the game grid, beginning with the starting node and progressing one element at a time toward the goal, assigning each node and its neighboring nodes a weighted goal score. This weighted goal score is just a number that determines whether a particular node is closer to the goal than the other nodes around it. There are many algorithm variations just on computing this weighted goal score, and other variations focus on optimizing this initial sweep through the grid.

Once the goal score for the nodes is computed, the algorithm places the starting node onto the Open list. During each iteration, the algorithm removes the node with the best goal score from the Open list. This node becomes the current node and is examined. If the current node refers to the goal location, you stop processing, as you've reached the goal. If the current node does not refer to the goal location, each of the current node's neighboring nodes are examined. If the neighboring node is not found on the Closed list, you determine if it leads you closer to the goal. If it does, then it is added to the Open list. However, if the neighboring node is listed on the Closed list, it is ignored. After all of the neighboring nodes are examined, the current node is placed onto the Closed list to indicate that it has been completely explored. It is also added to the optional Moves list. If the Open list becomes empty before the goal is found, this indicates that there is no possible path to the goal. Refer to Listing 1 for an example of this algorithm.

Listing 1. *Pseudocode for the A* Algorithm*

```
Empty out the closed list
Compute weighted goal scores
Put starting node in open list

While the open list is not empty
  Current node = closest node to goal from open list
  If current node is the same as the goal
    Add current node to the moves list
    We're done. Break out of the loop
  End if

  If node above current node is not in closed list
    If it is not an obstacle and is closer to the goal
      Add it to open list if not in open list already
    Else
      Add it to closed list
    End if
  End if
  If node below current node is not in closed list
    If it is not an obstacle and is closer to the goal
      Add it to open list if not in open list already
    Else
      Add it to closed list
    End if
  End if
  If node to left of current node is not in closed list
    If it is not an obstacle and is closer to the goal
      Add it to open list if not in open list already
    Else
      Add it to closed list
    End if
  End if
  If node to right of current node is not in closed list
    If it is not an obstacle and is closer to the goal
      Add it to open list if not in open list already
    Else
      Add it to closed list
    End if
  End if

  Remove current node from open list
  Add current node to closed list
  Add current node to the moves list

End while loop
```

Once you reach the end of the loop, the Moves list contains the nodes the algorithm thinks led to the goal. If the Moves list does not contain the goal node, there is only a partial solution to reach the goal. If the Moves list only contains the starting node, there is no solution.

A* is a very versatile algorithm. While it is used mainly in games programming for AI path finding, it can also be modified for use in games that don't need path finding, such as games that need to sweep through a game board looking for scoring conditions. One such game type is falling block games. In the following section, *iPhone Games Projects* lead author PJ Cabrera discusses an implementation of such a game.

Puyo

In this section, I'll discuss a simple game implementation. I have in mind the game Puyo, a falling blocks game. It seems like a simple thing, but it is a complex game to implement, as you will see. This implementation uses a simplified A* algorithm to find and keep track of matching blocks of the same color.

The object of the game is simple. Randomly colored blocks appear in pairs at the top of the screen and gradually fall to the bottom. These blocks will continue to fall until they collide with an immovable object, such as another block, or the bottom of the play area. The player controls this pair of blocks and must strategically position and rotate it in order to clear the other blocks that litter the playing field. To achieve this, the player must link together four or more blocks of the same color. Links can only be formed by blocks that lie adjacent to one another horizontally or vertically, not diagonally. Every block that is part of the link is then removed from play. Any block that loses its grounding will begin to fall until it finds another surface on which to rest.

In the simplest Puyo implementations, there is no winning game condition. You simply keep playing until the play area is so full of blocks that there isn't room for them at the top of the screen. At that point, the game is over. Refer to Figure 4 for a screenshot of the game in action.

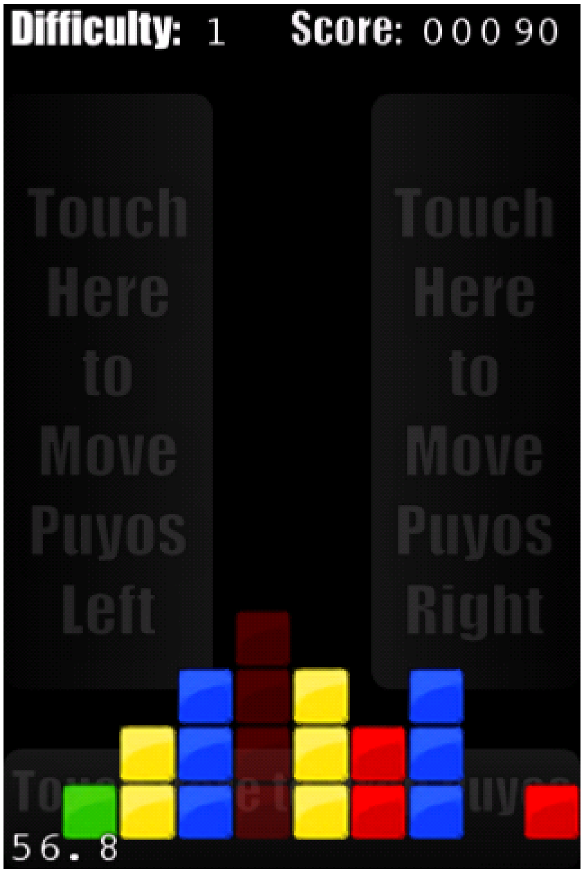


Figure 4. *The Puyo game in action. There is a group of four adjacent red blocks disappearing in this screenshot.*

Basic Code Design

This implementation of a Puyo game uses a few simple data structures. The game board is a simple C array of Puyo objects. To keep things simple, there are 13 rows of 10 Puyos each, but the game code is capable of handling more than twice as many Puyos without dropping frames.

Puyos are instances of an Objective-C class I created called Puyo, and have a number of properties: their position on the board, the Puyo color, whether the Puyo is stuck on another Puyo or at the bottom of the game board, and whether the Puyo has been marked to disappear by the part of the code that looks for groups of adjacent Puyos of the same color.

The algorithm for sweeping through the game board looking for groups of adjacent Puyos is a simplified variation of the A* algorithm. Rather than looking for a path from point A to point B, this A* variation looks horizontally for every row, and vertically for every column, on the game board. As it finds groups of adjacent Puyos of matching color, it puts them in a set, which is the Open list in the A* algorithm. Once it has four or more Puyos of the same color in the Open list, it adds these Puyos to another list, akin to the Moves list in A*.

Code Walkthrough

The game is implemented in Objective-C using a game library for iPhone called Cocos2D. This easy-to-learn library lets you concentrate on the logic behind your game and not on the technical details. But it also lets you get down and dirty when you need to.

The code is provided on the book's web site, so I will not go over it line by line. I will only cover the most important code. The game is about Puyos, so let's start with that class.

The Puyo class

The Puyo class is the in-code representation to the falling blocks on screen. Listing 2 shows the public interface to the Puyo class in the file Puyo.h.

