Table of Contents

Figures

1. Introduction

This "Database Transactions Project" implements a game similar to Tetris. The entire state of the game is stored in a collection of four database tables. A game in progress can be exited at any time without loss of information. When the game is resumed, the game's state is recovered from the database tables. The game continues right where it left off including the score, the configuration of the "game board", and the position and orientation of the current "Falling Block" (if any). The database also stores the score for every game and the date each game started. Figure 1 shows a Tetris game in two different states and identifies some of the game elements whose attributes are stored in database tables.

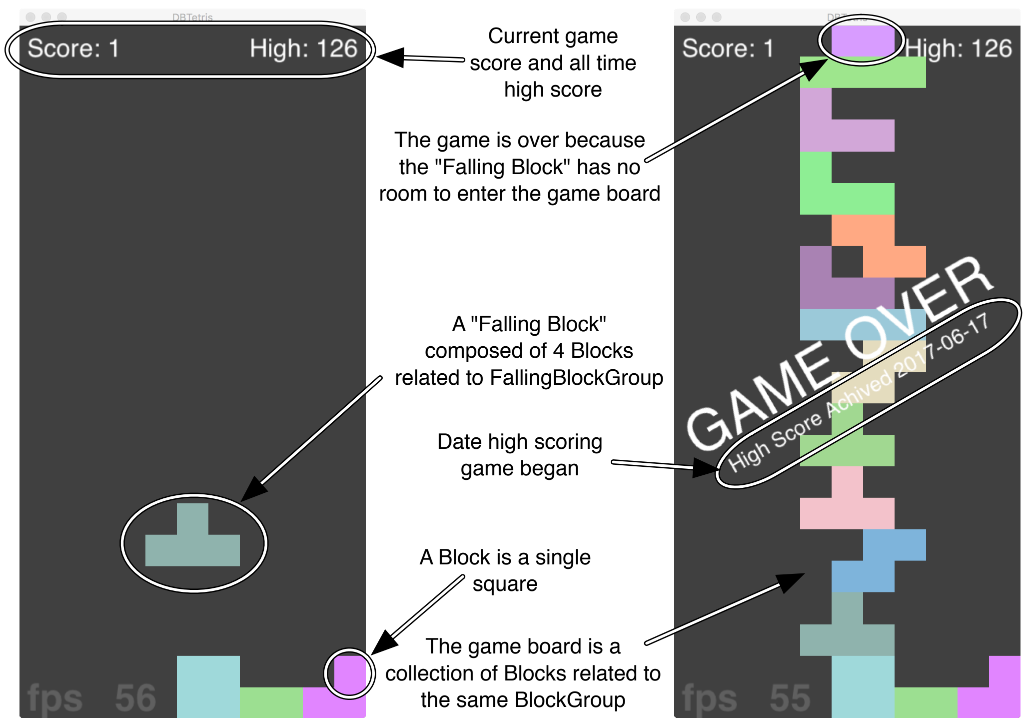

Figure 1 Game in Progress and Game Over

2. Overall Game Design

This section describes the overall design of this Database Transactions Project. All of the code and data within the project are partitioned into two subsystems, the View and the Model. For this project, the View subsystem is trivial and exists only as a way to visualize and stimulate the database transactions that occur within the Model subsystem.

2.1. View

The View subsystem uses the open source Cocos2D library for high performance 2D display. The View does not contain any Structured Query Language (SQL) and has no dependencies upon the way game data is stored or the "game rules" a.k.a. "Business Logic". The View subsystem merely depicts the current game, accepts input from the user, and periodically asks the Model subsystem to update the game state.

Accepted user inputs consist of the following:

Left Arrow Key – Request that the Model subsystem move the current "Falling Block" (if any) to the left. The Model subsystem may ignore/rollback the request based on the game rules encapsulated within the Model.

Right Arrow Key – Request that the Model subsystem move the current "Falling Block" (if any) to the right. The Model subsystem may ignore/rollback the request based on the game rules encapsulated within the Model.

A Key – Request that the Model subsystem rotate the current "Falling Block" (if any) counterclockwise. The Model subsystem may ignore/rollback the request based on the game rules encapsulated within the Model.

