# Multiplayer Tetris

Mobile Application for the
Android Operating System

Mark Ha ‘13
01/10/12
JP Adviser: Robert Dondero

## Introduction

When I started this project, my final objective was to have an implementation of a fully functional, multiplayer version of the popular classic puzzle game Tetris for the Android. Traditional Tetris is single-player, with just one person racing against time for the best score. In multiplayer Tetris, however, players compete against each other, and what happens in one player's game affects the other player's game. For this project, that would mean two or more phones connecting to each other, each playing their own game, playing against each other in real time, while being shown thumbnails or some variation of their opponents' games as well. The full schedule had also included the development of a database section, which would save and maintain users' information such as win-loss record and other statistics, but that goal had the least priority.

Unfortunately, and to some extent as expected, the end result was far from my original goal. Entering this project, I had known and somewhat anticipated to work slower than what was necessary to have a finished product by the end of the semester. I came in knowing nothing about developing an application for a smartphone, as well as knowing nothing about networking, and so, even though the coding language is Java which I am familiar with, I realized that I would have to invest time into learning and familiarizing myself with the Android platform, and the unique features of Android applications. And I realized that even after I had invested that time into learning the "Android" way of programming, I would run into problems that I had never encountered before, and so, the programming process would be delayed significantly by my inexperience and unfamiliarity with the Android platform. On top of that, I myself did not own a smartphone at the beginning of the semester, and because of that, I was unfamiliar with what a typical application might look like, or what controls it might take advantage of. And to add to that, I had never made a multiplayer anything, and had never studied networking, or databases for that matter, and so, I would have to learn how to connect two of anything together first, let alone two Android smartphones. These are the limitations and challenges that I knew I would have to overcome to complete this project successfully.

As time grew shorter and shorter, I realized that I wouldn't be able to do everything (far from it, in fact), and I decided to settle with simply having two phones connect and then play each other in Tetris – a stepping stone to what I had hoped the final result to be. And so, in the end, the final product was an application that connects two Android phones together over a wireless Internet connection, starts the games upon either player's cue, and then shows each player both players' games on each own phone as they play each other – so as one person is playing their own game, they can constantly see what the other player is doing as well. And thus, my application is the basic, barebones version of a "Multiplayer Tetris" game.

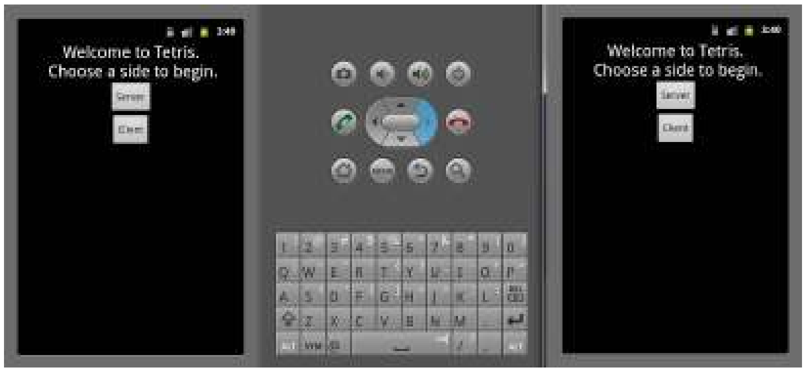
## Related Works

The game of Tetris has been around for almost three decades now, and has appeared on almost every game console and platform that has come out. But in more relevant terms, there exist quite a few versions of web-based Tetris online, on the Internet, some single player, some multiplayer. These websites, a popular one being www.tetrisfriends.com, are the model that I looked after as I began to develop my project. Some modes of Tetris enforce a time limit, where the player tries to get the most points as possible within that set amount of time. The classic mode simply runs endlessly, speeding up every so many points, until the player fills the screen and loses. The minimalistic multiplayer versions are for 2 players only, but include special interactions between the two players, such as the concept of "sending a line," where getting multiple lines at once or in a row create "garbage lines" on the other player's screen, which speeds up their path to defeat. But some multiplayer versions offer up to 4-way or 8-way Tetris, where 8 people can play together in real time, and are able to see everyone else's games as well, and in some cases choose who they want to "attack" with their "garbage lines." In short, Tetris on the Internet in particular is quite well-developed, with a variety of modes to choose from when playing Tetris.

The same is not true for the mobile phone platforms, however. Currently, searching "Tetris" on either the Android or iPhone application markets only yields one very well-made application for each platform. These applications, however, do not offer a multiplayer version of the Tetris game, instead choosing to offer a multitude of well-designed single-player variations and modes. And so, aside from connecting to the Internet browser and playing Tetris that way, there is no way to play Tetris with another friend, phone-to-phone; there is no way to compete against another person directly on the phone – instead, one would have to resort to playing two games side-by-side (physically), and comparing results. But even then, there is no true interaction between the two games – each game is still independent and on its own, as opposed to the more interactive multiplayer versions that one might see on the web. And for this reason, designing a new application that offered that multiplayer Tetris on a mobile phone seemed like something that would be both useful and interesting to do.

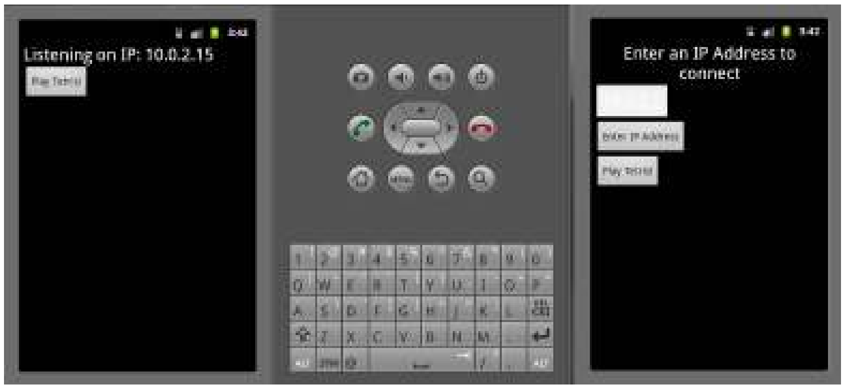
## Functionality

My application, in its current state, is really quite basic in terms of its functionality: put shortly, it is designed to link together two phones, and have them play Tetris against each other. More specifically, when a user opens up the application there are two choices to choose from – being the one "hosting" a game, or being the one "searching" for a game.



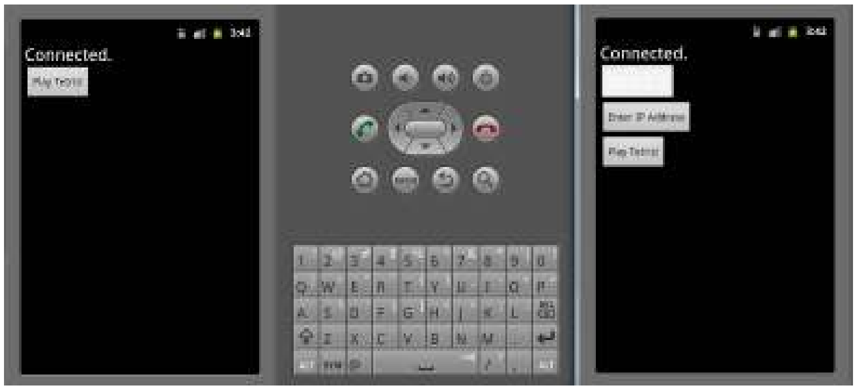
For the phone that chooses to host the game, the application retrieves and displays that phone's I.P. address. In my current implementation, the phones establish a connection through the use of I.P. addresses, and so for convenience and ease, the application displays the phone's the I.P. address. From then on, the host waits for another phone to attempt to connect to it. After the connection is established, the application waits until both players have hit the "Play Tetris" button (signifying that they are ready), and then the game starts.

For the phone that chooses to search for a game, the application asks the user to input an I.P. address. This user would thus need to know the other phone's I.P. address, meaning that external communication is required between the two phone users at the moment. After this user enters in an I.P. address and connects to the other phone, it waits until both users to have hit the "Play Tetris" button and then initiates the game on both phones.



Listening on IP: 10.0.2.15

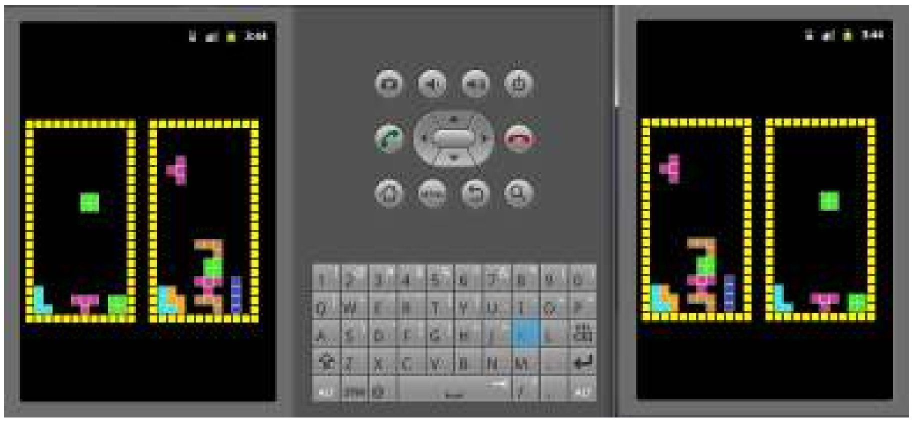
Enter an IP Address to connect

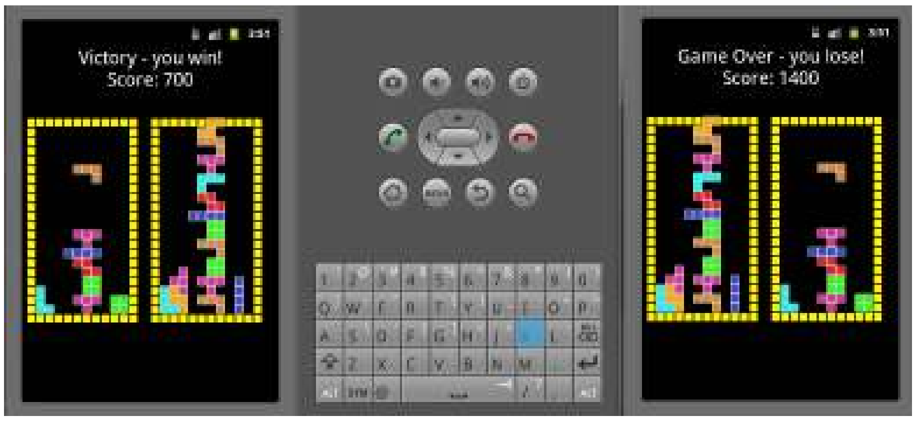


Connected.

Connected.

From there, everything is the same for both phones – the game begins, and each player's move is reflected on the other player's phone as well. And so, until one player loses (and the other players wins), both players are able to see the progress of both their own and their opponent's games. When the game ends, it either displays a victory or defeat message with that player's final score. Any key input that follows the final score screen exits and closes the application.



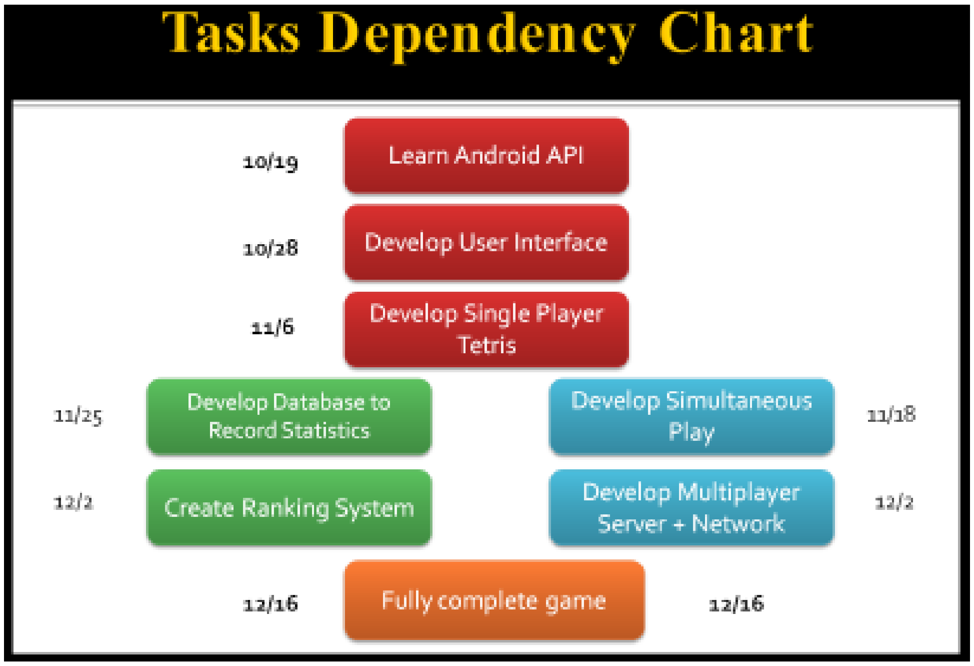


Regarding the game itself, it responds to five keyboard inputs: up, down, left, right, and space (or pressing the middle button on certain phones). Because a Tetris block never moves up in the game, the up key input instead rotates the block clockwise. Naturally, the down, left, and right keys inputs all move the block in each respective direction. Space bar causes the block to “fast drop,” or in other words, to go down as far as it can go instantly. Other than that, there are no inputs that affect the application.

## **Design Decisions**

The majority of this project, or rather, the most significant part of this project, was a result of the design decisions I made along the way. In fact, from the outset, before I had even written a line of code, I had a major decision to make – iPhone or Android? From an entrepreneurial perspective, iPhones are much more popular than Androids, and there are significantly more iPhone applications than Android applications, so if my intention was to make a sellable application, iPhone would probably be the right choice. But beyond that, choosing either or would mean a completely different set of user interface options – for example, iPhones are fairly standardized because there are only a few models, whereas there are more than plenty Android phone models, not to mention fifteen different levels of the Android SDK that vary from phone to phone. Thus, for something like designing the look of the game, it would have to generic enough to apply to several screen types and sizes, or else it would show up strangely on some phones. Probably the largest factor in my decision between iPhone and Android, however, was the fact that Android applications are programmed in Java, whereas iPhone applications are programmed in Objective-C. Because I was already exploring enough new territory, I figured sticking to the language I was more familiar with and had more experience would be the better idea, and so I ended up choosing to make an Android application.