Listing 2. *The Public Interface to the Puyo Class*

```
#import "cocos2d.h"

@interface Puyo : Sprite {
        int boardX, boardY;
        int puyoType;
        BOOL stuck;
        BOOL disappearing;
}

@property int boardX;
@property int boardY;
@property int puyoType;
@property BOOL stuck;
@property BOOL disappearing;

+ (Puyo *) newPuyo;
- (void) moveUp;
- (void) moveDown;
- (void) moveLeft;
- (void) moveRight;

@end

// Macros to define puyo position on the screen
// based on board coordinates
#define COMPUTE_X(x) (abs(x) * 32)
#define COMPUTE_Y(y) abs(480 - (abs(y) * 32))
#define COMPUTE_X_Y(x,y) ccp( COMPUTE_X(x), COMPUTE_Y(y) )
```

The Puyo class inherits from the Cocos2D class Sprite, which in the simplest terms means you can load a graphic and display it anywhere on the screen, apply transparency to the graphic, and other neat effects provided for us by Cocos2D. I have extended the Sprite class by adding the properties discussed in the previous "Basic Code Design" section: column and row position on the game board, a Puyo-type integer (which I use to determine if any other Puyo instance has the same color), as well as whether the Puyo is stuck and whether it's disappearing. I've also added a constructor method called newPuyo, and four instance methods for moving the Puyo around the screen and updating the board coordinates in one step.

At the end of Puyo.h, I have created a few C macros for turning board row and column coordinates into screen positions. These are used by the class implementation.

So now let's look at the implementation for the Puyo class, shown in Listing 3. This code is in the file Puyo.m.

Listing 3. *The Implementation of the Puyo Class*

```
#import "Puyo.h"

@interface Puyo (private)

- (void) initializeDefaultValues;
- (void) redrawPositionOnBoard;

@end

@implementation Puyo

@synthesize puyoType;
@synthesize stuck;
@synthesize boardX;
@synthesize boardY;
@synthesize disappearing;
```

At the top of the listing, I import the header file to define the interface. Immediately after the import of the Puyo interface definition, I define a category for the Puyo class, which I've called private. I put here any internal method definitions of class Puyo that I want to keep private. In C++, I would have used the private keyword, but Objective-C does not have private methods. Defining a category at the top of the class .m file is a common pattern to accomplish method hiding in Objective-C. Calling it private is just tradition passed down by experienced Objective-C developers.

The implementation of the Puyo class starts immediately after the private category definition ends. Here you synthesize all the properties defined in the public interface for the class. Synthesizing properties means the compiler generates magic code "behind the scenes" to set and get the values of those properties, and it also lets you use simplified syntax to define or retrieve the values of those properties. This simplified syntax makes for nicer coding when dealing with instances of the class. I will show you this in the game logic code, where it's all about dealing with Puyos.

Next up is the implementation of the newPuyo constructor method, shown in Listing 4.

Listing 4. *Continuing the Implementation Details of the Puyo Class with the newPuyo Constructor*

```
+ (Puyo *) newPuyo {
    NSString *filename = nil, *color = nil;
    Puyo *temp = nil;
    int puyoType = random() % 4;

    switch (puyoType) {
        case 0:
            color = @"blue";
            break;
        case 1:
            color = @"red";
            break;
        case 2:
            color = @"yellow";
            break;
        case 3:
            color = @"green";
            break;
        default:
            color = nil;
            break;
    }
    if (color) {
        filename =
            [[NSString alloc]
                initWithFormat:@"block_%@.pvr", color];
        temp = [self spriteWithFile:filename];
        [filename release];

        [temp initializeDefaultValues];
        [temp setPuyoType: puyoType];
    }
    return temp;
}
```

The code for creating the Puyo instance draws a random number between 0 and 3, and depending on the result, loads the corresponding graphic. The Puyo blocks are images 32 pixels wide by 32 pixels tall. There are images for green, yellow, blue, and red Puyos.

If you look at the line where I get the file name for the Puyo color, you may notice that I'm loading a strange file with an extension of .pvr. These are PowerVR (PVR) files, optimized explicitly for the GPU inside the iPhone and iPod Touch. These files are compressed and take up to four times less space in the iPhone's memory than graphics files of other types.

I use this whenever possible because the GPU shares up to 24 MB of RAM with the system in the iPhone platform. This means that your game can have up to 24 MB less memory available than if you did not use the GPU for graphics. (For example, plain old UIKit apps don't have this issue.) Using as little GPU RAM as possible for graphics by using PVR files means you have a bit more RAM available for other parts of your game. The only downside to PVR files is that they have to be square, and their size has to be a power of two. These Puyo graphics are 32 pixels by 32 pixels, so I made PVR files for this game.

Truth be told, this game only loads four small Puyo graphics and a 320 by 480 pixel background graphic at any one time. I was hardly pushing the memory limits of the GPU. But I wanted to show and explain PVR files to encourage developers to use them. There are several ways to create PVR files: You can use PowerVR's Windows-only GUI tools, or the converttool command-line utility provided by Apple with the iPhone SDK (discussed by Brian Greenstone in Chapter 4 of *iPhone Games Projects*).

Let's continue with the Puyo class implementation. Listing 5 shows the two private methods, initializeDefaultValues and redrawPositionOnBoard.

Listing 5. *The Implementation of the initializeDefaultValues and redrawPositionOnBoard Methods in the Puyo Class*

```
- (void) initializeDefaultValues {
    [self setTransformAnchor: ccp(0,0)];
    [self setPosition: ccp(0,0)];
    [self setOpacity: 255];
    [self setStuck: NO];
    [self setDisappearing: NO];
    [self setBoardX: 0];
    [self setBoardY: 0];
}

- (void) redrawPositionOnBoard {
    [self setPosition: COMPUTE_X_Y(boardX, boardY)];
}
```

The method initializeDefaultValues is used to set all the properties in a Puyo to nice defaults. The first one is the Sprite's transform anchor. In Cocos2D, the transform anchor is the set of coordinates that Cocos2D uses to rotate objects on screen. In Sprites, the default transform anchor is the middle of the image. I set it to the lower left of the image because this made it easier to calculate screen position based on game board coordinates.

The position property refers to on-screen position, not to board position. And opacity is the opposite of transparency. The opacity value of 255 means the Puyo is fully non-transparent, otherwise known as *opaque*. The other properties should be fairly obvious.

In the preceding code, you may have noticed there is some sort of function called ccp being used to set the transform anchor and position properties. The ccp is actually a convenience macro defined by Cocos2D, and it means "Cocos Point." What is happening is that the transform anchor and position properties take a C structure with one element for the X coordinate and another element for the Y coordinate. Using the ccp macro makes creating an instance of this structure simple and succinct. This is similar to the Core Graphics macro CGPointMake.

The method redrawPositionOnBoard sets the Puyo position on the screen to the coordinates corresponding to the board X and Y coordinates.

At last you come to the end of the Puyo class implementation. Listing 6 shows the implementation of the methods used to move the Puyos by board position.

Listing 6. *Implementation of Methods to Move the Puyos on the Game Board*

```
- (void) moveRight {
    boardX += 1;
    [self redrawPositionOnBoard];
}

- (void) moveLeft {
    boardX -= 1;
    [self redrawPositionOnBoard];
}

- (void) moveDown {
    boardY += 1;
    [self redrawPositionOnBoard];
}

- (void) moveUp {
    boardY -= 1;
    [self redrawPositionOnBoard];
}
```

These methods simply subtract or add one (1) to the corresponding board coordinate and redraw the Puyo at its new on-screen location. This is the end of the Puyo class implementation.

Next, let's examine the code that starts the game—the PuyoCloneAppDelegate class.

The PuyoCloneAppDelegate class

The PuyoCloneAppDelegate can implement several methods that get called by Cocoa Touch during the execution of your iPhone app. There are methods for letting you know that your code has finished launching, that memory is running low, and that the iPhone OS needs to interrupt your game because of an incoming phone call or SMS text message. Let's look at the implementation of some of these next, shown in Listing 7.