D Key – Request that the Model subsystem rotate the current "Falling Block" (if any) to the clockwise. The Model subsystem may ignore/rollback the request based on the game rules encapsulated within the Model.

SPACE Key – This input is handled entirely within the View subsystem. The View subsystem periodically requests that the Model subsystem update the game state. When the SPACE Key is pressed, the View subsystem requests Model updates at a faster rate thereby accelerating game play. The SPACE Key has the effect of making the current "Falling Block" (if any) fall faster. The View subsystem reverts to its normal rate of requesting Model updates when the View subsystem detects that there is no current "Falling Block" i.e. when the Model deletes the "Falling Block" because the "Falling Block" can no longer fall.

The entire View subsystem is implemented in one source code file, *Game.py.*

Note: The View subsystem uses Cocos2D *Actions* that are "co-routines" and may run concurrently when there are no data dependencies between *Actions*. The concurrent nature of the project implementation is described in Section 5.3.

2.2. Model

The game Model subsystem contains game specific data and "game rules" or "business logic". The Model has no dependencies on the View. It doesn't matter how the Model is displayed to the user. The game logic can be executed even when there is no display e.g. in a long running simulation. All game logic takes place exclusively in the Model, and all Model state is stored exclusively as tuples within database tables.

The database tables and database transactions that implement the Model are described in subsections 3 and 3.5. Figure 2 is the Entity Relationship Diagram for the Tetris game database.