Another major choice that I had to make early on was how I was going to budget my time, and what I was going to focus on. As I mentioned earlier, my "ideal" completed product would have everything from networking between multiple phones to a database or some sort to manage and display player statistics over time. There was also the matter of how in-depth I would make the Tetris game. The minimum approach would be to have the classic version of Tetris, where a block is falling and the player arranges it. A more zealous approach would include different modes and special features, such as “holding” a piece for later usage, being able to see what block is coming next, perhaps implementing a menu that shows each players current status (such as score, the number of lines cleared, etc.). There were several different paths I could go down, and I did not have time to go down everything path – so what was most important? What did I absolutely need to have by the end of the semester?



This is an image of the plan that I had made in the beginning of the semester. There were two main paths I could take – I could either focus on the multiplayer aspect of the application, or I could focus on the database and ranking statistics side of the application. While it would have been nice to have been able to do both, I was forced to stick to and devote all my time to just the multiplayer aspect of the game.

As I began looking up tutorials for programming in Android, I came across some sample source code for an implementation of the game Snake[1]. Because of its slight similarity in the way the game is displayed and the mechanics of the game, and because the source code was available to me, I decided to use the code for a Snake game as the starting template for my Tetris game. How exactly are they similar? In the game Snake, the player controls a segment of pieces that represent a snake that is constantly growing as it eats more and more "apple" pieces. Every tick of the clock, or every time cycle, depending on the game's speed mode, the snake moves forward (in the direction it is "facing") one square, or one tile. Every time it "eats" an apple, the length of the segment increases by one, and a new apple is put somewhere else in the 2-dimensional game grid. Similarly, for every tick of the clock, for every time cycle, the blocks in Tetris fall one space, toward the bottom. Like Snake, the user has the option of moving the block left, right, or down (but not up). And like Snake, there are "walls" that the block cannot pass, where hitting the bottom means that it is time for a new block. Of course, the games are far from the same, but I felt that they were similar enough to use the Snake code as a basis for my Tetris code – and in the end, it worked out pretty well; though I made a few changes to fit it to the Tetris game, I still use a slightly altered version of the *TileView.java* that came from the Snake game. And what that code does is it turns the phone's screen into a grid – for the purposes of my Tetris game, it is a 12x22 grid because a typical Tetris game has dimensions of10x20, and then there are walls on all sides. By using a grid, the concept of a block "moving" one in any direction is facilitated greatly, not to mention it's almost a natural decision given the grid-like nature of Tetris blocks. And so ultimately, I was able to use the tile grid concept (and some of the code) from the Snake game as a basis for my Tetris game.

[1] "Snake – Snake", http://developer.android.com/resources/samples/Snake/index.html (Jan 2012).

The only thing that I "stole" from the Snake source code was the *RefreshHandler()* function in my *TetrisView.java* class. I also used it as a basis when writing the *RefreshHandler()* function in my *Start.java* class. What is does is very simple – instead of creating a countdown timer or a clock system like other Java programs might do, Android applications usually create the same effect by using what are called handlers (they handle special cases) – in this case, *RefreshHandler()* is a handler that runs every X seconds, depending on how often I want it to run. In *TetrisView.java*, the handler clears and redraws the game (animates it) with the updated positions of any blocks that might have moved since the last time it redrew the game. It also checks to see if a clock cycle has passed, to see if the Tetris piece sinks again. This is the same functionality as that of the Snake segment that moves forward every time cycle.

As I was coding the standalone game (a regular Tetris game with no networking, no multiplayer), I ran into a lot of edge cases, particularly because of how I chose to represent the Tetris piece that the player controls. One idea that I considered was creating an Object class that took up a 4x4 area (Every Tetris piece is made up of 4 squares, so the maximum length piece is a 4x1 or a 1x4 piece), and to consider the Tetris piece as a whole, rather than as 4 individual squares. Originally, I had a separate subclass *Coordinate* to represent a Tetris piece, and everything dealing with the piece was done in the *TetrisView.java* class. However, things got too messy and unorganized, and so I created the *TetrisBlock.java* Object class, which simply represents a Tetris piece. But rather than go with the 4x4 area approach, I decided to make each Tetris piece one square essentially. Each TetrisBlock object was one square, and then, depending on the orientation (which is changed whenever the user rotates the piece) and the type of the block (there are 7 different types of pieces in the classic Tetris game), the TetrisBlock object determines where the other three squares go. And so, each TetrisBlock had a "centerpiece" from which everything else was determined. I prefer this approach because when making changes to Tetris piece, for instance, moving it left or right, I only have to move the centerpiece, and then recalculate the positions of everything else. It also means that instead of storing eight integers representing the horizontal and vertical position of each square of the piece, the program only needs to keep track of one of those squares, and then the block's orientation and type (which would need to be kept track of anyway). This method, however, also caused a plethora of problems when dealing with running into walls, and rotating the piece as well. For instance, I would rotate a piece, and then when redrawing the piece, part of it would overlap with either a wall edge or with previous blocks that were already set down. Because of those boundary cases, the application spends a good amount of time double-checking to make sure no illegal moves are being made.

As I finished up the standalone, single-player Tetris portion of the programming, I was faced with yet another series of decisions, this time regarding networking. Almost all, if not all, Android phones have a Bluetooth device built into the phone. There are plenty of applications that use Bluetooth to connect two phones together; the main limitation of this method, however, is that the two phones need to be physically nearby to each other when they connect. There was also the option of using a phone's signal with phone companies (as opposed to the Internet) in order to connect, but I discovered that using a phone service is about the same in terms of implementation and quality as using Wi-Fi, except that the phone company can block people out if they wish. So in the end, Wi-Fi was the way to go; but that was not the end of things.

There was also the question of *how* the two devices were going to connect. What I had originally imagined, and what most web-based Tetris games probably do and what a completed product would probably have required, is that there is a server running completely separately somewhere, and then all phones are all clients that connect to the server, and the server handles all of the pairing up and connecting and everything passes through the server before it is sent back to the clients. So a one-on-one game between two phones in that kind of structure would look like client -> server -> other client, so the two clients are never directly communicating. There was also the matter of TCP versus UDP – TCP guaranteed accuracy but UDP offers more speed. In the end, however, I decided to go with a TCP connection between two phones, with one phone acting as the server and the other phone acting as the client. This part was probably the most difficult part, simply because I had never used a thread, or a socket, or anything like that. The main reason why I chose to designate one phone as the server is mainly because I was running out of time for this semester, and figured that this approach would be slightly easier to implement than having a separate server and having the phone clients communicate through that server. But even then, it took me several weeks to figure out how to set up sockets that communicate with each other without the application crashing.

Another design decision had to do with the look of the application. From the beginning, I had wanted each phone to be able to see everyone's game, not just their own. And because mobile phones are in general, fairly small, space budgeting was crucial to the project. Originally, I had wanted to have the main player's game take up probably 75% of the screen, and then have thumbnails of his or her opponents' games on the side, and perhaps make those thumbnails optional even. But because I ended up only doing a two-player version of the game, I decided to just split the screen in half between the two games, since it was both better aesthetically and easier to program. If I continue to develop this game, however, how I choose to budget the space that I have will be a major factor in the playability of the game – it's one of the few limitations of multiplayer Tetris on a mobile platform, when compared to the popular web-based implementations.

Designing the second half of the screen actually took quite a bit of time as well, because of my unfamiliarity with how Android works. Originally, I wanted to create two instances of *TetrisView.java*, one on the left side representing the host phone's game, and one on the right side representing the opponent's game. Part of the way that Android applications determine what to display is actually done in XML code, as opposed to simply using pure Java. Because of that, I had to figure out the different options available in XML, and use them correctly to position the different text fields, buttons, and the games that the application uses. Because of how *TileView.java* worked, however, I actually had to make a second instance of the class, which is why there is a *TileView2.java* class. And the sad part is that the two classes are exactly the same, except for one line that offsets *TileView2.java* so that it appears on the right half of the screen inside of the left. Ideally, I would have just make *TileView.java* take in some parameter offset, and then there would be no need for a *TileView2.java*, but because of the way extending the Android SDK's *View* class works, I could not define a constructor for *TileView.java* that did not take in the parameters that it wanted, and was forced to create a *TileView2.java* (as well as a *TetrisView2.java* and a *TetrisGame2.java*).

The structure of my implementation started out very simple – one giant class of code – and as I continued programming, resulted in more and more classes, ending in 8 total classes that are used in the final product. The "main" class, or in Android terms, the activity that the application starts on, is the *Start.java* class. Everything happens in this class, and this class makes use of four different XML layout files – one for the initial screen, one for the host, one for the client, and one for the game being played. From the phone that chooses to host the game, the

*Start.java* class creates a server socket, which remains open as it listens for incoming connections. The client phone attempts to connect to the server phone after the I.P. address has been specified, and if the connection is made, communication between the two phones begins. After both phones signal that they are ready (the "Play Tetris" button), both phones switch to the *tetris_layout.xml* view, which displays two games and an overall status message.

As previously mentioned, the layout of the game is based on a tile grid, which comes from the *TileView.java* class. The *TetrisView.java* and *TetrisView2.java* classes extended *TileView* and *TileView2*, respectively, and when each class is initialized, they initiliaze a *TetrisGame* or *TetrisGame2* object, respectively. Each *TetrisGame* object keeps track of and updates the position and state of the game – meaning that the *TetrisBlock* object of each game is a part of the *TetrisGame* object. Finally, the *TetrisView* and *TetrisView2* classes determine what is shown in the main class – the state of the game, based on *TetrisView*, tells the application whether to keep going, or to exit out. That is how the application is organized.

## Conclusion

My main motivation for doing this project in particular – developing an Android application – was that I wanted to gain some experience with mobile application development. And that is exactly what I did – even though Android applications are in Java, there are subtleties that if you do not know about or do not pay attention too can cause serious problems. The whole structure of the code is different as well – the concept of an "Activity," where, in some applications, each screen is a new Activity (though in my implementation there is only one Activity), and the lack of a *main()* method, instead having *OnCreate()*, *OnStop()*, and *OnPause()*, methods, takes a little to get used to. Also, the concept of "overriding" a previously defined function is new to me, yet in Android applications it is extremely common, because the developer wants to redirect and redefine what user input does. But probably the most "subtle" trick about Android applications is that there is more to it than the one Activity class – every application has an *R.java* class where all of the attribute variables are stored; most programs use XML layouts to set the template for their activities (the activities can alter the templates afterward); the *AndroidManifest.xml* file determines which SDK versions that the application applies to, and which class file the application will run on startup. All of these little things add up to make things difficult for someone new to the Android platform, but now that I know about all of these things, I can take advantage of them to make programming easier or more efficient.

I also dealt with networking for the first time during this project – I probably spent a good day or two trying to figure out how threads worked, and why my threads kept crashing the application. And then after that, I spent another good chunk of time trying to sort of the buffers and communication between sockets – in one case where my code was ignoring one command and performing the second one twice, I spent hours reorganizing and trying to debug my code, and in the end, the problem was that I was point to something by reference rather than sending its value. Not to mention the countless number of times I got a NullPointerException throughout the course of this project – it's a good thing that I was enjoying myself while programming this application, or else I probably would have lost all motivation a long time ago.

To be honest, though, I am a little disappointed in my inability to make more progress on the application in a semester. I was expecting some setbacks, and I knew that it would be difficult, but I was hoping to have completed a bit more than the bare minimum "two people playing each other." Looking back at my original schedule of tasks, each task took about a week more than I had initially expected, which is why I finished such a small portion of the entire project. Part of my disappointment is that I probably spent an average of 6 or 7 hours on the project every week – comparable to a regular class – as well as the majority of my winter break working on programming the code, but was not even close to making my original goal. I'm satisfied in the sense that I know that I worked hard and put in the work, which makes the product so much more rewarding, but if the program had been a little further along, that would have been nice.

It's hard to say what I would have done differently, because the whole semester was more of a learning process for me than anything. Of course, if I had known everything I know now, I could have saved weeks' worth of work and gotten a lot farther than I did. But if I had to change anything though, I think it would be my usage of the *TileView.java* and *TetrisView.java* classes. I still do not quite fully understand how exactly an Android phone's *View* class works, but the way that my implementation works right now does not allow *TileView.java* and *TetrisView.java* to receive any parameters. My plan for the final structure of the application was to have a "display" class that simply took in a game state as a parameter, and then converted that game state into what the user sees on the phone. Because of Android's *View* class, however, the code cannot be modulated nicely like that, and a little extra work is required. It also took me a while to figure out a workaround because I kept trying to find a way to have every done neatly. Given more time, I might have investigated a little further into the merits of the client -> server -> client approach to save me some time when I take the step to a multiplayer version than works for more than two players, but simply doing the server <-> client approach was definitely a valuable learning experience, so I do not regret it at all.

Had I had more time, the next thing I would have done is probably polish up the one-on-one game – add some consequence for the other player doing better than you, or something along those lines, to make the game more competitive. And after that I would have looked into moving up to a four-player version (which would also require a restructuring of the layouts and how each player views the games). If I ever got to that point, where I had a game that was fully functional for four players, I would probably brush up the appearance of the application a bit, and then release it to the public, and see what they think of it. All in all though, I'm satisfied with the work I did this semester, and hopefully next semester I can either continue this project or start a new one that is just as interesting.

## Works Cited

Android Developers. 2012. 10 Jan. 2012
<http://developer.android.com/resources/samples/Snake/index.html>.