Listing 7. *Implementation Details of the PuyoCloneAppDelegate Class*

```
-(void) applicationWillResignActive:(UIApplication *)application
{
    [[Director sharedDirector] pause];

    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Resuming Game"
        message:@"Click OK to resume the game"
        delegate:self
        cancelButtonTitle:@"OK"
        otherButtonTitles: nil];
    [alert show];
    [alert release];
}

- (void)alertView:(UIAlertView *)alertView
        clickedButtonAtIndex:(NSInteger)buttonIndex
{
    [[Director sharedDirector] resume];
}
```

The applicationWillResignActive: method gets called when the user receives a phone call, SMS text message, or any other kind of notification while running the app. You use this method to pause your game and save the player's progress. In this simplified example, the game state is not saved, though. Instead, you ask the Cocos2D Director to pause the game and create an UIAlertView to let the user resume gameplay. The method alertView:clickedButtonAtIndex: is called when the user clicks the OK button on the alert. This method resumes the game.

The AppDelegate protocol defines a method applicationDidFinishLaunching:, which is called when the iPhone OS has finished loading your application. It is here that the game begins in earnest. Listing 8 shows my implementation of the applicationDidFinishLaunching: method.

Listing 8. *Implementation of the applicationDidFinishLaunching: Method in the PuyoCloneAppDelegate Class*

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    // Init the window
    window =
        [[UIWindow alloc] initWithFrame:
            [[UIScreen mainScreen] bounds]];
    [window setUserInteractionEnabled:YES];
    //[window setMultipleTouchEnabled:YES];

    // Init Cocos2D Director
    //[Director useFastDirector];
    //[[Director sharedDirector] setLandscape: YES];
    [[Director sharedDirector] setDisplayFPS:YES];

    // Attach the window to Cocos2D Director and make it visible.
    [[Director sharedDirector] attachInWindow:window];
    [window makeKeyAndVisible];

    // Seed random number generator.
    struct timeval tv;
    gettimeofday( &tv, 0 );
    srandom( tv.tv_usec + tv.tv_sec );

    // Start the game by running the GameScene.
    [[Director sharedDirector] runWithScene: [GameScene node]];
}
```

The method begins by creating the UIWindow instance that is going to display the game. The window is set to take the whole screen and to respond to user interaction, also known as *touch events*. The code to enable multi-touch is commented out in case you want to enable this while experimenting with the game code.

Next, I initialize the Cocos2D Director. The Cocos2D Director is a singleton class that displays your game screens. I have commented out the call to request use of the Cocos2D FastDirector because this game did not really need it. The FastDirector runs the Cocos2D display loop more aggressively and squeezes more performance out of the iPhone for your Cocos2D game.

Likewise, I have commented out the setting of landscape display mode because this is a portrait mode game. I have enabled the display of frames per second (FPS) in the lower-left corner of the screen. Comment this out or change the setting to NO to not display the FPS readout.

```
    // Attach Cocos2D Director to the window and make it visible.
    [[Director sharedDirector] attachInWindow:window];
    [window makeKeyAndVisible];
```

Once the Cocos2D Director is initialized, it attaches the Director to the UIWindow instance created at the beginning of the method. This tells Cocos2D Director to use this window for its display.

Then I make the window "key and visible." This is Cocoa Touch parlance for making the window accept input and show its content.

```
    // Seed random number generator.
    struct timeval tv;
    gettimeofday( &tv, 0 );
    srandom( tv.tv_usec + tv.tv_sec );
```

Immediately after making the window visible, I seed the standard C library random number generator function srandom, using a number derived from the current time by the standard C library function gettimeofday. This means the sequence of Puyos generated by the game should not repeat by any two runs of the game, unless you start the two runs at the exact same time of day, down to the same microsecond. That's good enough randomness for my needs.

```
    // Start the game by running the GameScene.
    [[Director sharedDirector] runWithScene: [GameScene node]];
```

And finally, I instantiate something called GameScene and ask the Cocos2D Director to run it. That's when the game actually starts. Let's now examine the GameScene class.

The GameScene class

Scenes are a Cocos2D abstraction used to contain the different elements you have on screen at any particular time. A Cocos2D game is made up of one or more Scene class instances. When you start your game, you set up Cocos2D and tell it to run your first scene. At some other point in time, your game logic determines that the current scene has ended, and it tells Cocos2D to run another scene.

But the name "scene" may be a little confusing. A Cocos2D scene does not imply you are constrained to making cinematic RPGs or graphic adventures. A scene can consist of something as simple as a background graphic and some falling Puyos. Think of a scene instance as just a container of all the stuff you're going to show on screen in a specific part of your game. As your game "moves" from one part to another (i.e., playing the main game logic, returning to the main menu, or displaying the high scores), you instantiate and tell Cocos2D to run different scenes.

In this game, there is only one scene, the GameScene class. I made my Puyo game really simple. When you run it, you go straight to the main game logic. There is no main menu, no high scores, no Help screen. It's meant as a code sample to learn about game development, not as a finished game product for sale; I'll leave that last detail to you as an exercise for the reader. So I concentrated on the game logic.

If you examine the GameScene.h file, you will see the GameScene class inherits from the Cocos2D Scene class and does not specify any new properties or methods. The specific differences between an instance of Cocos2D Scene and GameScene are in the implementation of the init and dealloc methods. Let's examine the implementation details of the GameScene class in Listing 9, which shows the file GameScene.m.

Listing 9. *Implementation Details of the GameScene Class*

```
#import "GameScene.h"
#import "GameLogicLayer.h"

@implementation GameScene

- (id) init {
    self = [super init];
    if (self != nil) {
        Sprite *bg = [Sprite spriteWithFile:@"background.png"];
        [bg setPosition:ccp(160, 240)];
        [self addChild:bg z:0];

        Layer *layer = [GameLogicLayer node];
        [self addChild:layer z:1];
    }
     return self;
}

- (void) dealloc {

    [self removeAllChildrenWithCleanup:YES];
    [super dealloc];
}

@end
```

The GameScene class imports the GameScene header file, and then it imports the GameLogicLayer header file. I'll discuss GameLogicLayer next, so I'm going to ignore it until I finish explaining GameScene.

In the init method, the GameScene creates a Sprite object, which loads the background image. Then it sets the Sprite's on-screen location to the middle of the screen. Finally, it adds the background sprite to the scene. You then proceed to instantiate the GameLogicLayer class and add it to the scene.

Note that the Scene class's addChild method has a parameter called z. This refers to the relative depth of the different elements added to the scene. The z parameter is optional, but when used, the elements with the highest z are displayed over elements with lower z. This is called *depth ordering*, and in this case, essentially means anything displayed by the GameLogicLayer class is displayed over the background image.

In the dealloc method, you remove all objects added to the scene, indicating you want Cocos2D to "clean up" by releasing the objects as well, and then you must call the dealloc method of the superclass (a.k.a. the Cocos2D Scene class from which GameScene is derived).

GameLogicLayer class

As the name implies, the GameLogicLayer class is where all the logic for the game is coded. The GameLogicLayer instance detects touch and interprets it, and then sets up a method to be called 60 times a second that runs the game logic. Everything that happens in the game is a direct result of these two actions. Let's examine the interface for GameLogicLayer in the file GameLogicLayer.h, shown in Listing 10.

Listing 10. *Interface of the GameLogicLayer Class*