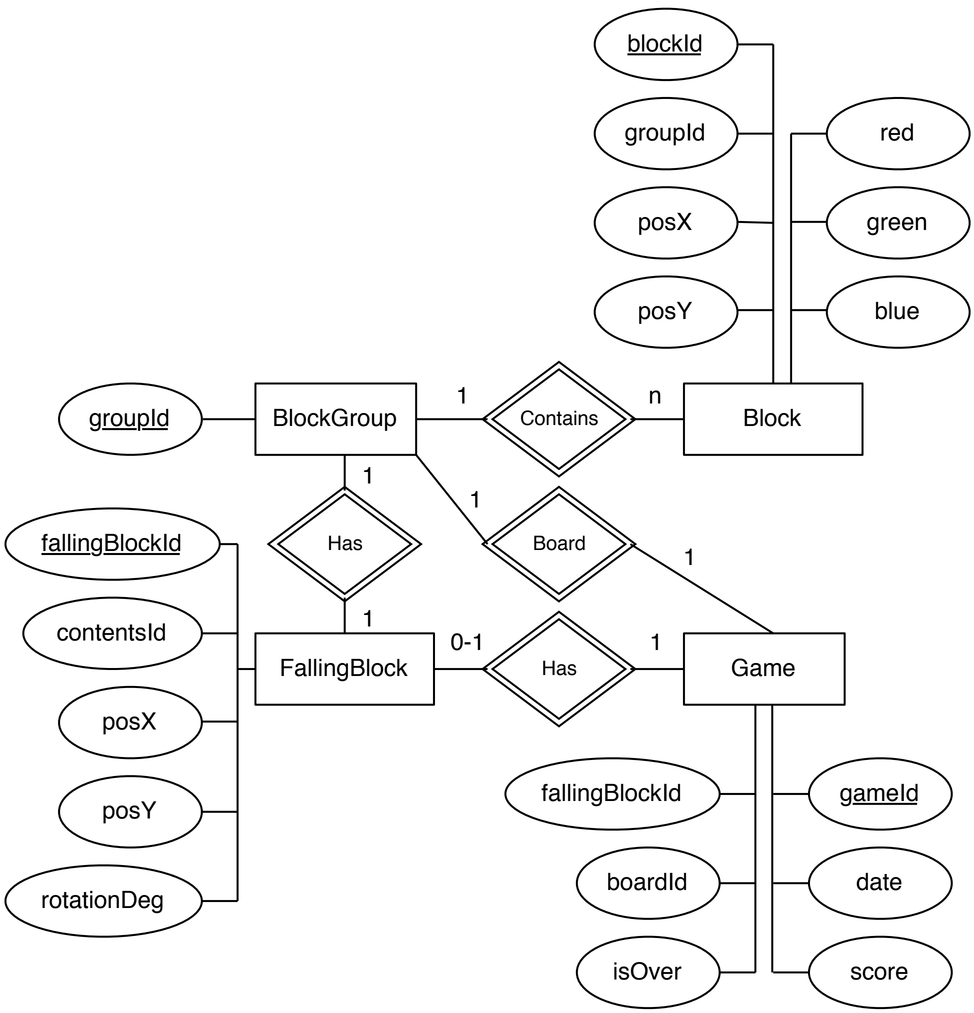

Figure 2 Entity Relationship Diagram for Tetris Game Database

Figure 3 uses UML Notation to describe the same entities and relations depicted in Figure 2.

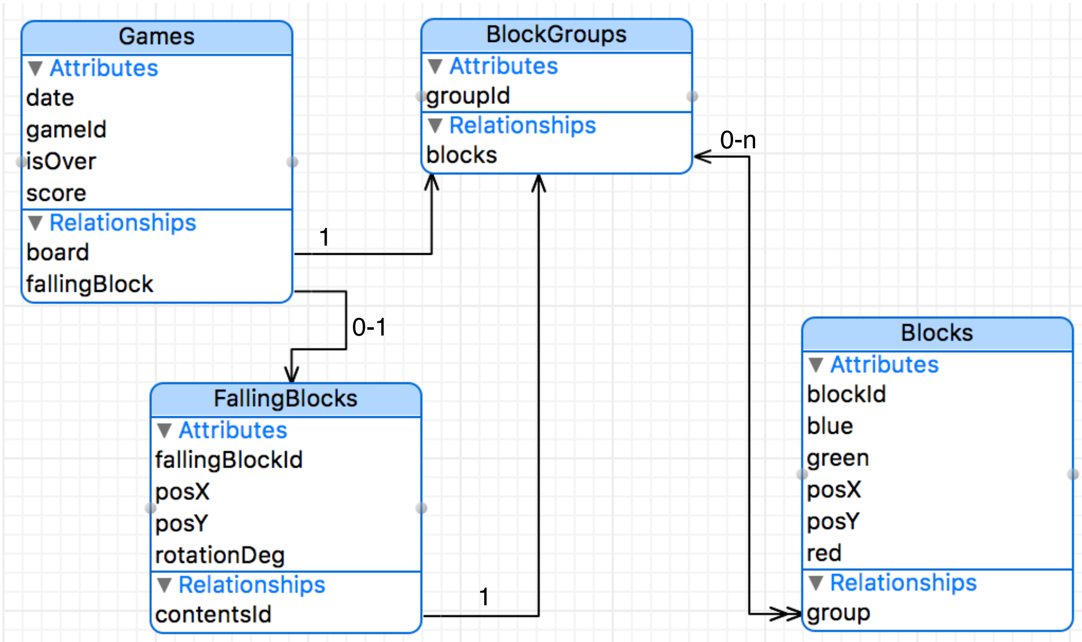

Figure 3 UML OO-ER Diagram (Object Oriented Database Schema)

3. Tables

Subsections within this section identify the database tables used in the implementation of this project.

3.1. Table *BlockGroups*

BlockGroups:

groupId

The *BlockGroups* table stores tuples containing a *groupId* attribute as an *Integer* that is auto-incremented and used as a Super Key. The *groupId* Super Key is used as a Foreign Key in other tables to provide a one-to-many relationship between *BlockGroups* and *Blocks* i.e. each *Blocks* tuple is related to exactly one *BlockGroups* tuple, but a single *BlockGroups tuple* may be related to any number of *Blocks* tuples.

```
## BlockGroups
cursor.executescript(""" CREATE TABLE IF NOT EXISTS BlockGroups(
   groupId INTEGER PRIMARY KEY AUTOINCREMENT); """)
```