# Appendix A: Start.java

```
package com.tetris;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.util.Enumeration;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.Window;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class Start extends Activity {

        private static final String TAG = "TetrisView";

        /**
         * mode macros
         */
        public static final int PAUSE = 0;
        public static final int READY = 1;
        public static final int RUNNING = 2;
        public static final int LOSE = 3;
        public static final int WIN = 4;

        TetrisView mTetrisView;
        TetrisView2 mTetrisView2;

        /**
         * runs every mDelay seconds and checks if a new block has been created and
         * whether or not the game is over
         */
        private RefreshHandler mRefreshHandler = new RefreshHandler();

        String ICICLE_KEY = "Tetris-view";
        private TextView startText;
        /**
         * mDelay is the delay between each run of the RefreshHandler
```

```
 */
private static final long mDelay = 50;
private long mLastMove;

// server
// DEFAULT IP
public static String SERVERIP = "";
// DESIGNATE A PORT
public static final int SERVERPORT = 8080;
private TextView serverStatus;
private ServerSocket serverSocket;
private Button playTetris;
private PrintWriter serverOut;
private PrintWriter clientOut;
protected static final int MSG_ID = 0x1337;

private Handler handler = new Handler();

// client
private TextView clientStatus;
private EditText serverIp;
private Button connectPhones;
private Button clientReadyButton;
private String serverIpAddress = "";
private boolean connected = false;

private Button serverButton;
private Button clientButton;
private boolean serverReady = false;
private boolean clientReady = false;
private boolean serverSide = false;
private boolean clientSide = false;
private String fromClient = "";
private String fromServer = "";

@Override
protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);

	// remove title bar
	requestWindowFeature(Window.FEATURE_NO_TITLE);

	setContentView(R.layout.start);

	startText = (TextView) findViewById(R.id.start_text);
	serverButton = (Button) findViewById(R.id.server);
	serverButton.setOnClickListener(serverClick);
	clientButton = (Button) findViewById(R.id.client);
	clientButton.setOnClickListener(clientClick);

}

private OnClickListener serverClick = new OnClickListener() {
	@Override
	public void onClick(View v) {
		serverSide = true;
```

```
			setContentView(R.layout.server);

			serverStatus = (TextView) findViewById(R.id.server_status);
			SERVERIP = getLocalIpAddress();
			serverStatus.setText("Waiting for connection at: " + SERVERIP);
			// serverReady button
			playTetris = (Button) findViewById(R.id.play_tetris);
			playTetris.setOnClickListener(serverReadyClick);

			Thread serverThread = new Thread(new ServerThread());
			serverThread.start();
		}
	};

	private OnClickListener clientClick = new OnClickListener() {
		@Override
		public void onClick(View v) {
			clientSide = true;

			setContentView(R.layout.client);

			clientStatus = (TextView) findViewById(R.id.client_status);
			serverIp = (EditText) findViewById(R.id.server_ip);
			connectPhones = (Button) findViewById(R.id.connect_phones);
			connectPhones.setOnClickListener(connectClick);
			// clientReady button
			clientReadyButton = (Button) findViewById(R.id.play_tetris);
			clientReadyButton.setOnClickListener(clientReadyClick);

			clientStatus.setText("Enter an IP Address to connect");
		}
	};

	// server
	private OnClickListener serverReadyClick = new OnClickListener() {

		@Override
		public void onClick(View v) {
			serverReady = true;
			if (serverOut != null)
				serverOut.println("playtetris");

			if (clientReady) {
				if (mTetrisView == null) {
					setContentView(R.layout.tetris_layout);

					mTetrisView = (TetrisView) findViewById(R.id.tetris);
					mTetrisView.setTextView((TextView) findViewById(R.id.text));
					mTetrisView2 = (TetrisView2) findViewById(R.id.tetris2);

					mTetrisView.setMode(READY);
					mTetrisView2.setMode(READY);
				}
			}
		}
```

```
};

// server
private OnClickListener clientReadyClick = new OnClickListener() {

	@Override
	public void onClick(View v) {
		clientReady = true;
		if (clientOut != null)
			clientOut.println("playtetris");

		if (serverReady) {
			if (mTetrisView == null) {
				setContentView(R.layout.tetris_layout);

				mTetrisView = (TetrisView) findViewById(R.id.tetris);
				mTetrisView.setTextView((TextView) findViewById(R.id.text));
				mTetrisView2 = (TetrisView2) findViewById(R.id.tetris2);

				mTetrisView.setMode(READY);
				mTetrisView2.setMode(READY);
			}
		}
	}
};

private OnClickListener connectClick = new OnClickListener() {

	@Override
	public void onClick(View v) {
		if (!connected) {
			clientStatus.setText("Attempting to connect.");

			serverIpAddress = serverIp.getText().toString();
			if (!serverIpAddress.equals("")) {
				Thread cThread = new Thread(new ClientThread());
				cThread.start();
			}
		}
	}
};

// GETS THE IP ADDRESS OF YOUR PHONE'S NETWORK
private String getLocalIpAddress() {
	try {
		for (Enumeration<NetworkInterface> en = NetworkInterface
				.getNetworkInterfaces(); en.hasMoreElements();) {
			NetworkInterface intf = en.nextElement();
			for (Enumeration<InetAddress> enumIpAddr = intf
					.getInetAddresses(); enumIpAddr.hasMoreElements();) {
				InetAddress inetAddress = enumIpAddr.nextElement();
				if (!inetAddress.isLoopbackAddress()) {
					return inetAddress.getHostAddress().toString();
				}
			}
		}
```

```
				} catch (SocketException ex) {
					// Log.e("Start", ex.toString());
				}
				return null;
			}

			class ServerThread implements Runnable {
				public void run() {
					try {
						if (SERVERIP != null) {
							handler.post(new Runnable() {
								@Override
								public void run() {
									serverStatus
											.setText("Listening on IP: " +
		SERVERIP);
								}
							});
							serverSocket = new ServerSocket(SERVERPORT);

							while (true) {
								// LISTEN FOR INCOMING CLIENTS
								Socket client = serverSocket.accept();
								handler.post(new Runnable() {
									@Override
									public void run() {
										serverStatus.setText("Connected.");
									}
								});

								try {
									BufferedReader serverIn = new BufferedReader(
											new InputStreamReader(

			client.getInputStream()));
									serverOut = new PrintWriter(new BufferedWriter(
											new OutputStreamWriter(

			client.getOutputStream())), true);

									while ((fromClient = serverIn.readLine()) != null) {
										if (fromClient.equals("playtetris")) {
											clientReady = true;
											fromClient = "0";
										} else if (fromClient.equals("gameover")) {
											fromClient = "0";
											handler.post(new Runnable() {
												@Override
												public void run() {

			mTetrisView.setMode(WIN);

			mTetrisView2.setMode(LOSE);
												}
											});
										} else if (fromClient.equals(""))
```

```
                                    fromClient = "0";
                                if (mTetrisView2 != null
    && !fromClient.equals("0")) {
                                    final String sendValue =
fromClient;
                                    handler.post(new Runnable() {
                                        @Override
                                        public void run() {
                                            if
((mTetrisView.getMode() == READY)
    && sendValue.equals("1"))
                                                if
(mTetrisView.getMode() == READY)
    mTetrisView.pressKey(1);
    mTetrisView2.pressKey(Integer
    .parseInt(sendValue));
                                        }
                                    });
                                }
                                if (mTetrisView == null) {
                                    if (serverReady && clientReady) {
                                        handler.post(new
Runnable() {
                                            @Override
                                            public void run()
{
                                                try {
    serverSocket.close();
                                                } catch
(Exception e) {
    serverStatus
        .setText("Close failed.");
                                                }
    setContentView(R.layout.tetris_layout);
    mTetrisView = (TetrisView) findViewById(R.id.tetris);
    mTetrisView
    .setTextView((TextView) findViewById(R.id.text));
    mTetrisView2 = (TetrisView2) findViewById(R.id.tetris2);
```

```
	mTetrisView.setMode(READY);

	mTetrisView2.setMode(READY);
									}
								});
							}
						}

						update();
						mRefreshHandler.sleep(mDelay);
					}
					break;
				} catch (Exception e) {
					handler.post(new Runnable() {
						@Override
						public void run() {
							serverStatus
									.setText("Oops.
Connection interrupted.");
						}
					});
					e.printStackTrace();
				}
			}
		} else {
			handler.post(new Runnable() {
				@Override
				public void run() {
					serverStatus
							.setText("Couldn't detect a
connection.");
				}
			});
		}
	} catch (Exception e) {
		handler.post(new Runnable() {
			@Override
			public void run() {
				serverStatus.setText("ERROR");
			}
		});
		e.printStackTrace();
	}
	}
}

	public class ClientThread implements Runnable {
		public void run() {
			try {
				if (!connected) {
					InetAddress serverAddr = InetAddress
							.getByName(serverIpAddress);
					// connecting
					Socket server = new Socket(serverAddr, SERVERPORT);
```

```
                    if (server != null)
                        connected = true;

                    while (connected) {
                        if (!serverReady || !clientReady) {
                            handler.post(new Runnable() {
                                @Override
                                public void run() {
                                    clientStatus.setText("Connected.");
                                }
                            });
                        }

                        try {
                            BufferedReader clientIn = new BufferedReader(
                                    new InputStreamReader(

    server.getInputStream()));
                            clientOut = new PrintWriter(new BufferedWriter(
                                    new OutputStreamWriter(

    server.getOutputStream())), true);

                            while ((fromServer = clientIn.readLine()) != null) {
                                if (fromServer.equals("playtetris")) {
                                    serverReady = true;
                                    fromServer = "0";
                                } else if (fromServer.equals("gameover")) {
                                    fromServer = "0";
                                    handler.post(new Runnable() {
                                        @Override
                                        public void run() {

    mTetrisView.setMode(WIN);

    mTetrisView2.setMode(LOSE);
                                        }
                                    });
                                } else if (fromServer.equals(""))
                                    fromServer = "0";

                                if (mTetrisView2 != null

    && !fromServer.equals("0")) {
                                    final String sendValue =
fromServer;
                                    handler.post(new Runnable() {
                                        @Override
                                        public void run() {
                                            if
(mTetrisView.getMode() == READY)

    mTetrisView.pressKey(1);


    mTetrisView2.pressKey(Integer
        .parseInt(sendValue));
                                                            }
                                                        });
                                                    }

                                                    if (mTetrisView == null) {
                                                        if (serverReady && clientReady) {
                                                            handler.post(new
Runnable() {
                                                                @Override
                                                                public void run()
{
        setContentView(R.layout.tetris_layout);


        mTetrisView = (TetrisView) findViewById(R.id.tetris);

        mTetrisView

        .setTextView((TextView) findViewById(R.id.text));

        mTetrisView2 = (TetrisView2) findViewById(R.id.tetris2);


        mTetrisView.setMode(READY);

        mTetrisView2.setMode(READY);
                                                                }
                                                            });
                                                        }
                                                    }

                                                    update();
                                                    mRefreshHandler.sleep(mDelay);
                                                }
                                                break;
                                            } catch (Exception e) {
                                                handler.post(new Runnable() {
                                                    @Override
                                                    public void run() {
                                                        clientStatus
                                                                .setText("Oops.
Connection interrupted");
                                                    }
                                                });
                                                e.printStackTrace();
                                            }
                                        }
                                    } else {
                                        handler.post(new Runnable() {
                                            @Override
                                            public void run() {
                                                clientStatus.setText("Couldn't detect");
                                            }
```

```
                        });
                    }
                } catch (Exception e) {
                    handler.post(new Runnable() {
                        @Override
                        public void run() {
                            clientStatus.setText("ERROR");
                        }
                    });
                    e.printStackTrace();
                }
            }
        }

        @Override
        public boolean onKeyDown(int keyCode, KeyEvent msg) {
            if (!serverSide && !clientSide)
                return false;

            if ((mTetrisView.getMode() == LOSE) || (mTetrisView.getMode() == WIN))
                System.exit(0);
            if (mTetrisView.getMode() == READY) {
                if (serverSide)
                    serverOut.println("1");
                else
                    clientOut.println("1");

                mTetrisView.pressKey(1);
                mTetrisView2.pressKey(1);
                return (true);
            }

            if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
                if (serverSide)
                    serverOut.println("1");
                else
                    clientOut.println("1");

                mTetrisView.pressKey(1);
                return (true);
            }

            if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
                if (serverSide)
                    serverOut.println("2");
                else
                    clientOut.println("2");

                mTetrisView.pressKey(2);
                return (true);
            }

            if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
                if (serverSide)
                    serverOut.println("3");
                else
```

```
				clientOut.println("3");

			mTetrisView.pressKey(3);
			return (true);
		}

		if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
			if (serverSide)
				serverOut.println("4");
			else
				clientOut.println("4");

			mTetrisView.pressKey(4);
			return (true);
		}

		if (keyCode == KeyEvent.KEYCODE_SPACE
				|| keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
			if (serverSide)
				serverOut.println("5");
			else
				clientOut.println("5");

			mTetrisView.pressKey(5);
			return (true);
		}

		return super.onKeyDown(keyCode, msg);
	}

	@Override
	protected void onPause() {
		super.onPause();

		if (serverSide && serverOut != null)
			serverOut.println("gameover");
		else if (clientSide && clientOut != null)
			clientOut.println("gameover");

		System.exit(0);

		// Pause the game along with the activity
		if (mTetrisView != null)
			mTetrisView.setMode(TetrisView.PAUSE);
		if (mTetrisView2 != null)
			mTetrisView2.setMode(TetrisView.PAUSE);
	}

	@Override
	protected void onStop() {
		super.onStop();

		if (serverSide) {
			try {
				// CLOSE THE SOCKET UPON EXITING
				serverSocket.close();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		}

		@Override
		public void onSaveInstanceState(Bundle outState) {
			// Store the game state
			if (mTetrisView != null)
				outState.putBundle(ICICLE_KEY, mTetrisView.saveState());
			if (mTetrisView2 != null)
				outState.putBundle(ICICLE_KEY, mTetrisView2.saveState());
		}

		class RefreshHandler extends Handler {

			@Override
			public void handleMessage(Message msg) {
				update();
			}

			public void sleep(long delayMillis) {
				this.removeMessages(0);
				sendMessageDelayed(obtainMessage(0), delayMillis);
			}
		};

		public void update() {
			if (mTetrisView != null) {
				if (mTetrisView.getMode() == RUNNING) {
					long now = System.currentTimeMillis();

					if (now - mLastMove > mDelay) {
						if (serverSide)
							serverOut.println((10 + mTetrisView.getBlockType()));
						else
							clientOut.println((10 + mTetrisView.getBlockType()));

						mLastMove = now;
					}
					mRefreshHandler.sleep(mDelay);
				} else if (mTetrisView.getMode() == LOSE) {
					if (serverSide)
						serverOut.println("gameover");
					else
						clientOut.println("gameover");
					mTetrisView2.setMode(WIN);
				}

			}
		}
	}
```