```
#import "Puyo.h"

@interface GameLogicLayer : Layer {
    enum touchTypes {
        kNone,
        kDropPuyos,
        kPuyoFlip,
        kMoveLeft,
        kMoveRight
    } touchType;

#define kLastColumn 9
#define kLastRow 12

    // The board is 10 puyos wide x 13 rows tall.
    Puyo *board[kLastColumn + 1][kLastRow + 1];
    Puyo *puyo1, *puyo2;
    int frameCount;
    int moveCycleRatio;
```

The first thing the code does is import the Puyo class header, as you will be declaring a few variables of that type in this class. Then you create an enumeration type to define the different kinds of touch events the game logic knows about.

Immediately after the enumeration of touch types, you declare a C array of Puyos. The array is 13 rows of 10 Puyos. This array of Puyos is a proxy to what is displayed on screen. It is the model in your game's Model-View-Controller pattern.

The C preprocessor macros called kLastColumn and kLastRow are used throughout the game logic to make sure Puyo movement is constrained within the array of Puyos, and by proxy, to the screen.

After declaring your array of Puyos, you declare two Puyos called puyo1 and puyo2, which will hold the two Puyos as they fall from the top of the screen and progress down the board. I declared two variables because sometimes one Puyo gets stuck on a vertical group of Puyos while the other Puyo keeps falling. I need to account for both Puyos separately in the game logic.

I declare two variables called frameCount and moveCycleRatio that work together. The frameCount is incremented every frame. The moveCycleRatio is the number of frames needed for a block to drop to the row below. You'll see this later when we examine the code implementation. That's why those two variables exist, as shown in the following code.

```
    int difficulty;
    int score;
    Label *scoreLabel;
    Label *difficultyLabel;
```

The next two variables are the difficulty and the score. The score is incremented by 10 for every Puyo of the same color that the game logic finds adjacent to one another in groups of four or more. The difficulty increases every 500 points, increasing the speed at which Puyos drop by decreasing the moveCycleRatio. The code also declares two Cocos2D labels. A *label* is a graphic that displays text or numbers. In this case, they display the score and the difficulty. These labels are updated each time the score and difficulty are modified.

```
    enum puyoOrientations {
        kPuyo2RightOfPuyo1,
        kPuyo2BelowPuyo1,
        kPuyo2LeftOfPuyo1,
        kPuyo2AbovePuyo1
    } puyoOrientation;
```

This code declares another enumeration, this time to hold the various ways the two falling Puyos can be oriented when "flipped." The default orientation locates the second Puyo, puyo2, immediately to the right of the first Puyo, puyo1. The other orientations locate puyo2 below, to the left, and above puyo1, respectively. The Puyos flip whenever the user taps the Puyos as they fall, as long as there are no obstructions where puyo2 will be located after the flip. If there is an obstruction, the tap is ignored.

```
    NSMutableSet *currentGrouping;
    NSMutableSet *groupings;
}

- (void) updateBoard:(ccTime)dt;
@end
```

Finally, the code declares two NSMutableSets. These are collection classes provided by Cocoa. These are used by the code that detects groups of adjacent Puyos of the same color. In this implementation, these are roughly equivalent to the A* Open list and Moves list.

This section of the code also defines the only public method in the class, updateBoard, which will be set up to be called 60 times per second. Now that you've examined the interface to the GameLogicLayer class, let's examine the implementation, beginning with Listing 11.

Listing 11. *Implementation of the GameLogicLayer Class*

```
#import "GameLogicLayer.h"

// List private methods here.
@interface GameLogicLayer (private)

- (void) startGame;
- (void) clearBoard;
- (void) tryToCreateNewPuyos;
- (void) createNewPuyos;
- (void) gameOver;

- (void) processTaps;
- (void) movePuyosAndDisappearGroupings;
- (void) findPuyoGroupings;
- (void) computeScore;
- (void) determineDifficultyIncrease;
- (void) updateInfoDisplays;

- (void) detectStrayGrouping;
- (void) checkForPuyoGroupings:(Puyo *)p1;

- (void) movePuyoLeft:(Puyo *)puyo;
- (void) movePuyoRight:(Puyo *)puyo;
- (void) movePuyoDown:(Puyo *)puyo;
- (void) movePuyoUp:(Puyo *)puyo;

@end
```

As with other classes covered so far, I import the class header file and declare some private methods. In this case, there are a lot of private methods. Rather than cover them all, I'm going to go over the main pieces of game logic: processing taps on the screen, moving Puyos on the board, and finding groups of adjacent Puyos of the same color.

Processing taps on the screen in Cocos2D is done by declaring a method called ccTouchesEnded: withEvent:. This method is called when the finger is lifted from the screen, so it's perfect for catching taps. The game doesn't support dragging, simultaneous taps, or drags with multiple fingers. Listing 12 shows the implementation of ccTouchesEnded: withEvent:.

Listing 12. *Implementation of the ccTouchesEnded: withEvent: Method*

```
- (BOOL)ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint point = [touch locationInView: [touch view]];
    point.y = 480 - point.y;

    if ( !puyo1.stuck && !puyo2.stuck &&
        ( abs((int)puyo1.position.x - (int)point.x) < 32 &&
        abs((int)puyo1.position.y - (int)point.y) < 32 ) ||
        ( abs((int)puyo2.position.x - (int)point.x) < 32 &&
        abs((int)puyo2.position.y - (int)point.y) < 32 ) )
    {

        touchType = kPuyoFlip;

    }

    else if ((int)point.y < 68) touchType = kDropPuyos;

    else if ((int)point.x < 116) touchType = kMoveLeft;

    else if ((int)point.x > 204) touchType = kMoveRight;

    return kEventHandled;
}
```

The code takes the touch and converts it to a point on the screen. In Cocos2D, the Y coordinate runs from the bottom of the screen to the top. But the touches are provided by UIKit, which runs the Y coordinate from top to bottom. You need to change the touch Y coordinate to match the way your screen coordinates run.

The code then checks if the screen tap happened within 32 pixels of either puyo1 or puyo2. If that is the case, it sets the variable touchType to kPuyoFlip.

The code then proceeds to check whether the user touched the region in the last 68 pixels at the bottom of the screen. The code sets touchType to kDropPuyos if this is the case. The rest of the code checks whether the user touched the leftmost or rightmost region of the screen. These regions are 116 pixels wide and start at the left and right borders of the screen. The code sets the touchType variable according to which region was touched. So the touchType variable is set to different values depending on where the user has touched the screen. There's a bottom region, left and right regions, and a special region within 32 pixels of either Puyo. The method processTaps is called by the updateBoard method 60 times per second to process the touchType variable. The implementation of the updateBoard method is shown in Listing 13.

Listing 13. *Implementation of the updateBoard: Method*

```
- (void) updateBoard:(ccTime)dt {
    frameCount++;
    [self processTaps];
    [self disappearPuyos];
    if (frameCount % moveCycleRatio == 0) {
        [self movePuyosDown];
        [self findPuyoGroupings];
        [self computeScore];
        [self determineDifficultyIncrease];
        [self updateInfoDisplays];
    }
}
```

The updateBoard: method is executed 60 times per second and calls the different methods that make up the logic of the game. I coded it this way to keep the different parts of the game logic small and manageable. One big method for all the game logic would have been harder to debug and harder to move around.

The first method the updateBoard: method calls is the processTaps method, which processes the touchType variable. Listing 14 shows the first portion of the processTaps method. I'll break this into several portions.

Listing 14. *Implementation of the processTaps Method*