3.2. Table *Blocks*

Blocks:

blockId	posX	posY	groupId	red	green	blue

A "block" represents a single square shape within a Tetris game. The *Blocks* table stores the attributes of every "block". Each tuple in the *Blocks* table stores a unique *blockId as* an Integer that is auto-incremented and used as a Super Key. Other attributes are the *posX* and *posY* coordinates of the "block's" position with respect to a *BlockGroup*, a *groupId* used as a Foreign Key to identify a *BlockGroup*, and the *red*, *green*, and *blue* color component intensities of the block.

```
## Blocks
cursor.executescript(""" CREATE TABLE IF NOT EXISTS Blocks(
   blockId INTEGER PRIMARY KEY AUTOINCREMENT,
   posX INT,
   posY INT,
   groupId INT,
   red INT,
   green INT,
   blue INT,
   FOREIGN KEY(groupId) REFERENCES BlockGroups(groupId)); """)
```

3.3. Table *FallingBlocks*

FallingBlocks:

<u>fallingBlockId</u>	posX	posY	angleDeg	contentsId

Each "Falling Block" is conceptually a collection of four blocks that move and rotate together partially under user control. The *FallingBlocks* table stores tuples with the following attributes: *fallingBlockId*, *posX*, *posY*, *angleDeg*, *contentsId*. The *fallingBlockId* attribute is an Integer that is auto-incremented and used as a Super Key. The *posX* and *posY* attributes store the current position of the falling block with respect to a game board. The *angleDeg* attribute stores the current rotation of the falling block with respect to a game board. The *contentsId* attribute is a Foreign Key that identifies which *BlockGroups* tuple relates to all of the "blocks" that are conceptually "within" the "Falling Block".

```
## Falling Blocks
cursor.executescript(""" CREATE TABLE IF NOT EXISTS FallingBlocks(
   fallingBlockId INTEGER PRIMARY KEY AUTOINCREMENT,
   posX INT,
   posY INT,
   angleDeg INT,
   contentsId INT,
   FOREIGN KEY(contentsId) REFERENCES
   BlockGroups(groupId)); """)
```

When the game logic creates a new "Falling Block", a single transaction is used to perform the following steps as an atomic group:

1) insert a new *BlockGroups* tuple in the *BlockGroups* table.
2) insert a new *FallingBlocks* tuple into the *FallingBlocks* table and relate the new *FallingBlocks* tuple to the new *BlockGroups* tuple via the *FallingBlocks* tuple's *contentsId* attribute.

Insert four *Blocks* tuples (in a configuration randomly selected from 7 possible configurations shown in

3) Figure 4) into the *Blocks* table and relate each new *Blocks* tuple to the new block group via the *Blocks* tuples' *groupId* attribute. A random color is generated and the four new *Blocks* tuples' *red*, *green*, and *blue* attributes are respectively set to the generated color's red, green, and blue component intensities.

The following code shows the implementation of the transaction to create a new Falling Block :

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")


## Create BlockGroup to store falling block's contents
self.cursor.execute("""INSERT INTO BlockGroups VALUES(NULL);""")
```

```
self.contentsId = self.cursor.lastrowid

## Insert FallingBlock
self.cursor.execute("""INSERT INTO FallingBlocks(
      posX, posY, angleDeg, contentsId) VALUES(?,?,?,?)""",
      (x, y, 0, self.contentsId))
self.load(self.cursor.lastrowid)

## Populate the content group with Blocks in a randomly selected
# arrangement and color
r = random.randint(128,255)
g = random.randint(128,255)
b = random.randint(128,255)
randomIndex = random.randint(0, len(standard_block_arrangements)-1)
arrangement = standard_block_arrangements[randomIndex]
for pos in arrangement:
   self.cursor.execute("""INSERT INTO Blocks(
         posX, posY, groupId, red, green, blue)
         VALUES(?,?,?,?,?,?)""",
         (pos[0], pos[1], self.contentsId, r, g, b))