# Appendix B: TetrisView.java

```
package com.tetris;

import android.content.Context;
import android.content.res.Resources;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.util.AttributeSet;
import android.view.View;
import android.widget.TextView;

/**
 * TetrisView: implementation of a simple game of Tetris
 */
public class TetrisView extends TileView {

        private static final String TAG = "TetrisView";

        /**
         * Current mode of application: READY to run, RUNNING, or you have already
         * lost. static final ints are used instead of an enum for performance
         * reasons.
         */
        public static final int PAUSE = 0;
        public static final int READY = 1;
        public static final int RUNNING = 2;
        public static final int LOSE = 3;
        public static final int WIN = 4;

        /**
         * Labels for the drawables that will be loaded into the TileView class
         */
        private static final int BLUEUNIT = 1;
        private static final int BROWNUNIT = 2;
        private static final int CYANUNIT = 3;
        private static final int GREENUNIT = 4;
        private static final int ORANGEUNIT = 5;
        private static final int PURPLEUNIT = 6;
        private static final int REDUNIT = 7;
        private static final int WALL = 8;
        private static final int WALLTOP = 9;

        /**
         * mScore: used to keep score mMoveDelay: number of milliseconds between
         * Tetris movements. This will decrease over time.
         */
        private static final long mMoveDelay = 50;

        /**
         * mLastMove: tracks the absolute time when the Tetris last moved, and is
         * used to determine if a move should be made based on mMoveDelay.
         */
        private long mLastMove;
```

```
/**
 * mStatusText: text shows to the user in some run states
 */
private TextView mStatusText;

/**
 * mTetrisGame: a game state containing all the relevant information about
 * the game.
 */
private TetrisGame mTetrisGame;

/**
 * Create a simple handler that we can use to cause animation to happen. We
 * set ourselves as a target and we can use the sleep() function to cause an
 * update/invalidate to occur at a later date.
 */
private RefreshHandler mRedrawHandler = new RefreshHandler();

class RefreshHandler extends Handler {

        @Override
        public void handleMessage(Message msg) {
                TetrisView.this.update();
                TetrisView.this.invalidate();
        }

        public void sleep(long delayMillis) {
                this.removeMessages(0);
                sendMessageDelayed(obtainMessage(0), delayMillis);
        }
};

/**
 * Constructs a TetrisView based on inflation from XML
 *
 * @param context
 * @param attrs
 */
public TetrisView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initTetrisView();
}

public TetrisView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        initTetrisView();
}

private void initTetrisView() {
        setFocusable(true);

        Resources r = this.getContext().getResources();

        resetTiles(10);
        loadTile(BLUEUNIT, r.getDrawable(R.drawable.blueunit));
```

```
		loadTile(BROWNUNIT, r.getDrawable(R.drawable.brownunit));
		loadTile(CYANUNIT, r.getDrawable(R.drawable.cyanunit));
		loadTile(GREENUNIT, r.getDrawable(R.drawable.greenunit));
		loadTile(ORANGEUNIT, r.getDrawable(R.drawable.orangeunit));
		loadTile(PURPLEUNIT, r.getDrawable(R.drawable.purpleunit));
		loadTile(REDUNIT, r.getDrawable(R.drawable.redunit));
		loadTile(WALL, r.getDrawable(R.drawable.wall));
		loadTile(WALLTOP, r.getDrawable(R.drawable.walltop));

		if (mTetrisGame == null)
			mTetrisGame = new TetrisGame();
	}

	/**
	 * Save game state so that the user does not lose anything if the game
	 * process is killed while we are in the background.
	 *
	 * @return a Bundle with this view's state
	 */
	public Bundle saveState() {
		Bundle map = new Bundle();

		map.putLong("mTimeDelay", Long.valueOf(mTetrisGame.getTimeDelay()));
		map.putLong("mScore", Long.valueOf(mTetrisGame.getScore()));
		map.putInt("mTetrisBlockOrientation", mTetrisGame.getOrientation());
		map.putInt("mTetrisBlockType", mTetrisGame.getBlockType());
		map.putInt("mTetrisBlock1x", Integer.valueOf(mTetrisGame.getX()));
		map.putInt("mTetrisBlock1y", Integer.valueOf(mTetrisGame.getY()));

		return map;
	}

	/**
	 * Restore game state if our process is being relaunched
	 *
	 * @param savedState
	 *          a Bundle containing the game state
	 */
	public void restoreState(Bundle savedState) {
		setMode(PAUSE);
		mTetrisGame = new TetrisGame(new TetrisBlock(
					savedState.getInt("mTetrisBlock1x"),
					savedState.getInt("mTetrisBlock1y"),
					savedState.getInt("mTetrisBlockType"),
					savedState.getInt("mTetrisBlockOrientation")),
					savedState.getLong("mScore"), savedState.getLong("mTimeDelay"));
	}

	public boolean pressKey(int input) {
		if ((mTetrisGame.getMode() == LOSE) || (mTetrisGame.getMode() == WIN)) {
			System.exit(0);
		}

		switch (input) {
		case 0: {
			return false;
		}
		case 1: {
			mTetrisGame.update(1);
			update();
			return (true);
		}
		case 2: {
			mTetrisGame.update(2);
			return (true);
		}
		case 3: {
			mTetrisGame.update(3);
			return (true);
		}
		case 4: {
			mTetrisGame.update(4);
			return (true);
		}
		case 5: {
			mTetrisGame.update(5);
			return (true);
		}
		}
		return false;
	}

	public int getBlockType() {
		return mTetrisGame.getBlockType();
	}

	public int getMode() {
		return mTetrisGame.getMode();
	}

	/**
	 * Sets the TextView that will be used to give information (such as "Game
	 * Over" to the user.
	 *
	 * @param newView
	 */
	public void setTextView(TextView newView) {
		mStatusText = newView;
	}

	/**
	 * Updates the current mode of the application (RUNNING or PAUSED or the
	 * like) as well as sets the visibility of textview for notification
	 *
	 * @param newMode
	 */
	public void setMode(int newMode) {
		if (newMode == RUNNING & mTetrisGame.getMode() != RUNNING) {
			mTetrisGame.setMode(newMode);
			mStatusText.setVisibility(View.INVISIBLE);
			update();
			return;
		}

		mTetrisGame.setMode(newMode);

		CharSequence str = "";
		if (newMode == READY) {
			str = "Tetris\nPress Any Key to Begin";
		} else if (newMode == PAUSE) {
			str = "Paused\nPress Up To Resume";
		} else if (newMode == LOSE) {
			str = "Game Over - you lose!\nScore: " + mTetrisGame.getScore()
					+ "";
		} else if (newMode == WIN) {
			str = "Victory - you win!\nScore: " + mTetrisGame.getScore() + "";
		}

		mStatusText.setText(str);
		mStatusText.setVisibility(View.VISIBLE);
	}

	/**
	 * Handles the basic update loop, checking to see if we are in the running
	 * state, determining if a move should be made, updating the Tetris's
	 * location.
	 */
	public void update() {

		mTetrisGame.checkTop();
		mTetrisGame.update(0);
		setMode(mTetrisGame.getMode());

		if (mTetrisGame.getMode() == RUNNING) {
			long now = System.currentTimeMillis();

			if (now - mLastMove > mMoveDelay) {
				clearTiles();
				mTetrisGame.updateFalling();
				mTetrisGame.clearRow();
				updateWalls();
				drawBlock();
				mLastMove = now;
			}
			mRedrawHandler.sleep(mMoveDelay);
		}
	}

	/**
	 * Draws some walls.
	 *
	 */
	private void updateWalls() {
		for (int x = 0; x < mXTileCount; x++) {
			setTile(WALL, x, 0);
			setTile(WALL, x, mYTileCount - 1);
		}
		for (int y = 1; y < mYTileCount - 1; y++) {
```

```
                setTile(WALL, 0, y);
                setTile(WALL, mXTileCount - 1, y);
            }
        }

        /**
         * Figure out which way the Tetris is going, see if he's run into anything
         * (the walls, himself). If he's not going to die, we then add to the front
         * and subtract from the rear in order to simulate motion. If we want to
         * grow him, we don't subtract from the rear.
         *
         */

        private void drawBlock() {
            int unit = mTetrisGame.getBlockType();
            TetrisBlock mTetrisBlock = new TetrisBlock(mTetrisGame.getX(),
                    mTetrisGame.getY(), unit, mTetrisGame.getOrientation());

            setTile(unit, mTetrisBlock.x1, mTetrisBlock.y1);
            setTile(unit, mTetrisBlock.x2, mTetrisBlock.y2);
            setTile(unit, mTetrisBlock.x3, mTetrisBlock.y3);
            setTile(unit, mTetrisBlock.x4, mTetrisBlock.y4);

            for (int x = 0; x < mXTileCount; x++) {
                for (int y = 0; y < mYTileCount; y++) {
                    if (mTetrisGame.getBlocks(x, y)) {
                        setTile(mTetrisGame.getColors(x, y), x, y);
                    }
                }
            }
        }
    }
```

# Appendix C: TetrisView2.java

```
package com.tetris;

import android.content.Context;
import android.content.res.Resources;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.util.AttributeSet;

/**
 * TetrisView: implementation of a simple game of Tetris
 */
public class TetrisView2 extends TileView2 {

        /**
         * Current mode of application: READY to run, RUNNING, or you have already
         * lost. static final ints are used instead of an enum for performance
         * reasons.
         */
        // private int mMode = READY;
        public static final int PAUSE = 0;
        public static final int READY = 1;
        public static final int RUNNING = 2;
        public static final int LOSE = 3;
        public static final int WIN = 4;

        /**
         * Labels for the drawables that will be loaded into the TileView class
         */
        private static final int BLUEUNIT = 1;
        private static final int BROWNUNIT = 2;
        private static final int CYANUNIT = 3;
        private static final int GREENUNIT = 4;
        private static final int ORANGEUNIT = 5;
        private static final int PURPLEUNIT = 6;
        private static final int REDUNIT = 7;
        private static final int BLACK = 8;
        private static final int WALL = 9;
        private static final int WALLTOP = 10;

        /**
         * mScore: used to keep score mMoveDelay: number of milliseconds between
         * Tetris movements. This will decrease over time.
         */
        private static final long mMoveDelay = 50;

        /**
         * mLastMove: tracks the absolute time when the Tetris last moved, and is
         * used to determine if a move should be made based on mMoveDelay.
         */
        private long mLastMove;

        /**
 * mTetrisGame: a game state containing all the relevant information about
 * the game.
 */
private TetrisGame2 mTetrisGame;

/**
 * Create a simple handler that we can use to cause animation to happen. We
 * set ourselves as a target and we can use the sleep() function to cause an
 * update/invalidate to occur at a later date.
 */
private RefreshHandler mRedrawHandler = new RefreshHandler();

class RefreshHandler extends Handler {

        @Override
        public void handleMessage(Message msg) {
                TetrisView2.this.update();
                TetrisView2.this.invalidate();
        }

        public void sleep(long delayMillis) {
                this.removeMessages(0);
                sendMessageDelayed(obtainMessage(0), delayMillis);
        }
};

/**
 * Constructs a TetrisView based on inflation from XML
 *
 * @param context
 * @param attrs
 */
public TetrisView2(Context context, AttributeSet attrs) {
        super(context, attrs);
        initTetrisView();
}

public TetrisView2(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        initTetrisView();
}

private void initTetrisView() {
        setFocusable(true);

        Resources r = this.getContext().getResources();

        resetTiles(11);
        loadTile(BLUEUNIT, r.getDrawable(R.drawable.blueunit));
        loadTile(BROWNUNIT, r.getDrawable(R.drawable.brownunit));
        loadTile(CYANUNIT, r.getDrawable(R.drawable.cyanunit));
        loadTile(GREENUNIT, r.getDrawable(R.drawable.greenunit));
        loadTile(ORANGEUNIT, r.getDrawable(R.drawable.orangeunit));
        loadTile(PURPLEUNIT, r.getDrawable(R.drawable.purpleunit));
        loadTile(REDUNIT, r.getDrawable(R.drawable.redunit));
        loadTile(BLACK, r.getDrawable(R.drawable.black));
        loadTile(WALL, r.getDrawable(R.drawable.wall));
        loadTile(WALLTOP, r.getDrawable(R.drawable.walltop));

        if (mTetrisGame == null)
            mTetrisGame = new TetrisGame2();
}

/**
 * Save game state so that the user does not lose anything if the game
 * process is killed while we are in the background.
 *
 * @return a Bundle with this view's state
 */
public Bundle saveState() {
        Bundle map = new Bundle();

        map.putLong("mTimeDelay", Long.valueOf(mTetrisGame.getTimeDelay()));
        map.putLong("mScore", Long.valueOf(mTetrisGame.getScore()));
        map.putInt("mTetrisBlockOrientation", mTetrisGame.getOrientation());
        map.putInt("mTetrisBlockType", mTetrisGame.getBlockType());
        map.putInt("mTetrisBlock1x", Integer.valueOf(mTetrisGame.getX()));
        map.putInt("mTetrisBlock1y", Integer.valueOf(mTetrisGame.getY()));

        return map;
}

/**
 * Restore game state if our process is being relaunched
 *
 * @param savedState
 *            a Bundle containing the game state
 */
public void restoreState(Bundle savedState) {
        setMode(PAUSE);
        mTetrisGame = new TetrisGame2(new TetrisBlock(
                    savedState.getInt("mTetrisBlock1x"),
                    savedState.getInt("mTetrisBlock1y"),
                    savedState.getInt("mTetrisBlockType"),
                    savedState.getInt("mTetrisBlockOrientation")),
                    savedState.getLong("mScore"), savedState.getLong("mTimeDelay"));
}

/**
 * handles key events in the game. Update the direction our Tetris is
 * traveling based on the DPAD.
 *
 * @see android.view.View#onKeyDown(int, android.os.KeyEvent)
 */
public boolean pressKey(int input) {
        switch (input) {
        case 0: {
            return false;
        }
        case 1: {
            mTetrisGame.update(1);
            update();
```

```
		return (true);
	}
	case 2: {
		mTetrisGame.update(2);
		return (true);
	}
	case 3: {
		mTetrisGame.update(3);
		return (true);
	}
	case 4: {
		mTetrisGame.update(4);
		return (true);
	}
	case 5: {
		mTetrisGame.update(5);
		return (true);
	}
	case 11: {
		if (mTetrisGame.getBlockType() != 1) {
			if (mTetrisGame.getBlockType() != 8)
				mTetrisGame.update(5);
			mTetrisGame.initNewBlock(1);
			update();
		}
		return (true);
	}
	case 12: {
		if (mTetrisGame.getBlockType() != 2) {
			if (mTetrisGame.getBlockType() != 8)
				mTetrisGame.update(5);
			mTetrisGame.initNewBlock(2);
			update();
		}
		return (true);
	}
	case 13: {
		if (mTetrisGame.getBlockType() != 3) {
			if (mTetrisGame.getBlockType() != 8)
				mTetrisGame.update(5);
			mTetrisGame.initNewBlock(3);
			update();
		}
		return (true);
	}
	case 14: {
		if (mTetrisGame.getBlockType() != 4) {
			if (mTetrisGame.getBlockType() != 8)
				mTetrisGame.update(5);
			mTetrisGame.initNewBlock(4);
			update();
		}
		return (true);
	}
	case 15: {
		if (mTetrisGame.getBlockType() != 5) {
				if (mTetrisGame.getBlockType() != 8)
					mTetrisGame.update(5);
				mTetrisGame.initNewBlock(5);
				update();
			}
			return (true);
		}
		case 16: {
			if (mTetrisGame.getBlockType() != 6) {
				if (mTetrisGame.getBlockType() != 8)
					mTetrisGame.update(5);
				mTetrisGame.initNewBlock(6);
				update();
			}
			return (true);
		}
		case 17: {
			if (mTetrisGame.getBlockType() != 7) {
				if (mTetrisGame.getBlockType() != 8)
					mTetrisGame.update(5);
				mTetrisGame.initNewBlock(7);
				update();
			}
			return (true);
		}
		}
		return false;
	}

	/**
	 * Updates the current mode of the application (RUNNING or PAUSED or the
	 * like) as well as sets the visibility of textview for notification
	 *
	 * @param newMode
	 */
	public void setMode(int newMode) {
		mTetrisGame.setMode(newMode);
	}

	/**
	 * Handles the basic update loop, checking to see if we are in the running
	 * state, determining if a move should be made, updating the Tetris's
	 * location.
	 */
	public void update() {

		mTetrisGame.checkTop();
		mTetrisGame.update(0);
		setMode(mTetrisGame.getMode());

		if (mTetrisGame.getMode() == RUNNING) {
			long now = System.currentTimeMillis();

			if (now - mLastMove > mMoveDelay) {
				clearTiles();
				mTetrisGame.updateFalling();
```

```
                    mTetrisGame.clearRow();
                    updateWalls();
                    drawBlock();
                    mLastMove = now;
                }
                mRedrawHandler.sleep(mMoveDelay);
            }
        }

        /**
         * Draws some walls.
         *
         */
        private void updateWalls() {
            for (int x = 0; x < mXTileCount; x++) {
                setTile(WALL, x, 0);
                setTile(WALL, x, mYTileCount - 1);
            }
            for (int y = 1; y < mYTileCount - 1; y++) {
                setTile(WALL, 0, y);
                setTile(WALL, mXTileCount - 1, y);
            }
        }

        /**
         * Figure out which way the Tetris is going, see if he's run into anything
         * (the walls, himself). If he's not going to die, we then add to the front
         * and subtract from the rear in order to simulate motion. If we want to
         * grow him, we don't subtract from the rear.
         *
         */

        private void drawBlock() {
            int unit = mTetrisGame.getBlockType();
            TetrisBlock mTetrisBlock = new TetrisBlock(mTetrisGame.getX(),
                    mTetrisGame.getY(), unit, mTetrisGame.getOrientation());

            setTile(unit, mTetrisBlock.x1, mTetrisBlock.y1);
            setTile(unit, mTetrisBlock.x2, mTetrisBlock.y2);
            setTile(unit, mTetrisBlock.x3, mTetrisBlock.y3);
            setTile(unit, mTetrisBlock.x4, mTetrisBlock.y4);

            for (int x = 0; x < mXTileCount; x++) {
                for (int y = 0; y < mYTileCount; y++) {
                    if (mTetrisGame.getBlocks(x, y)) {
                        setTile(mTetrisGame.getColors(x, y), x, y);
                    }
                }
            }
        }
}
```