```
- (void) processTaps {
    if (touchType == kPuyoFlip) {
        // Reset touch type
        touchType = kNone;

        if (!puyo1.stuck && !puyo2.stuck &&
            abs(puyo1.boardX - puyo2.boardX) <= 1 &&
            abs(puyo1.boardY - puyo2.boardY) <= 1)
        {

            switch (puyoOrientation) {
                case kPuyo2RightOfPuyo1:
                    if ( (puyo1.boardY + 1) <= kLastRow &&
                        nil == board[puyo1.boardX][puyo1.boardY + 1])
                    {
                        [self movePuyoDown:puyo2];
                        [self movePuyoLeft:puyo2];
                        puyoOrientation = kPuyo2BelowPuyo1;
                    }
                    break;
                case kPuyo2BelowPuyo1:
                    if ( (puyo1.boardX - 1) >= 0 &&
                        nil == board[puyo1.boardX - 1][puyo1.boardY])
                    {
                        [self movePuyoLeft:puyo2];
                        [self movePuyoUp:puyo2];
                        puyoOrientation = kPuyo2LeftOfPuyo1;
                    }
                    break;
                case kPuyo2LeftOfPuyo1:
                    if ( (puyo1.boardY - 1) >= 0 &&
                        nil == board[puyo1.boardX][puyo1.boardY - 1])
                    {
                        [self movePuyoUp:puyo2];
                        [self movePuyoRight:puyo2];
                        puyoOrientation = kPuyo2AbovePuyo1;
                    }
                    break;
                case kPuyo2AbovePuyo1:
                    if ( (puyo1.boardX + 1) <= kLastColumn &&
                        nil == board[puyo1.boardX + 1][puyo1.boardY])
                    {
                        [self movePuyoRight:puyo2];
                        [self movePuyoDown:puyo2];
                        puyoOrientation = kPuyo2RightOfPuyo1;
                    }
                    break;
            }

        }
```

The processTaps method works by examining the touchType variable that was set earlier by the ccTouchesEnded:withEvent: method. The first portion of processTaps deals with flipping the Puyos. The code resets the touchType variable immediately so that you don't forget and are ready for the next touch event. Then you make sure the two falling Puyos are next to each other and are not “stuck.” The code also checks the location where puyo2 would be moved to after the flip. If this location was not vacant, if either puyo1 or puyo2 was stuck, or if the Puyos were not next to one another, the code would not execute the flip. The code then proceeds to move puyo2 to its new location based on the current value of the puyoOrientation variable, then it updates this variable to match the new orientation.

Figures 5 through 9 show some screenshots of the game as a pair of Puyos is flipped four times in the course of a few seconds.

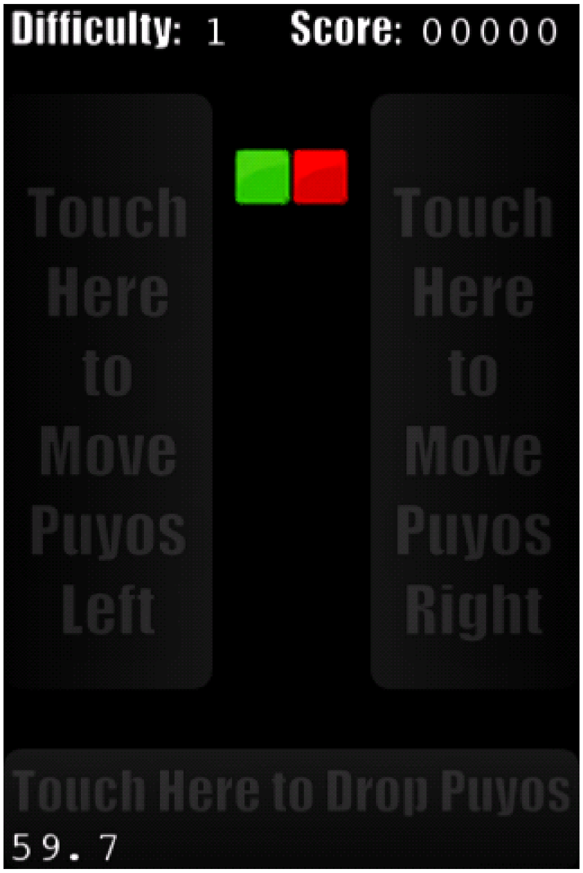


Figure 5. *A pair of Puyos in the default orientation*

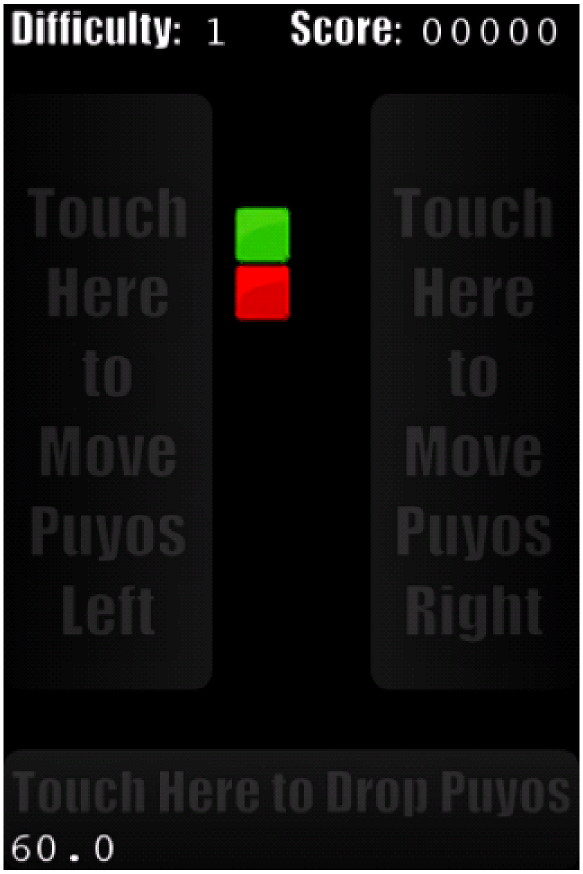


Figure 6. *The same pair of Puyos flipped once*

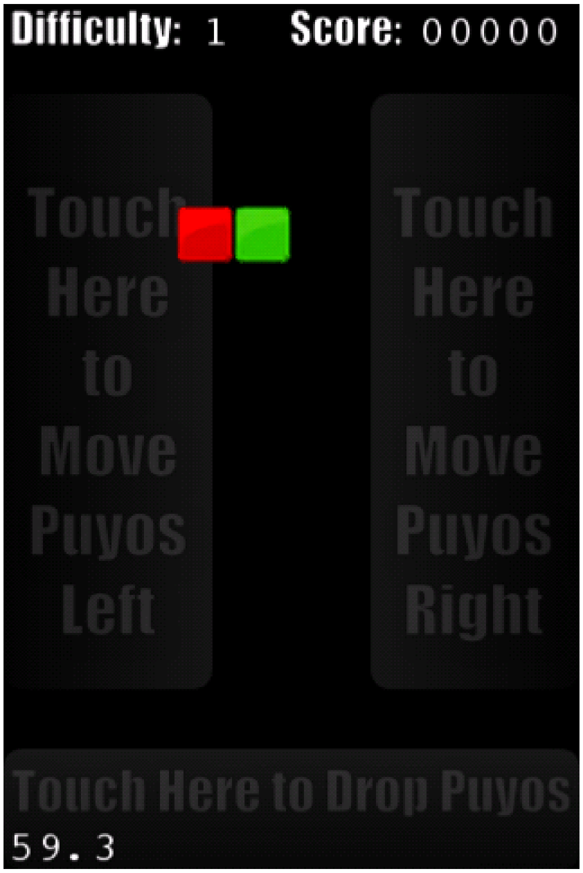


Figure 7. *The same pair of Puyos flipped a second time*

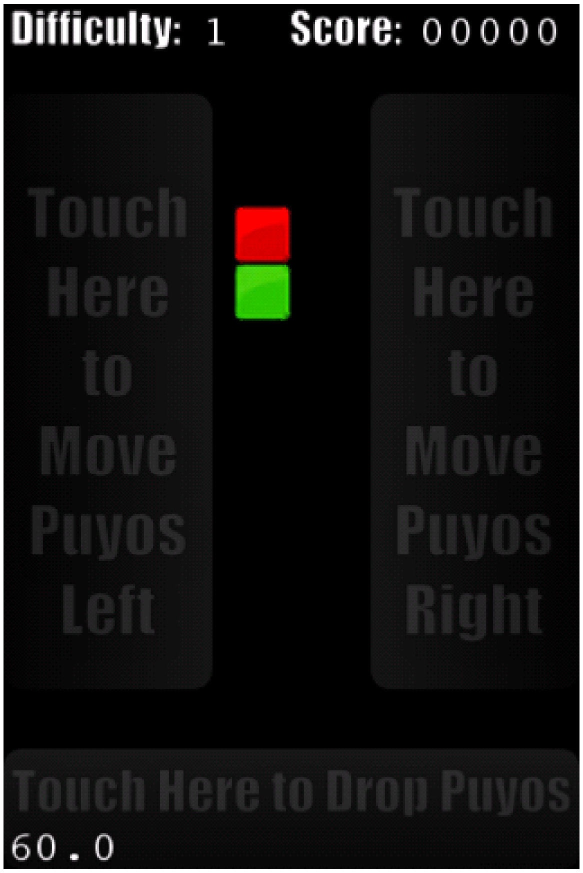


Figure 8. *The same pair of Puyos flipped for the third time*

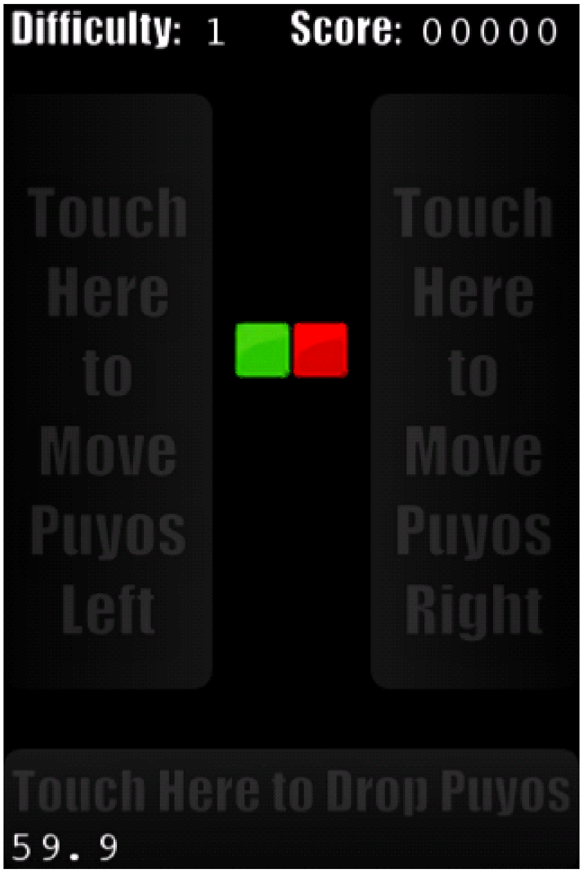


Figure 9. *The same pair of Puyos flipped back to the original orientation*

Let's now examine the code for moving the Puyos to the left, shown in Listing 15.

Listing 15. *Continuing Implementation of the processTaps Method*