DBTetris.connection.commit()                           ## <-- Commit
```

Figure 4 shows the available configurations of "blocks" conceptually "within" a "Falling Block".

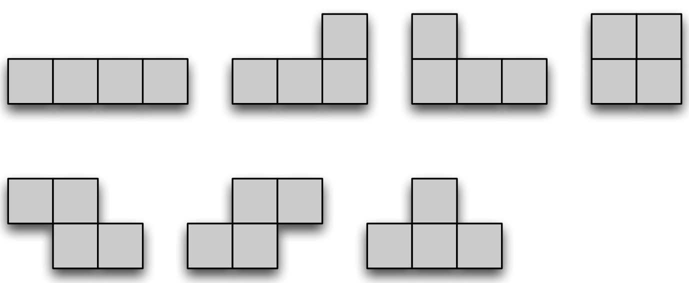

Figure 4 Available configurations of "blocks" conceptually "within" each "Falling Block"

3.4. Table *Games*

Games:

<u>gameId</u>	date	score	isOver	boardId	fallingBlockId

The *Games* table stores tuples with the following attributes: *gameId*, *date*, *score*, *isOver, boardId,* and *fallingBlockId.* The *gameId* attribute is an Integer that is auto-incremented and used as a Super Key. The *date* each game started is stored as Text. Each game's *score* is stored as an Integer. Whether the game is over or not is conceptually a Boolean, but it is stored as an Integer attribute, *isOver*. (0 means not over, every other value means the game is over.) The *boardId* attribute is a Foreign Key that identifies the *BlockGroups* tuple related to all of the blocks conceptually "in the game board". The *fallingBlockId* attribute is a Foreign Key that identifies the current *FallingBlocks* tuple or *Null* if there is no current falling block.

```
## Games
cursor.executescript(""" CREATE TABLE IF NOT EXISTS Games(
   gameId INTEGER PRIMARY KEY AUTOINCREMENT,
   date TEXT,
   score INT,
   isOver INT,
   boardId INT,
   fallingBlockId INT,
   FOREIGN KEY(boardId) REFERENCES BlockGroups(groupId),
   FOREIGN KEY(fallingBlockId) REFERENCES
      FallingBlocks(fallingBlockId)); """)
```

3.5. Analysis of Relations

The database for the Tetris game is in 5th Normal Form.

1NF) All attributes in all database tables are atomic.

2NF) Each database table has exactly one prime/key attribute. Therefore no non-prime attribute can have a dependency on a subset of a key.

3NF) There are no transitive dependencies in any tables.

BCNF) There is no redundancy based on functional dependence in any table.

4NF) There are no multi-value dependencies in any table.

5NF) Every non-trivial join dependency in every table is implied by the candidate keys in the tables.

4. Transactions

Subsections of this section identify all of the database transactions used to implement this project. Most transactions are implemented in the *DBTetrisGame* class/file, but a transaction related to creating a new falling block is implemented in the *DBTetrisFallingBlock* class/file.

4.1. *DBTetrisGame*: Create Game and Board

This transaction creates a new empty game board by inserting a tuple in the *BlockGroups* table and then creates a new game by inserting a tuple into the *Games* table. The new *Games* tuple is initialized with the *date* = current date, *score* = 0, *isOver* = 0, and *boardId* = groupId of the last tuple inserted into *BlockGroups*.

This transaction is invoked whenever the user launches the game application if and only if the database does not contain any *Games* tuples that have attribute *isOver* = 0. If the database contains one or more *Games* tuples that have attribute *isOver* = 0, the last (most recent) game in the database that is not over is resumed rather than create a new game.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
## Create BlockGroup to store board
DBTetris.cursor.execute("""INSERT INTO BlockGroups VALUES(NULL);""")
self.boardId = DBTetris.cursor.lastrowid

gameTuple = (str(date.today()), 0, 0, self.boardId)
## Create Game
DBTetris.cursor.execute("""INSERT INTO Games
      (date, score, isOver, boardId)
      VALUES(?,?,?,?)""", gameTuple)
self.ownId = DBTetris.cursor.lastrowid

DBTetris.connection.commit()                       ## <-- Commit
```

4.2. *DBTetrisGame*: Remove All Blocks in Complete Rows within the Game Board and Update the Game's Score

Within this transaction, it is possible for many tuples in the *Blocks* table to be modified. This transaction removes at most one row of blocks from the game board per invocation.

This transaction uses a native code loop to detect game board rows that are completely filled with blocks. A native code loop is needed because the order full rows are detected matters. If a full row is detected, the *Blocks* tuples for blocks that comprise the full row are deleted.