# Appendix D: TetrisGame.java

```
package com.tetris;

import java.util.Random;

/**
 * TetrisView: implementation of a simple game of Tetris
 */
public class TetrisGame {

        private static final String TAG = "TetrisGame";

        /**
         * Current mode of application: READY to run, RUNNING, or you have already
         * lost. static final ints are used instead of an enum for performance
         * reasons.
         */
        private int mMode = READY;
        public static final int PAUSE = 0;
        public static final int READY = 1;
        public static final int RUNNING = 2;
        public static final int LOSE = 3;
        public static final int WIN = 4;

        // constants for direction
        private static final int SOUTH = 2;
        private static final int EAST = 3;
        private static final int WEST = 4;

        // types
        private static final int IBLOCK = 1;
        private static final int JBLOCK = 2;
        private static final int LBLOCK = 3;
        private static final int OBLOCK = 4;
        private static final int SBLOCK = 5;
        private static final int TBLOCK = 6;
        private static final int ZBLOCK = 7;

        /**
         * mScore: used to keep score mMoveDelay: number of milliseconds between
         * Tetris movements. This will decrease over time.
         */
        private long mScore;
        private long mTimeDelay;

        /**
         * mLastMove: tracks the absolute time when the Tetris last moved, and is
         * used to determine if a move should be made based on mMoveDelay.
         */
        private long mLastTimedMove;

        /**
         * mTetrisBlock: a list of Coordinates that make up the Tetris piece
         */
```

```
private TetrisBlock mTetrisBlock = new TetrisBlock(xSize / 2, 1,
                        1 + RNG.nextInt(7));
private boolean[][] oldBlocks = new boolean[xSize][ySize];
private int[][] savedColors = new int[xSize][ySize];

/**
 * Everyone needs a little randomness in their life
 */
private static final Random RNG = new Random();

/**
 * dimensions of the Tetris world.
 */
private static final int xSize = 12;
private static final int ySize = 22;

public TetrisGame() {
            mTetrisBlock = new TetrisBlock(xSize / 2, 1, 1 + RNG.nextInt(7));
            oldBlocks = new boolean[xSize][ySize];
            savedColors = new int[xSize][ySize];
            mScore = 0;
            mTimeDelay = 1000;
            mMode = READY;
}

public TetrisGame(int blockType) {
            mTetrisBlock = new TetrisBlock(xSize / 2, 1, blockType);
            oldBlocks = new boolean[xSize][ySize];
            savedColors = new int[xSize][ySize];
            mScore = 0;
            mTimeDelay = 1000;
            mMode = RUNNING;
}

public TetrisGame(TetrisBlock mTetrisBlock, long mScore, long mTimeDelay) {
            this.mTetrisBlock = mTetrisBlock;
            oldBlocks = new boolean[xSize][ySize];
            savedColors = new int[xSize][ySize];
            this.mScore = mScore;
            this.mTimeDelay = mTimeDelay;
            mMode = READY;
}

private void initNewGame() {
            mTetrisBlock = new TetrisBlock(xSize / 2, 1, 1 + RNG.nextInt(7));
            oldBlocks = new boolean[xSize][ySize];
            savedColors = new int[xSize][ySize];
            mScore = 0;
            mTimeDelay = 1000;
}

public void updateFalling() {
            if (mMode == RUNNING) {
                        long now = System.currentTimeMillis();

                        if (now - mLastTimedMove > mTimeDelay) {
```

```
                        if (!checkCollision())
                            mTetrisBlock.fall();
                        mLastTimedMove = now;
                    }
                }
            }

            public TetrisBlock getTetrisBlock() {
                return mTetrisBlock;
            }

            public long getScore() {
                return mScore;
            }

            public int getMode() {
                return mMode;
            }

            public long getTimeDelay() {
                return mTimeDelay;
            }

            public boolean getBlocks(int x, int y) {
                return oldBlocks[x][y];
            }

            public int getColors(int x, int y) {
                return savedColors[x][y];
            }

            public int getOrientation() {
                return mTetrisBlock.getOrientation();
            }

            public int getBlockType() {
                return mTetrisBlock.getBlockType();
            }

            public int getX() {
                return mTetrisBlock.x1;
            }

            public int getY() {
                return mTetrisBlock.y1;
            }

            public int update(int cmd) {
                inputCmd(cmd);
                checkTop();
                updateFalling();
                clearRow();
                return mMode;
            }

            public void inputCmd(int cmd) {
```

```
		// up = 1, down = 2, right = 3, left = 4, space = 5
		// no input = 0
		switch (cmd) {
		case 0: {
			break;
		}
		case 1: {
			if (mMode == READY) {
				/*
				 * At the beginning of the game, or the end of a previous one,
				 * we should start a new game.
				 */
				initNewGame();
				setMode(RUNNING);
				checkTop();
				updateFalling();
				clearRow();
				break;
			}
			if (mMode == PAUSE) {
				/*
				 * If the game is merely paused, we should just continue where
				 * we left off.
				 */
				setMode(RUNNING);
				checkTop();
				updateFalling();
				clearRow();
				break;
			}
			if (mMode == RUNNING) {
				rotateClockwise();
				checkTop();
				clearRow();
				break;
			}
		}
		case 2: {
			// if not at the bottom
			if (!checkCollision()) {
				if (!((mTetrisBlock.y1 == ySize - 2)
						|| (mTetrisBlock.y2 == ySize - 2)
						|| (mTetrisBlock.y3 == ySize - 2) || (mTetrisBlock.y4 == ySize
- 2))) {
					// if not directly above existing pieces
					if (!((oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1 + 1])
							|| (oldBlocks[mTetrisBlock.x2][mTetrisBlock.y2 + 1])
							|| (oldBlocks[mTetrisBlock.x3][mTetrisBlock.y3 + 1])
|| (oldBlocks[mTetrisBlock.x4][mTetrisBlock.y4 + 1]))) {
						mTetrisBlock.moveBlock(SOUTH);
					}
				}
			}
			break;
		}
		case 3: {
				// if not on the left edge
				if (!((mTetrisBlock.x1 < 2) || (mTetrisBlock.x2 < 2)
						|| (mTetrisBlock.x3 < 2) || (mTetrisBlock.x4 < 2))) {
					// if not directly to the right of existing pieces
					if (!((oldBlocks[mTetrisBlock.x1 - 1][mTetrisBlock.y1])
							|| (oldBlocks[mTetrisBlock.x2 - 1][mTetrisBlock.y2])
							|| (oldBlocks[mTetrisBlock.x3 - 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 - 1][mTetrisBlock.y4])))
						mTetrisBlock.moveBlock(WEST);
				}
				break;
			}
			case 4: {
				// if not on the right edge
				if (!((mTetrisBlock.x1 == (xSize - 2))
						|| (mTetrisBlock.x2 == (xSize - 2))
						|| (mTetrisBlock.x3 == (xSize - 2)) || (mTetrisBlock.x4 == (xSize - 2))))
{
					// if not directly to the left of existing pieces
					if (!((oldBlocks[mTetrisBlock.x1 + 1][mTetrisBlock.y1])
							|| (oldBlocks[mTetrisBlock.x2 + 1][mTetrisBlock.y2])
							|| (oldBlocks[mTetrisBlock.x3 + 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 + 1][mTetrisBlock.y4])))
						mTetrisBlock.moveBlock(EAST);
				}
				break;
			}
			case 5: {
				// as long as it doesn't hit anything on the way down
				while (!checkCollision())
					mTetrisBlock.moveBlock(SOUTH);
				break;
			}
			}
		}

		/**
		 * creates a new block at the top, and adds 100 to the player's score.
		 */
		private void initNewBlock() {
			if (!checkTop()) {
				// generate a new block of random type that is not the type of the
				// previous block
				int rng = 1 + RNG.nextInt(7);
				while (rng == mTetrisBlock.getBlockType())
					rng = 1 + RNG.nextInt(7);

				mTetrisBlock = new TetrisBlock(xSize / 2, 1, rng);
				mScore += 100;
			}
		}

		public void newBlock(int blockType, int x, int y) {
			mTetrisBlock = new TetrisBlock(xSize / 2, 1, blockType);
		}
```

```
/**
 * Updates the current mode of the application (RUNNING or PAUSED or the
 * like) as well as sets the visibility of textview for notification
 *
 * @param newMode
 */
public void setMode(int newMode) {
		mMode = newMode;
}

private void saveBlocks() {
		oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1] = true;
		oldBlocks[mTetrisBlock.x2][mTetrisBlock.y2] = true;
		oldBlocks[mTetrisBlock.x3][mTetrisBlock.y3] = true;
		oldBlocks[mTetrisBlock.x4][mTetrisBlock.y4] = true;

		savedColors[mTetrisBlock.x1][mTetrisBlock.y1] = mTetrisBlock
				.getBlockType();
		savedColors[mTetrisBlock.x2][mTetrisBlock.y2] = mTetrisBlock
				.getBlockType();
		savedColors[mTetrisBlock.x3][mTetrisBlock.y3] = mTetrisBlock
				.getBlockType();
		savedColors[mTetrisBlock.x4][mTetrisBlock.y4] = mTetrisBlock
				.getBlockType();
}

/**
 * Check to see if the game is over
 *
 * @return true when game is over, false otherwise
 */
public boolean checkTop() {
		for (int x = 0; x < xSize; x++) {
			if (oldBlocks[x][1]) {
				setMode(LOSE);
				return true;
			}
		}

		return false;
}

/**
 * Collision detection, and handler. Return true if collision, false if not.
 */
public boolean checkCollision() {
		// if it hits the bottom
		if (mTetrisBlock.y1 > (ySize - 3) || mTetrisBlock.y2 > (ySize - 3)
				|| mTetrisBlock.y3 > (ySize - 3)
				|| mTetrisBlock.y4 > (ySize - 3)) {
			saveBlocks();
			initNewBlock();
			clearRow();
			return true;
		}
```

```
        // if it runs into a block
        if (oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1 + 1]
                        || oldBlocks[mTetrisBlock.x2][mTetrisBlock.y2 + 1]
                        || oldBlocks[mTetrisBlock.x3][mTetrisBlock.y3 + 1]
                        || oldBlocks[mTetrisBlock.x4][mTetrisBlock.y4 + 1]) {
                saveBlocks();
                initNewBlock();
                clearRow();
                return true;
        }

        return false;
}

/**
 * check to see if there are any full rows. If there are, clear them.
 */
public void clearRow() {
        boolean full = true;
        boolean[][] temp = new boolean[xSize][ySize];
        int[][] tempColors = new int[xSize][ySize];

        for (int z = ySize - 2; z > 1; z--) {
                for (int x = 1; x < xSize - 1; x++) {
                        if (!oldBlocks[x][z])
                                full = false;

                        // ideally figure out way to break loop sooner
                        // if (!full)
                        // return;
                }

                if (full) {
                        mScore += 1000;
                        for (int x = 0; x < xSize; x++) {
                                oldBlocks[x][z] = false;
                                savedColors[x][z] = 0;
                        }

                        for (int x = 1; x < xSize - 1; x++) {
                                for (int y = z; y > 1; y--) {
                                        temp[x][y] = oldBlocks[x][y - 1];
                                        tempColors[x][y] = savedColors[x][y - 1];
                                }
                                for (int y = z + 1; y < ySize; y++) {
                                        temp[x][y] = oldBlocks[x][y];
                                        tempColors[x][y] = savedColors[x][y];
                                }
                        }
                        oldBlocks = temp;
                        savedColors = tempColors;
                }

                full = true;
        }
}
```

```
// figure out edge cases
public void rotateClockwise() {
	TetrisBlock mTempBlock = new TetrisBlock(mTetrisBlock.x1,
			mTetrisBlock.y1, mTetrisBlock.getBlockType(),
			mTetrisBlock.getOrientation());

	mTempBlock.rotateClockwise();
	if ((oldBlocks[mTempBlock.x1][mTempBlock.y1])
			|| (oldBlocks[mTempBlock.x2][mTempBlock.y2])
			|| (oldBlocks[mTempBlock.x3][mTempBlock.y3])
			|| (oldBlocks[mTempBlock.x4][mTempBlock.y4]))
		return;
	else {
		mTetrisBlock.rotateClockwise();

		// left side
		if (((mTetrisBlock.x1 < 2) || (mTetrisBlock.x2 < 2)
				|| (mTetrisBlock.x3 < 2) || (mTetrisBlock.x4 < 2))) {
			mTetrisBlock.x1 = 2;
			mTetrisBlock.refreshBlock();
		}

		// right side
		if (((mTetrisBlock.x1 > (xSize - 3))
				|| (mTetrisBlock.x2 > (xSize - 3))
				|| (mTetrisBlock.x3 > (xSize - 3)) || (mTetrisBlock.x4 > (xSize - 3)))) {
			mTetrisBlock.x1 = xSize - 3;
			mTetrisBlock.refreshBlock();
		}

		// bottom
		if ((mTetrisBlock.y1 == (ySize - 2))
				|| (mTetrisBlock.y2 == (ySize - 2))
				|| (mTetrisBlock.y3 == (ySize - 2))
				|| (mTetrisBlock.y4 == (ySize - 2)))
		// || (mTetrisBlock.y2 == (ySize - 4)) || (mTetrisBlock.y3 == (ySize
		// - 4)) || (mTetrisBlock.y4 == (ySize - 4))))
		{
			mTetrisBlock.y1 = ySize - 3;
			mTetrisBlock.refreshBlock();
		}

		// IBLOCK, bottom
		if ((mTetrisBlock.getBlockType() == IBLOCK)
				&& (mTetrisBlock.y1 > ySize - 4)) {
			if (mTetrisBlock.getOrientation() == 1) {
				mTetrisBlock.y1 = ySize - 4;
				mTetrisBlock.refreshBlock();
			}
		}

		// IBLOCK, left side
		if ((mTetrisBlock.getBlockType() == IBLOCK)
				&& (mTetrisBlock.x1 < 3)) {
			if (mTetrisBlock.getOrientation() == 0) {
```

```
                                        mTetrisBlock.x1 = 3;
                                        mTetrisBlock.refreshBlock();
                                }
                                if (mTetrisBlock.getOrientation() == 2) {
                                        mTetrisBlock.x1 = 2;
                                        mTetrisBlock.refreshBlock();
                                }
                        }

                        // IBLOCK, right side
                        if ((mTetrisBlock.getBlockType() == IBLOCK)
                                        && (mTetrisBlock.x1 > xSize - 5)) {
                                if (mTetrisBlock.getOrientation() == 0) {
                                        mTetrisBlock.x1 = xSize - 3;
                                        mTetrisBlock.refreshBlock();
                                }
                                if (mTetrisBlock.getOrientation() == 2) {
                                        mTetrisBlock.x1 = xSize - 4;
                                        mTetrisBlock.refreshBlock();
                                }
                        }

                        // IBLOCK, conflict with oldBlocks
                        if ((mTetrisBlock.getBlockType() == IBLOCK)
                                        && (oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1 + 2])) {
                                mTetrisBlock.y1 -= 1;
                                mTetrisBlock.refreshBlock();
                        }

                        // right, conflict with blocks
                        if (mTetrisBlock.x1 < (xSize - 3) || mTetrisBlock.x2 < (xSize - 3)
                                        || mTetrisBlock.x3 < (xSize - 3)
                                        || mTetrisBlock.x4 < (xSize - 3)) {
                                if (((oldBlocks[mTetrisBlock.x1 + 1][mTetrisBlock.y1])
                                                || (oldBlocks[mTetrisBlock.x2 + 1][mTetrisBlock.y2])
                                                || (oldBlocks[mTetrisBlock.x3 + 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 + 1][mTetrisBlock.y4]))) {
                                        mTetrisBlock.x1 -= 1;
                                        mTetrisBlock.refreshBlock();
                                }
                        }

                        // left, conflict with blocks
                        if (mTetrisBlock.x1 > 2 || mTetrisBlock.x2 > 2
                                        || mTetrisBlock.x3 > 2 || mTetrisBlock.x4 > 2) {
                                if (((oldBlocks[mTetrisBlock.x1 - 1][mTetrisBlock.y1])
                                                || (oldBlocks[mTetrisBlock.x2 - 1][mTetrisBlock.y2])
                                                || (oldBlocks[mTetrisBlock.x3 - 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 - 1][mTetrisBlock.y4]))) {
                                        mTetrisBlock.x1 += 1;
                                        mTetrisBlock.refreshBlock();
                                }
                        }
                }
        }
}
```