```
} else if (touchType == kMoveLeft) {
    // Reset touch type
    touchType = kNone;

    // Determine the leftmost puyo so that it will be moved first.
    Puyo *p1, *p2;
    if (puyo1.boardX < puyo2.boardX) {
        p1 = puyo1; p2 = puyo2;
    } else {
        p1 = puyo2; p2 = puyo1;
    }
```

```
        if (p1.boardX > 0 && !p1.stuck) {
            if (nil == board[p1.boardX - 1][p1.boardY]) {
                [self movePuyoLeft: p1];
            }
        }

        if (p2.boardX > 0 && !p2.stuck) {
            if (nil == board[p2.boardX - 1][p2.boardY]) {
                [self movePuyoLeft: p2];
            }
        }
```

Moving the Puyos left, right, or down is kind of tricky. In this case, moving left, you first need to determine which of the two Puyos is leftmost on the board. Then check if there is any obstruction to the left of this Puyo. You also need to account for the left border of the map and whether the Puyo is already stuck. You do this for both Puyos (see Listing 16).

Listing 16. *Continuing Implementation of the processTaps Method*

```
    } else if (touchType == kMoveRight) {
        // Reset touch type
        touchType = kNone;

        // Determine the rightmost puyo so that it will be moved first.
        Puyo *p1, *p2;
        if (puyo1.boardX > puyo2.boardX) {
            p1 = puyo1; p2 = puyo2;
        } else {
            p1 = puyo2; p2 = puyo1;
        }

        if (p1.boardX < kLastColumn && !p1.stuck) {
            if (nil == board[p1.boardX + 1][p1.boardY]) {
                [self movePuyoRight: p1];
            }
        }

        if (p2.boardX < kLastColumn && !p2.stuck) {
            if (nil == board[p2.boardX + 1][p2.boardY]) {
                [self movePuyoRight: p2];
            }
        }
```

Moving right is accomplished in a similar manner: determine which is the rightmost Puyo, take into account the right border, whether there is an obstruction immediately right of the Puyo, and whether the Puyo is stuck and can't be moved. Again, you do this for both Puyos (see Listing 17).

Listing 17. *Continuing Implementation of the processTaps Method*

```
    } else if (touchType == kDropPuyos) {
        // Reset touch type
        touchType = kNone;

        // Determine the bottommost puyo so that it will be moved first.
        Puyo *p1, *p2;
        if (puyo1.boardY > puyo2.boardY) {
            p1 = puyo1; p2 = puyo2;
        } else {
            p1 = puyo2; p2 = puyo1;
        }

        if (!p1.stuck) {
            while (p1.boardY != kLastRow &&
                nil == board[p1.boardX][p1.boardY + 1])
            {
                [self movePuyoDown: p1];
            }
        }

        if (!p2.stuck) {
            while (p2.boardY != kLastRow &&
                nil == board[p2.boardX][p2.boardY + 1])
            {
                [self movePuyoDown: p2];
            }
        }