Then, an SQL statement decrements the value of the *posY* attribute of every block higher than the deleted row within the game board. The effect is to lower blocks above a removed row of blocks in accordance with Tetris game logic. This could have been accomplished in a native code loop, but SQL is well suited to this type of transaction, order doesn't matter, and SQL may provide more optimal database access order than a naively coded native code loop.

Finally, the *score* attribute of the game's tuple within the *Games* table is increased.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
for i in xrange(GAME_BOARD_HEIGHT-1, 0-1, -1):
   DBTetris.cursor.execute("""SELECT count(*) FROM Blocks
         WHERE groupId = ? AND posY = ? """, (self.boardId, i))
   count = DBTetris.cursor.fetchone()
   if GAME_BOARD_WIDTH <= count[0]:
      ## Remove row
      DBTetris.cursor.execute("""DELETE FROM Blocks
            WHERE groupId = ? AND posY = ? """, (self.boardId, i))
      DBTetris.cursor.execute("""UPDATE Blocks SET posY = posY - 1
            WHERE groupId = ? AND posY > ?;""", (self.boardId, i))
      DBTetris.cursor.execute("""UPDATE Games SET score = score + 1
            WHERE gameId = ?; """, (self.ownId,))
      DBTetris.connection.commit()               ## <-- Commit
      return ## Early Exit from loop!

DBTetris.connection.commit()                     ## <-- Commit
```

4.3. *DBTetrisGame*: Update Falling Block

The current Falling Block is moved down by one row in the game board. If the movement does not cause the Falling Block to leave the game board or collide with any blocks already in the game board, the transaction is committed.

If the movement does make the Falling Block leave the game board or collide with any blocks already in the game board, the movement transaction is rolled back. Then, a new transaction is started to perform the following operations as an atomic unit:

1) The blocks that are part of the Falling Block's block group are added to the game board by changing each affected tuples' *groupId* (Foreign Key) attribute to match the *groupId* (Super Key) attribute of the block group that conceptually "contains" the blocks on the game board.
2) The game tuple's *fallingBlockId* (Foreign Key) attribute is set to *Null* to disassociate the Falling Block from the current game.
3) The Falling Block's tuple is deleted from the *FallingBlocks* table.

The new transaction is committed leaving the game is a state where there is no current Falling Block. This condition is asynchronously detected by a separate co-routine/thread, and that thread uses the "Add Falling Block" transaction described in Section 4.4 to add a new Falling Block at the top of the game board.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
```

```
## Move the Falling Block down (may be rolled back)
DBTetris.cursor.execute("""UPDATE FallingBlocks
      SET posY = posY - 1 WHERE fallingBlockId = ?""",
      (self._fallingBlock.ownId,))

## Rollback if transaction produces collision with board sides
if (not self.doesFallingBlockFit() or\
      self.doesFallingBlockCollide()) and not self.getIsOver():

   DBTetris.connection.rollback() ## <--- Rollback

   DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
   ## Move falling block's content blocks to board
   blocks, x, y, r = self.getFallingBlocksPosistionAndRotation()
   blocks = self.getTransformedBlockPositions(x, y, r, blocks)

   for block in blocks:
      boardX = block[1]
      boardY = block[2]
      DBTetris.cursor.execute("""UPDATE Blocks SET groupId = ?,
            posX = ?, posY = ? WHERE blockId = ?;""",
            (self.boardId, boardX, boardY, block[0],))

   ## Remove relation to current falling block
   DBTetris.cursor.execute("""UPDATE Games
         SET fallingBlockId = NULL
         WHERE gameId = ? """, (self.ownId,))

   ## Delete the current falling block
   DBTetris.cursor.execute("""DELETE FROM FallingBlocks
         WHERE fallingBlockId = ? """, (self._fallingBlock.ownId,))

   ## There is no falling block anymore
   self._fallingBlock = None

   DBTetris.connection.commit()                 ## <-- Commit

else:
   ## No collision: Commit move
   DBTetris.connection.commit()                 ## <-- Commit
```