# Appendix E: TetrisGame2.java

```
package com.tetris;

/**
 * TetrisView: implementation of a simple game of Tetris
 */
public class TetrisGame2 {

	private static final String TAG = "TetrisGame";

	/**
	 * Current mode of application: READY to run, RUNNING, or you have already
	 * lost. static final ints are used instead of an enum for performance
	 * reasons.
	 */
	private int mMode = READY;
	public static final int PAUSE = 0;
	public static final int READY = 1;
	public static final int RUNNING = 2;
	public static final int LOSE = 3;
	public static final int WIN = 4;

	// constants for direction
	private static final int SOUTH = 2;
	private static final int EAST = 3;
	private static final int WEST = 4;

	// types
	private static final int IBLOCK = 1;
	private static final int JBLOCK = 2;
	private static final int LBLOCK = 3;
	private static final int OBLOCK = 4;
	private static final int SBLOCK = 5;
	private static final int TBLOCK = 6;
	private static final int ZBLOCK = 7;

	/**
	 * mScore: used to keep score mMoveDelay: number of milliseconds between
	 * Tetris movements. This will decrease over time.
	 */
	private long mScore;
	private long mTimeDelay;

	/**
	 * mLastMove: tracks the absolute time when the Tetris last moved, and is
	 * used to determine if a move should be made based on mMoveDelay.
	 */
	private long mLastTimedMove;

	/**
	 * mTetrisBlock: a list of Coordinates that make up the Tetris piece
	 */
	private TetrisBlock mTetrisBlock;
	private boolean[][] oldBlocks = new boolean[xSize][ySize];
private int[][] savedColors = new int[xSize][ySize];

/**
 * dimensions of the Tetris world.
 */
private static final int xSize = 12;
private static final int ySize = 22;

public TetrisGame2() {
		mTetrisBlock = new TetrisBlock(xSize / 2, 2, 8);
		oldBlocks = new boolean[xSize][ySize];
		savedColors = new int[xSize][ySize];
		mScore = 0;
		mTimeDelay = 1000;
		mMode = READY;
}

public TetrisGame2(int blockType) {
		mTetrisBlock = new TetrisBlock(xSize / 2, 1, blockType);
		oldBlocks = new boolean[xSize][ySize];
		savedColors = new int[xSize][ySize];
		mScore = 0;
		mTimeDelay = 1000;
		mMode = RUNNING;
}

public TetrisGame2(TetrisBlock mTetrisBlock, long mScore, long mTimeDelay) {
		this.mTetrisBlock = mTetrisBlock;
		oldBlocks = new boolean[xSize][ySize];
		savedColors = new int[xSize][ySize];
		this.mScore = mScore;
		this.mTimeDelay = mTimeDelay;
		mMode = READY;
}

private void initNewGame() {
		mTetrisBlock = new TetrisBlock(xSize / 2, 2, 8);
		oldBlocks = new boolean[xSize][ySize];
		savedColors = new int[xSize][ySize];
		mScore = 0;
		mTimeDelay = 1000;
}

public void updateFalling() {
		if (mMode == RUNNING) {
				long now = System.currentTimeMillis();

				if (now - mLastTimedMove > mTimeDelay) {
						if (!checkCollision())
								mTetrisBlock.fall();
						mLastTimedMove = now;
				}
		}
}

public TetrisBlock getTetrisBlock() {
		return mTetrisBlock;
	}

	public long getScore() {
		return mScore;
	}

	public int getMode() {
		return mMode;
	}

	public long getTimeDelay() {
		return mTimeDelay;
	}

	public boolean getBlocks(int x, int y) {
		return oldBlocks[x][y];
	}

	public int getColors(int x, int y) {
		return savedColors[x][y];
	}

	public int getOrientation() {
		return mTetrisBlock.getOrientation();
	}

	public int getBlockType() {
		return mTetrisBlock.getBlockType();
	}

	public int getX() {
		return mTetrisBlock.x1;
	}

	public int getY() {
		return mTetrisBlock.y1;
	}

	public int update(int cmd) {
		inputCmd(cmd);
		checkTop();
		updateFalling();
		clearRow();
		return mMode;
	}

	public void inputCmd(int cmd) {
		// up = 1, down = 2, right = 3, left = 4, space = 5
		// no input = 0
		switch (cmd) {
		case 0: {
			break;
		}
		case 1: {
			if (mMode == READY) {
```

```
                    /*
                     * At the beginning of the game, or the end of a previous one,
                     * we should start a new game.
                     */
                    initNewGame();
                    setMode(RUNNING);
                    checkTop();
                    updateFalling();
                    clearRow();
                    break;
                }
                if (mMode == PAUSE) {
                    /*
                     * If the game is merely paused, we should just continue where
                     * we left off.
                     */
                    setMode(RUNNING);
                    checkTop();
                    updateFalling();
                    clearRow();
                    break;
                }
                if (mMode == RUNNING) {
                    rotateClockwise();
                    checkTop();
                    clearRow();
                    break;
                }
            }
            case 2: {
                // if not at the bottom
                if (!checkCollision()) {
                    if (!((mTetrisBlock.y1 == ySize - 2)
                                || (mTetrisBlock.y2 == ySize - 2)
                                || (mTetrisBlock.y3 == ySize - 2) || (mTetrisBlock.y4 == ySize
- 2))) {
                        // if not directly above existing pieces
                        if (!((oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1 + 1])
                                    || (oldBlocks[mTetrisBlock.x2][mTetrisBlock.y2 + 1])
                                    || (oldBlocks[mTetrisBlock.x3][mTetrisBlock.y3 + 1])
|| (oldBlocks[mTetrisBlock.x4][mTetrisBlock.y4 + 1]))) {
                            mTetrisBlock.moveBlock(SOUTH);
                        }
                    }
                }
                break;
            }
            case 3: {
                // if not on the left edge
                if (!((mTetrisBlock.x1 < 2) || (mTetrisBlock.x2 < 2)
                        || (mTetrisBlock.x3 < 2) || (mTetrisBlock.x4 < 2))) {
                    // if not directly to the right of existing pieces
                    if (!((oldBlocks[mTetrisBlock.x1 - 1][mTetrisBlock.y1])
                                || (oldBlocks[mTetrisBlock.x2 - 1][mTetrisBlock.y2])
                                || (oldBlocks[mTetrisBlock.x3 - 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 - 1][mTetrisBlock.y4])))
```

```
                                        mTetrisBlock.moveBlock(WEST);
                                }
                                break;
                        }
                        case 4: {
                                // if not on the right edge
                                if (!((mTetrisBlock.x1 == (xSize - 2))
                                                || (mTetrisBlock.x2 == (xSize - 2))
                                                || (mTetrisBlock.x3 == (xSize - 2)) || (mTetrisBlock.x4 == (xSize - 2))))
{
                                        // if not directly to the left of existing pieces
                                        if (!((oldBlocks[mTetrisBlock.x1 + 1][mTetrisBlock.y1])
                                                        || (oldBlocks[mTetrisBlock.x2 + 1][mTetrisBlock.y2])
                                                        || (oldBlocks[mTetrisBlock.x3 + 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 + 1][mTetrisBlock.y4])))
                                                mTetrisBlock.moveBlock(EAST);
                                }
                                break;
                        }
                        // if (keyCode == KeyEvent.KEYCODE_SPACE || keyCode ==
                        // KeyEvent.KEYCODE_DPAD_CENTER) {
                        case 5: {
                                // as long as it doesn't hit anything on the way down
                                while (!checkCollision())
                                        mTetrisBlock.moveBlock(SOUTH);
                                break;
                        }
                        }
                }

                /**
                 * creates a new block at the top, and adds 100 to the player's score.
                 */
                public void initNewBlock(int blockType) {
                        if (!checkTop()) {
                                mTetrisBlock = new TetrisBlock(xSize / 2, 1, blockType);
                                mScore += 100;
                        }
                }

                /**
                 * Updates the current mode of the application (RUNNING or PAUSED or the
                 * like) as well as sets the visibility of textview for notification
                 *
                 * @param newMode
                 */
                public void setMode(int newMode) {
                        mMode = newMode;
                }

                private void saveBlocks() {
                        oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1] = true;
                        oldBlocks[mTetrisBlock.x2][mTetrisBlock.y2] = true;
                        oldBlocks[mTetrisBlock.x3][mTetrisBlock.y3] = true;
                        oldBlocks[mTetrisBlock.x4][mTetrisBlock.y4] = true;
```

```
		savedColors[mTetrisBlock.x1][mTetrisBlock.y1] = mTetrisBlock
				.getBlockType();
		savedColors[mTetrisBlock.x2][mTetrisBlock.y2] = mTetrisBlock
				.getBlockType();
		savedColors[mTetrisBlock.x3][mTetrisBlock.y3] = mTetrisBlock
				.getBlockType();
		savedColors[mTetrisBlock.x4][mTetrisBlock.y4] = mTetrisBlock
				.getBlockType();
	}

	/**
	 * Check to see if the game is over
	 *
	 * @return true when game is over, false otherwise
	 */
	public boolean checkTop() {
		for (int x = 0; x < xSize; x++) {
			if (oldBlocks[x][1]) {
				setMode(LOSE);
				return true;
			}
		}

		return false;
	}

	/**
	 * Collision detection, and handler. Return true if collision, false if not.
	 */
	public boolean checkCollision() {
		// if it hits the bottom
		if (mTetrisBlock.y1 > (ySize - 3) || mTetrisBlock.y2 > (ySize - 3)
				|| mTetrisBlock.y3 > (ySize - 3)
				|| mTetrisBlock.y4 > (ySize - 3)) {
			saveBlocks();
			clearRow();
			return true;
		}

		// if it runs into a block
		if (oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1 + 1]
				|| oldBlocks[mTetrisBlock.x2][mTetrisBlock.y2 + 1]
				|| oldBlocks[mTetrisBlock.x3][mTetrisBlock.y3 + 1]
				|| oldBlocks[mTetrisBlock.x4][mTetrisBlock.y4 + 1]) {
			saveBlocks();
			clearRow();
			return true;
		}

		return false;
	}

	/**
	 * check to see if there are any full rows. If there are, clear them.
	 */
	public void clearRow() {
		boolean full = true;
		boolean[][] temp = new boolean[xSize][ySize];
		int[][] tempColors = new int[xSize][ySize];

		for (int z = ySize - 2; z > 1; z--) {
			for (int x = 1; x < xSize - 1; x++) {
				if (!oldBlocks[x][z])
					full = false;
			}

			if (full) {
				mScore += 1000;
				for (int x = 0; x < xSize; x++) {
					oldBlocks[x][z] = false;
					savedColors[x][z] = 0;
				}

				for (int x = 1; x < xSize - 1; x++) {
					for (int y = z; y > 1; y--) {
						temp[x][y] = oldBlocks[x][y - 1];
						tempColors[x][y] = savedColors[x][y - 1];
					}
					for (int y = z + 1; y < ySize; y++) {
						temp[x][y] = oldBlocks[x][y];
						tempColors[x][y] = savedColors[x][y];
					}
				}
				oldBlocks = temp;
				savedColors = tempColors;
			}

			full = true;
		}
	}

	// figure out edge cases
	public void rotateClockwise() {
		TetrisBlock mTempBlock = new TetrisBlock(mTetrisBlock.x1,
				mTetrisBlock.y1, mTetrisBlock.getBlockType(),
				mTetrisBlock.getOrientation());

		mTempBlock.rotateClockwise();
		if ((oldBlocks[mTempBlock.x1][mTempBlock.y1])
				|| (oldBlocks[mTempBlock.x2][mTempBlock.y2])
				|| (oldBlocks[mTempBlock.x3][mTempBlock.y3])
				|| (oldBlocks[mTempBlock.x4][mTempBlock.y4]))
			return;
		else {
			mTetrisBlock.rotateClockwise();

			// left side
			if (((mTetrisBlock.x1 < 2) || (mTetrisBlock.x2 < 2)
					|| (mTetrisBlock.x3 < 2) || (mTetrisBlock.x4 < 2))) {
				mTetrisBlock.x1 = 2;
				mTetrisBlock.refreshBlock();
			}
```

```
// right side
if (((mTetrisBlock.x1 > (xSize - 3))
                || (mTetrisBlock.x2 > (xSize - 3))
                || (mTetrisBlock.x3 > (xSize - 3)) || (mTetrisBlock.x4 > (xSize - 3)))) {
        mTetrisBlock.x1 = xSize - 3;
        mTetrisBlock.refreshBlock();
}

// bottom
if ((mTetrisBlock.y1 == (ySize - 2))
                || (mTetrisBlock.y2 == (ySize - 2))
                || (mTetrisBlock.y3 == (ySize - 2))
                || (mTetrisBlock.y4 == (ySize - 2)))
// || (mTetrisBlock.y2 == (ySize - 4)) || (mTetrisBlock.y3 == (ySize
// - 4)) || (mTetrisBlock.y4 == (ySize - 4))))
{
        mTetrisBlock.y1 = ySize - 3;
        mTetrisBlock.refreshBlock();
}

// IBLOCK, bottom
if ((mTetrisBlock.getBlockType() == IBLOCK)
                && (mTetrisBlock.y1 > ySize - 4)) {
        if (mTetrisBlock.getOrientation() == 1) {
                mTetrisBlock.y1 = ySize - 4;
                mTetrisBlock.refreshBlock();
        }
}

// IBLOCK, left side
if ((mTetrisBlock.getBlockType() == IBLOCK)
                && (mTetrisBlock.x1 < 3)) {
        if (mTetrisBlock.getOrientation() == 0) {
                mTetrisBlock.x1 = 3;
                mTetrisBlock.refreshBlock();
        }
        if (mTetrisBlock.getOrientation() == 2) {
                mTetrisBlock.x1 = 2;
                mTetrisBlock.refreshBlock();
        }
}

// IBLOCK, right side
if ((mTetrisBlock.getBlockType() == IBLOCK)
                && (mTetrisBlock.x1 > xSize - 5)) {
        if (mTetrisBlock.getOrientation() == 0) {
                mTetrisBlock.x1 = xSize - 3;
                mTetrisBlock.refreshBlock();
        }
        if (mTetrisBlock.getOrientation() == 2) {
                mTetrisBlock.x1 = xSize - 4;
                mTetrisBlock.refreshBlock();
        }
}
```

```
                        // IBLOCK, conflict with oldBlocks
                        if ((mTetrisBlock.getBlockType() == IBLOCK)
                                        && (oldBlocks[mTetrisBlock.x1][mTetrisBlock.y1 + 2])) {
                                mTetrisBlock.y1 -= 1;
                                mTetrisBlock.refreshBlock();
                        }

                        // right, conflict with blocks
                        if (mTetrisBlock.x1 < (xSize - 3) || mTetrisBlock.x2 < (xSize - 3)
                                        || mTetrisBlock.x3 < (xSize - 3)
                                        || mTetrisBlock.x4 < (xSize - 3)) {
                                if (((oldBlocks[mTetrisBlock.x1 + 1][mTetrisBlock.y1])
                                                || (oldBlocks[mTetrisBlock.x2 + 1][mTetrisBlock.y2])
                                                || (oldBlocks[mTetrisBlock.x3 + 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 + 1][mTetrisBlock.y4]))) {
                                        mTetrisBlock.x1 -= 1;
                                        mTetrisBlock.refreshBlock();
                                }
                        }

                        // left, conflict with blocks
                        if (mTetrisBlock.x1 > 2 || mTetrisBlock.x2 > 2
                                        || mTetrisBlock.x3 > 2 || mTetrisBlock.x4 > 2) {
                                if (((oldBlocks[mTetrisBlock.x1 - 1][mTetrisBlock.y1])
                                                || (oldBlocks[mTetrisBlock.x2 - 1][mTetrisBlock.y2])
                                                || (oldBlocks[mTetrisBlock.x3 - 1][mTetrisBlock.y3]) ||
(oldBlocks[mTetrisBlock.x4 - 1][mTetrisBlock.y4]))) {
                                        mTetrisBlock.x1 += 1;
                                        mTetrisBlock.refreshBlock();
                                }
                        }
                }
        }
}
```