    } // End of if (touchType . . .
}
```

Finally, you get to the end of processTaps. In this portion, you get into the process of dropping Puyos as far down the board as they can in one step. You determine which Puyo is the bottommost of the two. Then if the Puyo is not stuck, you move it down until it reaches an obstruction or the bottom of the board. You do this for both Puyos.

In Figures 10 through 13, you see a pair of Puyos falling. As the images continue, one of the Puyos gets stuck, while the other continues to respond to taps. Because the other Puyo is stuck, flip taps are ignored.

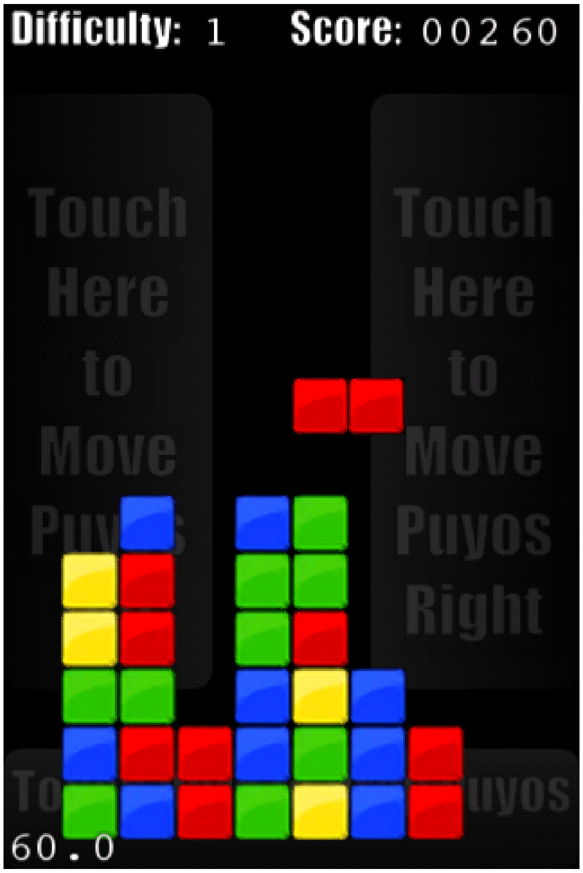


Figure 10. *A pair of Puyos falling*

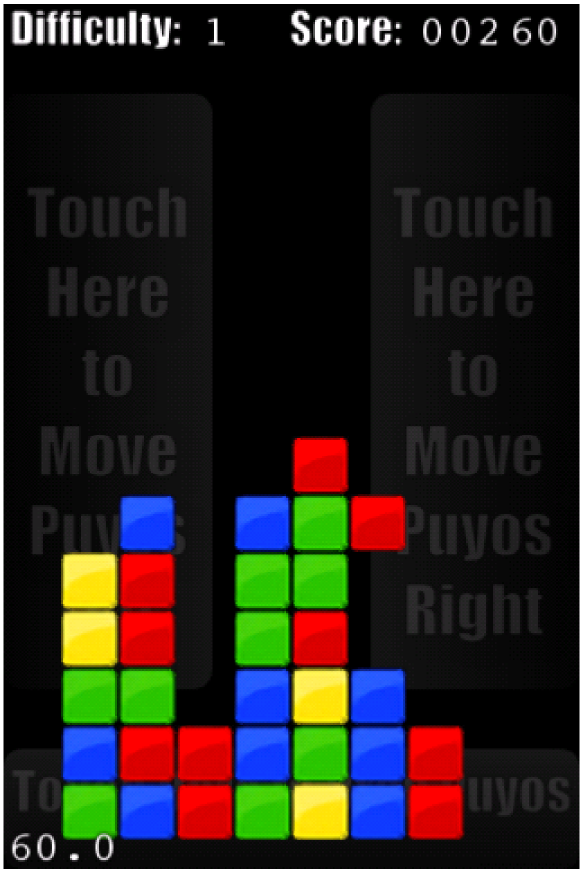


Figure 11. *One of the Puyos has gotten stuck, but the other Puyo continues falling.*

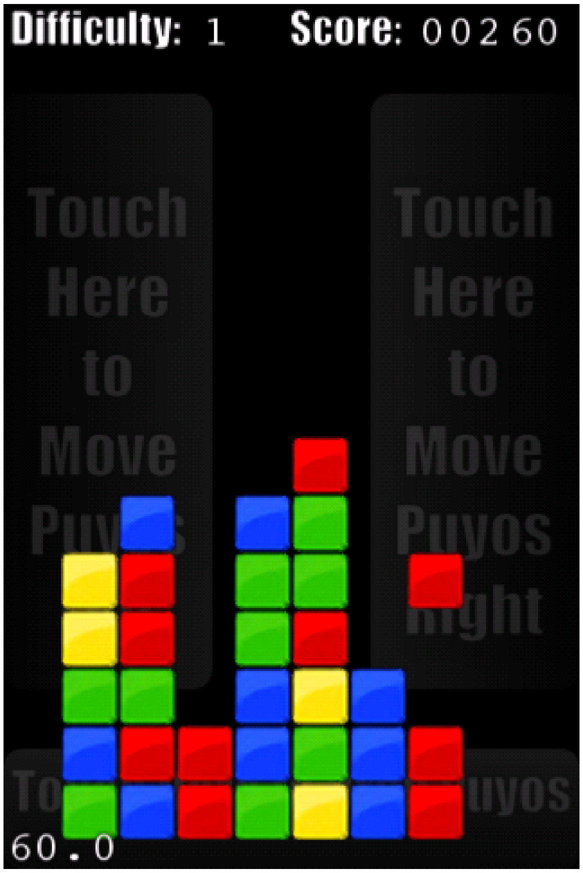


Figure 12. *The free Puyo can still be moved into position.*

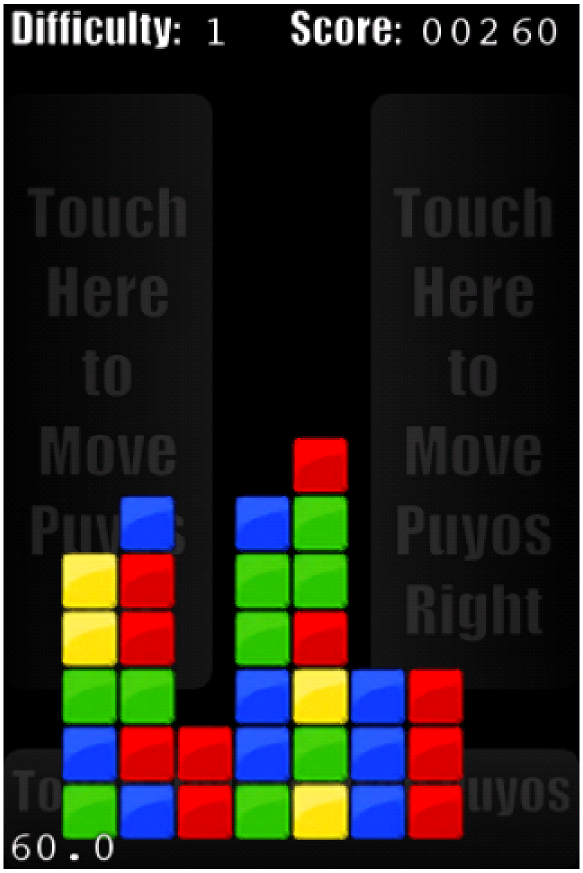


Figure 13. *The free Puyo now is "stuck" on other Puyos.*

If you recall the updateBoard: method, after calling processTaps, it calls the disappearPuyos method . This method takes any Puyos marked as disappearing, and well, makes them disappear by changing their opacity. Listing 18 shows this method.

Listing 18. *Implementation of the disappearPuyos Method*