4.4. *DBTetrisGame*: Add Falling Block

This transaction starts by creating a new instance of the *FallingBlock* class which has the side effect of invoking the “Create Falling Block” transaction described in Section 4.5. The created Falling Block is related to the current game by updating the game tuple’s *fallingBlockId* (Foreign Key) attribute to match the Falling Block’s *fallingBlockId* (Super Key) attribute value.

```
self._fallingBlock = FallingBlock(DBTetris.cursor,
```

```
      GAME_BOARD_WIDTH // 2, GAME_BOARD_HEIGHT)
DBTetris.cursor.execute("""UPDATE Games SET fallingBlockId = ?
      WHERE gameId = ?""",
      (self._fallingBlock.ownId, self.ownId))
```

4.5. *DBTetrisFallingBlock*: Create Falling Block

This is the only transaction implemented in the *DBTetrisFallingBlock* class. See Section 3.3, "Table FallingBlocks" for details about the transaction to create a new "Falling Block".

4.6. *DBTetrisGame*: Move Falling Block Left

The "Move Falling Block Left" transaction is similar to the "Move Falling Block Right", "Rotate Falling Block Clockwise" and "Rotate Falling Block Counterclockwise" transactions. In each case, the attributes of the current Falling Block's tuple are modified and then evaluated using game logic to determine whether the modifications should be committed or rolled back. This approach is "optimistic" because changes are seldom rolled back.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
DBTetris.cursor.execute("""UPDATE FallingBlocks SET
      posX = posX - 1
      WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
if (not self.doesFallingBlockFit() or\
      self.doesFallingBlockCollide()) and not self.getIsOver():
   ## Lateral movement and rotation are not allowed to
   # cause collisions or cause blocks to leave the
   # board/play area, so rollback the change
   DBTetris.connection.rollback()              ## <-- Rollback
else:
   DBTetris.connection.commit()                ## <-- Commit
```

4.7. *DBTetrisGame*: Move Falling Block Right

See Section 4.6 for a description of this transaction's "optimistic" changes to the current Falling Block's attributes followed by either commit or rollback.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
DBTetris.cursor.execute("""UPDATE FallingBlocks SET
      posX = posX + 1
      WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
if (not self.doesFallingBlockFit() or\
      self.doesFallingBlockCollide()) and not self.getIsOver():
   ## Lateral movement and rotation are not allowed to
   # cause collisions or cause blocks to leave the
   # board/play area, so rollback the change
   DBTetris.connection.rollback()              ## <-- Rollback
else:
   DBTetris.connection.commit()                ## <-- Commit
```

4.8. *DBTetrisGame*: Rotate Falling Block Counterclockwise

See Section 4.6 for a description of this transaction's "optimistic" changes to the current Falling Block's attributes followed by either commit or rollback.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
DBTetris.cursor.execute("""UPDATE FallingBlocks SET
      angleDeg = angleDeg + 90
      WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
if (not self.doesFallingBlockFit() or\
       self.doesFallingBlockCollide()) and not self.getIsOver():
   ## Lateral movement and rotation are not allowed to
   # cause collisions or cause blocks to leave the
   # board/play area, so rollback the change
   DBTetris.connection.rollback()             ## <-- Rollback
else:
   DBTetris.connection.commit()               ## <-- Commit
```

4.9. *DBTetrisGame*: Rotate Falling Block Clockwise

See Section 4.6 for a description of this transaction's "optimistic" changes to the current Falling Block's attributes followed by either commit or rollback.