## Appendix F: TetrisBlock.java

```
package com.tetris;

public class TetrisBlock {
	private static final String TAG = "TetrisView";

	private int blockType;
	private static final int IBLOCK = 1;
	private static final int JBLOCK = 2;
	private static final int LBLOCK = 3;
	private static final int OBLOCK = 4;
	private static final int SBLOCK = 5;
	private static final int TBLOCK = 6;
	private static final int ZBLOCK = 7;
	private static final int BLACK = 8;

	private int mOrientation = FACEUP;
	private static final int FACEUP = 0;
	private static final int FACERIGHT = 1;
	private static final int FACEDOWN = 2;
	private static final int FACELEFT = 3;

	private static final int SOUTH = 2;
	private static final int EAST = 3;
	private static final int WEST = 4;

	public int x1;
	public int x2;
	public int x3;
	public int x4;
	public int y1;
	public int y2;
	public int y3;
	public int y4;

	public TetrisBlock(int newx1, int newy1, int newBlockType) {
		x1 = newx1;
		y1 = newy1;
		blockType = newBlockType;
		mOrientation = FACEUP;
		refreshBlock();
	}

	public TetrisBlock(int newx1, int newy1, int newBlockType,
			int newOrientation) {
		x1 = newx1;
		y1 = newy1;
		blockType = newBlockType;
		mOrientation = newOrientation;
		refreshBlock();
	}

	public int getOrientation() {
		return mOrientation;
	}
```

```
public int getBlockType() {
        return blockType;
}

public void setBlockType(int blockType) {
        this.blockType = blockType;
}

public void fall() {
        y1 += 1;
        refreshBlock();
}

public void moveBlock(int mInputDirection) {
        switch (mInputDirection) {
        case EAST: {
                x1 += 1;
                x2 += 1;
                x3 += 1;
                x4 += 1;
                break;
        }
        case WEST: {
                x1 -= 1;
                x2 -= 1;
                x3 -= 1;
                x4 -= 1;
                break;
        }
        case SOUTH: {
                y1 += 1;
                y2 += 1;
                y3 += 1;
                y4 += 1;
                break;
        }
        }
}

public void flip0() {
        switch (blockType) {
        case IBLOCK: {
                x2 = x1 + 1;
                x3 = x1 - 1;
                x4 = x1 - 2;
                y2 = y1;
                y3 = y1;
                y4 = y1;
                break;
        }
        case JBLOCK: {
                x2 = x1 - 1;
                x3 = x1 + 1;
                x4 = x1 + 1;
                y2 = y1;
```

```
                    y3 = y1;
                    y4 = y1 + 1;
                    break;
          }
          case LBLOCK: {
                    x2 = x1 + 1;
                    x3 = x1 - 1;
                    x4 = x1 - 1;
                    y2 = y1;
                    y3 = y1;
                    y4 = y1 + 1;
                    break;
          }
          case OBLOCK: {
                    x2 = x1 + 1;
                    x3 = x1;
                    x4 = x1 + 1;
                    y2 = y1;
                    y3 = y1 + 1;
                    y4 = y1 + 1;
                    break;
          }
          case SBLOCK: {
                    x2 = x1 - 1;
                    x3 = x1;
                    x4 = x1 + 1;
                    y2 = y1;
                    y3 = y1 - 1;
                    y4 = y1 - 1;
                    break;
          }
          case TBLOCK: {
                    x2 = x1;
                    x3 = x1 - 1;
                    x4 = x1 + 1;
                    y2 = y1 + 1;
                    y3 = y1;
                    y4 = y1;
                    break;
          }
          case ZBLOCK: {
                    x2 = x1 + 1;
                    x3 = x1;
                    x4 = x1 - 1;
                    y2 = y1;
                    y3 = y1 - 1;
                    y4 = y1 - 1;
                    break;
          }
          }
}

public void flip90() {
          switch (blockType) {
          case IBLOCK: {
                    x2 = x1;
```

```
        x3 = x1;
        x4 = x1;
        y2 = y1 - 1;
        y3 = y1 + 1;
        y4 = y1 + 2;
        break;
}
case JBLOCK: {
        x2 = x1;
        x3 = x1;
        x4 = x1 - 1;
        y2 = y1 - 1;
        y3 = y1 + 1;
        y4 = y1 + 1;
        break;
}
case LBLOCK: {
        x2 = x1;
        x3 = x1;
        x4 = x1 - 1;
        y2 = y1 + 1;
        y3 = y1 - 1;
        y4 = y1 - 1;
        break;
}
case OBLOCK: {
        x2 = x1 + 1;
        x3 = x1;
        x4 = x1 + 1;
        y2 = y1;
        y3 = y1 + 1;
        y4 = y1 + 1;
        break;
}
case SBLOCK: {
        x2 = x1;
        x3 = x1 + 1;
        x4 = x1 + 1;
        y2 = y1 - 1;
        y3 = y1;
        y4 = y1 + 1;
        break;
}
case TBLOCK: {
        x2 = x1 - 1;
        x3 = x1;
        x4 = x1;
        y2 = y1;
        y3 = y1 - 1;
        y4 = y1 + 1;
        break;
}
case ZBLOCK: {
        x2 = x1;
        x3 = x1 + 1;
        x4 = x1 + 1;
```

```
                        y2 = y1 + 1;
                        y3 = y1;
                        y4 = y1 - 1;
                        break;
                }
                }
        }

        public void flip180() {
                switch (blockType) {
                case IBLOCK: {
                        x2 = x1 - 1;
                        x3 = x1 + 1;
                        x4 = x1 + 2;
                        y2 = y1;
                        y3 = y1;
                        y4 = y1;
                        break;
                }
                case JBLOCK: {
                        x2 = x1 + 1;
                        x3 = x1 - 1;
                        x4 = x1 - 1;
                        y2 = y1;
                        y3 = y1;
                        y4 = y1 - 1;
                        break;
                }
                case LBLOCK: {
                        x2 = x1 - 1;
                        x3 = x1 + 1;
                        x4 = x1 + 1;
                        y2 = y1;
                        y3 = y1;
                        y4 = y1 - 1;
                        break;
                }
                case OBLOCK: {
                        x2 = x1 + 1;
                        x3 = x1;
                        x4 = x1 + 1;
                        y2 = y1;
                        y3 = y1 + 1;
                        y4 = y1 + 1;
                        break;
                }
                case SBLOCK: {
                        x2 = x1 + 1;
                        x3 = x1;
                        x4 = x1 - 1;
                        y2 = y1;
                        y3 = y1 + 1;
                        y4 = y1 + 1;
                        break;
                }
                case TBLOCK: {
```

```
			x2 = x1;
			x3 = x1 + 1;
			x4 = x1 - 1;
			y2 = y1 - 1;
			y3 = y1;
			y4 = y1;
			break;
		}
		case ZBLOCK: {
			x2 = x1 - 1;
			x3 = x1;
			x4 = x1 + 1;
			y2 = y1;
			y3 = y1 + 1;
			y4 = y1 + 1;
			break;
		}
		}
	}

	public void flip270() {
		switch (blockType) {
		case IBLOCK: {
			x2 = x1;
			x3 = x1;
			x4 = x1;
			y2 = y1 + 1;
			y3 = y1 - 1;
			y4 = y1 - 2;
			break;
		}
		case JBLOCK: {
			x2 = x1;
			x3 = x1;
			x4 = x1 + 1;
			y2 = y1 + 1;
			y3 = y1 - 1;
			y4 = y1 - 1;
			break;
		}
		case LBLOCK: {
			x2 = x1;
			x3 = x1;
			x4 = x1 + 1;
			y2 = y1 - 1;
			y3 = y1 + 1;
			y4 = y1 + 1;
			break;
		}
		case OBLOCK: {
			x2 = x1 + 1;
			x3 = x1;
			x4 = x1 + 1;
			y2 = y1;
			y3 = y1 + 1;
			y4 = y1 + 1;
```

```
                        break;
                }
                case SBLOCK: {
                        x2 = x1;
                        x3 = x1 - 1;
                        x4 = x1 - 1;
                        y2 = y1 + 1;
                        y3 = y1;
                        y4 = y1 - 1;
                        break;
                }
                case TBLOCK: {
                        x2 = x1 + 1;
                        x3 = x1;
                        x4 = x1;
                        y2 = y1;
                        y3 = y1 + 1;
                        y4 = y1 - 1;
                        break;
                }
                case ZBLOCK: {
                        x2 = x1;
                        x3 = x1 - 1;
                        x4 = x1 - 1;
                        y2 = y1 - 1;
                        y3 = y1;
                        y4 = y1 + 1;
                        break;
                }
                }
        }

        public void refreshBlock() {
                switch (mOrientation) {
                case FACEUP: {
                        flip0();
                        break;
                }
                case FACERIGHT: {
                        flip90();
                        break;
                }
                case FACEDOWN: {
                        flip180();
                        break;
                }
                case FACELEFT: {
                        flip270();
                        break;
                }
                }
        }

        public void rotateClockwise() {
                mOrientation = (mOrientation + 1) % 4;
```

```
			if (blockType == IBLOCK) {
				if (mOrientation == FACEDOWN)
					x1 -= 1;
				if (mOrientation == FACEUP)
					x1 += 1;
			}

			refreshBlock();
		}
}
```