```
- (void) disappearPuyos {
    Puyo *puyo = nil;

    for (int x = 0; x <= kLastColumn; x++) {
        for (int y = 0; y <= kLastRow; y++) {

            puyo = board[x][y];

            if (nil != puyo && puyo.disappearing) {
                if (5 < puyo.opacity) {

                    puyo.opacity -= 5;

                } else {

                    [self removeChild:puyo cleanup:YES];
                    puyo = nil;
                    board[x][y] = nil;

                } // End of if (5 < puyo.opacity)

            } // End of if (nil != puyo && puyo.disappearing)

        } // End of for y loop.
    } // End of for x loop.

    if ( puyo1.stuck && puyo2.stuck ) {
        [self tryToCreateNewPuyos];
    }

}
```

The disappearPuyos method sweeps through the board in a loop, moving from left to right and top to bottom. It checks whether the Puyos on the board are marked as disappearing by other parts of the game logic, and it decreases their opacity (makes them more transparent). If it finds a Puyo whose opacity is below 5, it removes it from the board and the screen.

The disappearPuyos method also checks if puyo1 and puyo2 are stuck. This condition signals that the code needs to create new Puyos (see Listing 19). Check the implementation of the method tryToCreateNewPuyos in the code download for the details.

Listing 19. *Implementation of the movePuyosDown Method*

```
- (void) movePuyosDown {
    Puyo *puyo = nil;

    for (int x = kLastColumn; x >= 0; x--) {
        for (int y = kLastRow; y >= 0; y--) {
            puyo = board[x][y];

            // Is puyo "solid" (i.e., not disappearing)?
            if (nil != puyo && !puyo.disappearing) {

                // Can this puyo drop down to the next cell?
                if ( kLastRow != y && (nil == board[x][y + 1]) ) {

                    // Channel Bob Barker: Come on down!
                    [self movePuyoDown:puyo];
                    puyo.stuck = NO;
                } else {
                    // This puyo can't drop anymore; it's stuck.
                    puyo.stuck = YES;
                }

            } // End of if (nil != puyo && !puyo.disappearing)

        } // End of for y loop.
    } // End of for x loop.

    if (kLastRow == puyo1.boardY) {
        puyo1.stuck = YES;
    }
    if (kLastRow == puyo2.boardY) {
        puyo2.stuck = YES;
    }
}
```

Unlike the disappearPuyos method, the movePuyosDown method sweeps through the board in a loop moving up from the bottom. Here's what it does:

- It checks if the Puyos are not disappearing and whether there is an obstruction immediately below the Puyo.
- If the Puyo is not disappearing and there isn't an obstruction below, it moves the Puyo down one block.
- If there is an obstruction, it marks the Puyo as stuck.
- The method also checks if either of puyo1 or puyo2 are located in the last row of the game board and marks them as stuck if that's the case.

The last code I'd like to discuss in the GameLogicLayer class is the logic that finds groups of adjacent Puyos of the same color. This logic is carried out by several methods, but it is launched by the updateBoard: method. That method calls the findPuyoGroupings method, which sweeps the board horizontally and then vertically. Detecting and collecting the groups is the same whether you are sweeping through the board from left to right or up and down, so the detection and collection logic is in the checkForPuyoGroupings: method, shown in Listing 20.

Listing 20. *Implementation of the checkForPuyoGroupings: Method*

```
- (void) checkForPuyoGroupings:(Puyo *)p1 {

    if (nil != p1 && p1.stuck && !p1.disappearing) {

        if ([currentGrouping count] > 0) {

            Puyo *p2 = [currentGrouping anyObject];
            if ( p2.puyoType == p1.puyoType ) {

                [currentGrouping addObject:p1];
            } else {

                if ([currentGrouping count] > 3) {

                    [groupings addObject:currentGrouping];

                }

                [currentGrouping release];
                currentGrouping = nil;

                currentGrouping = [[NSMutableSet alloc] init];
                [currentGrouping addObject:p1];

            } // End of if ( p2.puyoType == p1.puyoType )

        } else {

            [currentGrouping addObject:p1];

        } // End of if ([currentGrouping count] > 0)

    } else {

        if ([currentGrouping count] > 3) {

            [groupings addObject:currentGrouping];

        }

        [currentGrouping release];
        currentGrouping = nil;

        currentGrouping = [[NSMutableSet alloc] init];

    } // End of if (nil != p1)
}
```

The checkForPuyoGroupings: method is a complex one with intricate rules. Here is a breakdown of what the method does:

- First, it checks if the Puyo it is currently looking at is stuck and not disappearing. (Disappearing Puyos are already part of a group that has been counted in previous steps.)
- If the Puyo is stuck and not disappearing, the code checks whether Puyos of the same color have already been seen in previous runs through this code. (Remember, this code runs in a loop within the findPuyoGroupings method.)
- If the Puyo is the first Puyo seen recently, the Puyo is automatically added to the current set.
- If there are more Puyos of the same color, the current Puyo is added to the set.

- If not of the same color, the set is checked for whether it has at least four Puyos of the same color.
- If this is the case, the set is put aside for scoring in the set of all groups.
- Ultimately, the set is released and the current Puyo is added to a new set.
- If the Puyo is not stuck, is disappearing, or if the current board block being examined is empty, the code checks the current set for whether it has at least four Puyos of the same color.
- If this is the case, the set is put aside for scoring in the set of all groups.
- In the end, the set is released and a new empty set is allocated.

And this is the end of my breakdown of the code. Be sure to download the complete source to see it in its entirety. There are many more comments in the code download than you see here. I removed comments from the code listings because they gave away the code's secrets. With comments included, there is nothing left for me to explain!

Exercises for the Reader

As the Puyo game stands right now, it could use some improvements. Here is a list of ideas you could try to implement:

- Add a main menu: Create another Scene class, and use Cocos2D Menu and MenuItem instances to put a menu up when the game is started. Make one of the MenuItems start the GameScene when tapped.
- Add some sound or background music: Presently, the game is eerily quiet.
- Increase difficulty in a more exciting way: the way it is now, the code increases difficulty every 500 points. How about increasing difficulty each time the player clears the board completely?
- Add bonus points for when the player makes two groupings at the same time. Add sound effects for these special bonuses.

Summary

You have covered a lot of ground in this chapter, but the key points to remember are these:

- Consider the needs of your game to determine the technologies that you will need to use to develop your game. The iPhone platform gives you a lot of technologies to choose from, and some fit some game designs better than others. Casual games can often be implemented with simpler technologies than those used in larger console titles.

- Always be aware of how you are managing your memory. The iPhone has limited RAM, and if you're not careful, the OS will kick your game out of memory to make room for the work of background processes. This could happen at any time, ruining your game.
- A* is a very versatile algorithm. It is typically used for game AI, for guiding game units as they move around a map filled with obstacles. Many games, from driving simulations to real-time strategy war games, rely heavily on A*. It's a good algorithm to learn if you are serious about game development.
- Puyo seems like a simple game, but there's a lot to keep track of at the same time—the game logic can get complicated fast. The best way to become good at making games is by jumping right in and getting your hands dirty! Make changes to the game code provided by adding features or changing the game logic to put your own innovative stamp on falling block games.

The iPhone platform is very exciting. There are lots of technologies that you will need to learn in order to make games. I hope this chapter has provided you with some fat to chew on as you begin your explorations. Good luck, and see you on the App Store!