# Appendix G: TileView.java

```
package com.tetris;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.drawable.Drawable;
import android.util.AttributeSet;
import android.util.DisplayMetrics;
import android.view.View;

/**
 * TileView: a View-variant designed for handling arrays of "icons" or other
 * drawables.
 *
 */
public class TileView extends View {

	/**
	 * Parameters controlling the size of the tiles and their range within view.
	 * Width/Height are in pixels, and Drawables will be scaled to fit to these
	 * dimensions. X/Y Tile Counts are the number of tiles that will be drawn.
	 */

	protected static int mTileSize;

	// want it to be 10x20, add 2 to each dimension for walls.
	protected final static int mXTileCount = 12;
	protected final static int mYTileCount = 22;

	private static int mXOffset;
	private static int mYOffset;

	/**
	 * A hash that maps integer handles specified by the subclasser to the
	 * drawable that will be used for that reference
	 */
	private Bitmap[] mTileArray;

	/**
	 * A two-dimensional array of integers in which the number represents the
	 * index of the tile that should be drawn at that locations
	 */
	private int[][] mTileGrid;

	private final Paint mPaint = new Paint();

	public TileView(Context context, AttributeSet attrs, int defStyle) {
		super(context, attrs, defStyle);

		DisplayMetrics display = getResources().getDisplayMetrics();
		int w = display.widthPixels;
		mTileSize = (int) (Math.floor(w / mYTileCount) * .9);
	}
```

```
public TileView(Context context, AttributeSet attrs) {
        super(context, attrs);

        DisplayMetrics display = getResources().getDisplayMetrics();
        int w = display.widthPixels;
        mTileSize = (int) (Math.floor(w / mYTileCount) * .9);
}

/**
 * Rests the internal array of Bitmaps used for drawing tiles, and sets the
 * maximum index of tiles to be inserted
 *
 * @param tilecount
 */

public void resetTiles(int tilecount) {
        mTileArray = new Bitmap[tilecount];
}

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        mXOffset = 0;
        mYOffset = ((h - (mTileSize * mYTileCount)) / 2);

        mTileGrid = new int[mXTileCount][mYTileCount];
        clearTiles();
}

/**
 * Function to set the specified Drawable as the tile for a particular
 * integer key.
 *
 * @param key
 * @param tile
 */
public void loadTile(int key, Drawable tile) {
        Bitmap bitmap = Bitmap.createBitmap(mTileSize, mTileSize,
                        Bitmap.Config.ARGB_8888);
        Canvas canvas = new Canvas(bitmap);
        tile.setBounds(0, 0, mTileSize, mTileSize);
        tile.draw(canvas);

        mTileArray[key] = bitmap;
}

/**
 * Resets all tiles to 0 (empty)
 *
 */
public void clearTiles() {
        for (int x = 0; x < mXTileCount; x++) {
                for (int y = 0; y < mYTileCount; y++) {
                        setTile(0, x, y);
                }
        }
	}

	/**
	 * Used to indicate that a particular tile (set with loadTile and referenced
	 * by an integer) should be drawn at the given x/y coordinates during the
	 * next invalidate/draw cycle.
	 *
	 * @param tileindex
	 * @param x
	 * @param y
	 */
	public void setTile(int tileindex, int x, int y) {
		if (mTileGrid == null)
			mTileGrid = new int[mXTileCount][mYTileCount];

		mTileGrid[x][y] = tileindex;
	}

	@Override
	public void onDraw(Canvas canvas) {
		for (int x = 0; x < mXTileCount; x += 1) {
			for (int y = 0; y < mYTileCount; y += 1) {
				if (mTileGrid[x][y] > 0) {
					canvas.drawBitmap(mTileArray[mTileGrid[x][y]], mXOffset + x
							* mTileSize, mYOffset + y * mTileSize, mPaint);
				}
			}
		}
	}
}
```

## Appendix H: TileView2.java

```
package com.tetris;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.drawable.Drawable;
import android.util.AttributeSet;
import android.util.DisplayMetrics;
import android.util.Log;
import android.view.View;

/**
 * TileView: a View-variant designed for handling arrays of "icons" or other
 * drawables.
 *
 */
public class TileView2 extends View {

	/**
	 * Parameters controlling the size of the tiles and their range within view.
	 * Width/Height are in pixels, and Drawables will be scaled to fit to these
	 * dimensions. X/Y Tile Counts are the number of tiles that will be drawn.
	 */

	protected static int mTileSize;

	// want it to be 10x20, add 2 to each dimension for walls.
	protected final static int mXTileCount = 12;
	protected final static int mYTileCount = 22;

	private static int mXOffset;
	private static int mYOffset;

	/**
	 * A hash that maps integer handles specified by the subclasser to the
	 * drawable that will be used for that reference
	 */
	private Bitmap[] mTileArray;

	/**
	 * A two-dimensional array of integers in which the number represents the
	 * index of the tile that should be drawn at that locations
	 */
	private int[][] mTileGrid;

	private final Paint mPaint = new Paint();

	public TileView2(Context context, AttributeSet attrs, int defStyle) {
		super(context, attrs, defStyle);

		DisplayMetrics display = getResources().getDisplayMetrics();
```

```
        int w = display.widthPixels;
        mTileSize = (int) (Math.floor(w / mYTileCount) * .9);
}

public TileView2(Context context, AttributeSet attrs) {
        super(context, attrs);

        DisplayMetrics display = getResources().getDisplayMetrics();
        int w = display.widthPixels;
        mTileSize = (int) (Math.floor(w / mYTileCount) * .9);
}

/**
 * Rests the internal array of Bitmaps used for drawing tiles, and sets the
 * maximum index of tiles to be inserted
 *
 * @param tilecount
 */

public void resetTiles(int tilecount) {
        mTileArray = new Bitmap[tilecount];
}

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        mXOffset = (w / 2);
        mYOffset = ((h - (mTileSize * mYTileCount)) / 2);

        mTileGrid = new int[mXTileCount][mYTileCount];
        clearTiles();
}

/**
 * Function to set the specified Drawable as the tile for a particular
 * integer key.
 *
 * @param key
 * @param tile
 */
public void loadTile(int key, Drawable tile) {
        Bitmap bitmap = Bitmap.createBitmap(mTileSize, mTileSize,
                        Bitmap.Config.ARGB_8888);
        Canvas canvas = new Canvas(bitmap);
        tile.setBounds(0, 0, mTileSize, mTileSize);
        tile.draw(canvas);

        mTileArray[key] = bitmap;
}

/**
 * Resets all tiles to 0 (empty)
 *
 */
public void clearTiles() {
        for (int x = 0; x < mXTileCount; x++) {
                for (int y = 0; y < mYTileCount; y++) {
                        setTile(0, x, y);
                    }
                }
            }

            /**
             * Used to indicate that a particular tile (set with loadTile and referenced
             * by an integer) should be drawn at the given x/y coordinates during the
             * next invalidate/draw cycle.
             *
             * @param tileindex
             * @param x
             * @param y
             */
            public void setTile(int tileindex, int x, int y) {
                if (mTileGrid == null)
                    mTileGrid = new int[mXTileCount][mYTileCount];

                mTileGrid[x][y] = tileindex;
            }

            @Override
            public void onDraw(Canvas canvas) {
                for (int x = 0; x < mXTileCount; x += 1) {
                    for (int y = 0; y < mYTileCount; y += 1) {
                        if (mTileGrid[x][y] > 0) {
                            canvas.drawBitmap(mTileArray[mTileGrid[x][y]], mXOffset + x
                                    * mTileSize, mYOffset + y * mTileSize, mPaint);
                        }
                    }
                }
            }
        }
```

## Appendix I: start.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

<TextView
   android:id="@+id/start_text"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center_horizontal"
   android:gravity="center_horizontal"
   android:text="Welcome to Tetris.\n Choose a side to begin."
   android:textColor="#ffffff"
   android:textSize="24sp"
   android:visibility="visible" />


<Button
   android:id="@+id/server"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center_horizontal"
   android:text="Server" />


<Button
   android:id="@+id/client"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center_horizontal"
   android:text="Client" />

</LinearLayout>
```

## Appendix J: server.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

<TextView
                        android:id="@+id/server_status"
                                    android:text="@string/server_layout_text"
                                    android:visibility="visible"
                                    android:layout_width="wrap_content"
                                    android:layout_height="wrap_content"
                                    android:layout_centerInParent="true"
                                    android:gravity="center_horizontal"
                                    android:textColor="#ffffff"
                                    android:textSize="24sp"/>

<Button
                android:id="@+id/play_tetris"
                                    android:text="@string/play_button"
                                    android:layout_width="wrap_content"
                                    android:layout_height="wrap_content"/>

</LinearLayout>
```

# Appendix K: client.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

<TextView
                    android:id="@+id/client_status"
                             android:text="@string/client_layout_text"
                             android:visibility="visible"
                             android:layout_width="wrap_content"
                             android:layout_height="wrap_content"
                             android:layout_centerInParent="true"
                             android:gravity="center_horizontal"
                             android:textColor="#ffffff"
                             android:textSize="24sp"/>

<EditText
                    android:id="@+id/server_ip"
                             android:text="10.0.2.2"
                             android:visibility="visible"
                             android:layout_width="wrap_content"
                             android:layout_height="wrap_content"
                             android:gravity="center_horizontal"
                             android:textColor="#ffffff"
                             android:textSize="24sp"/>

<Button
             android:id="@+id/connect_phones"
                             android:text="@string/client_layout_text"
                             android:layout_width="wrap_content"
                             android:layout_height="wrap_content"/>

<Button
             android:id="@+id/play_tetris"
                             android:text="@string/play_button"
                             android:layout_width="wrap_content"
                             android:layout_height="wrap_content"/>
</LinearLayout>
```

# Appendix L: tetris_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <view class="com.tetris.TetrisView"
                android:id="@+id/tetris"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>

        <view class="com.tetris.TetrisView2"
                android:id="@+id/tetris2"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>

        <TextView
          android:id="@+id/text"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_alignParentTop="true"
          android:layout_centerHorizontal="true"
          android:layout_weight="1"
          android:gravity="center_horizontal"
          android:text=""
          android:textColor="#ffffff"
          android:textSize="24sp"
          android:visibility="visible" />

</RelativeLayout>
```