```
DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
DBTetris.cursor.execute("""UPDATE FallingBlocks SET
      angleDeg = angleDeg - 90
      WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
if (not self.doesFallingBlockFit() or\
     self.doesFallingBlockCollide()) and not self.getIsOver():
   ## Lateral movement and rotation are not allowed to
   # cause collisions or cause blocks to leave the
   # board/play area, so rollback the change
   DBTetris.connection.rollback()             ## <-- Rollback
else:
   DBTetris.connection.commit()               ## <-- Commit
```

5. Discussion

The subsections of this section describe the database manager used, aspects of the design for this project, alternative designs, concurrency of transactions, and performance.

5.1. SQLite Version: 3.13.0

The SQLite 3.13.0 database manager supports multiple mechanisms for managing concurrent transactions from multiple threads within a single process. Online documentation is available at https://www.sqlite.org/lockingv3.html. The mechanism all allow atomic commits of transactions involving multiple database tables and multiple database files. Locking and concurrency control make SQLite "ACID" (Atomic, Consistent, Isolated, and Durable) compliant.

By default, SQLite operates in *autocommit* mode. In *autocommit* mode, all changes to the database are committed as soon as all operations associated with the current database connection complete.

The SQL command "BEGIN TRANSACTION" is used to take SQLite out of *autocommit* mode. **Note**: the BEGIN command does not acquire any locks on the database. After a BEGIN command, a SHARED lock will be acquired when the first SELECT statement is executed. A RESERVED lock will be acquired when the first INSERT, UPDATE, or DELETE statement is executed. No EXCLUSIVE lock is acquired until either the memory cache fills up and must be spilled to disk or until the transaction commits. In this way, the system delays blocking read access to the file until the last possible moment.

The SQL command "COMMIT" does not actually commit the changes to disk. It just turns *autocommit* back on. Then, at the conclusion of the command, the regular *autocommit* logic takes over and causes the actual commit to disk to occur. The SQL command "ROLLBACK" also operates by turning *autocommit* back on, but it also sets a flag that tells the *autocommit* logic to rollback rather than commit.

If multiple commands are being executed against the same SQLite database connection at the same time, the *autocommit* is deferred until the very last command completes. For example, if a SELECT statement is being executed, the execution of the command will pause as each row of the result is returned. During this pause other INSERT, UPDATE, or DELETE commands can be executed against other tables in the database. But none of these changes will commit until the original SELECT statement finishes.

5.2. Design Analysis

The design of the relational database for storing Tetris game state is not complex. I previously created a similar game using the iOS Cocoa Touch frameworks with an Object Oriented database. The significant difference between the Object Oriented database schema and the relational schema hinges on the representation of “to-many” relationships. The relational schema includes a table, *BlockGroups*, that stores only one attribute.

BlockGroups:

groupId

Having a table for *BlockGroups’ groupId* enables the use of the *groupId* as a Foreign Key in other tables. For example, tuples in the *Games* table used *boardId* as a Foreign Key relation to *BlockGroups’ groupId* (Super Key). The tuples in the *FallingBlocks* table use *contentsId* as a Foreign Key relation to *BlockGroups’ groupId* (Super Key). As a result, given a *FallingBlocks* tuple, it is possible to efficiently find the *BlockGroups’* tuples related by *groupId = contentsId*. In other words, it is possible to find all of the “blocks” that compose a “Falling Block”. The *Blocks* table uses *groupId* as a Foreign Key relation to *BlockGroups’ groupId* (Super Key). It is possible to efficiently find all of the “blocks” that compose a “board” by *groupId = boardId*. The *BlockGroups* table is the mechanism for supporting “to-many” relations between the game “board” and “blocks” and between the “falling block”, and “blocks”.

In contrast, the Object Oriented database schema supports “to-many” relationships without any “extra” table(s) for the purpose. However, the Object Oriented schema imposes a different constraint: all relationships must be “reflexive/bi-directional”. In other words, if “board” has a “to-many” relation to “block”, then “block” must have a corresponding “to-one” or “to-many” relationship with “board”. This is problematic when more than one class has a relationship with “block” a shown in Figure 5, “An Object Oriented ER Diagram showing Inheritance Relationships as well as “to-one” and “to-many” relationships”.

The *BlockGroup* class in Figure 5 has a “one-to-one” relationship with the *Game* class. The *AnimatedBlockGroup* class also has a “one-to-one” relationship with the *Game* class in spite of also inheriting the relationship from *BlockGroup*. In other words, each *AnimatedBlockGroup* instance ends up with two relationships to the same *Game* instance even though only one is strictly need. The redundancy is present to satisfy automatic database validations that include run time type information. *Game*’s *fallingBlock* relation is to type *AnimatedBlockGroup*, and *Game*’s *board* relation is to type *BlockGroup*. Because they have different types, they cannot be consolidated into one relation.

Using SQL statements instead of objects mapped to an object oriented database roughly tripled the amount of code needed to implement the game.

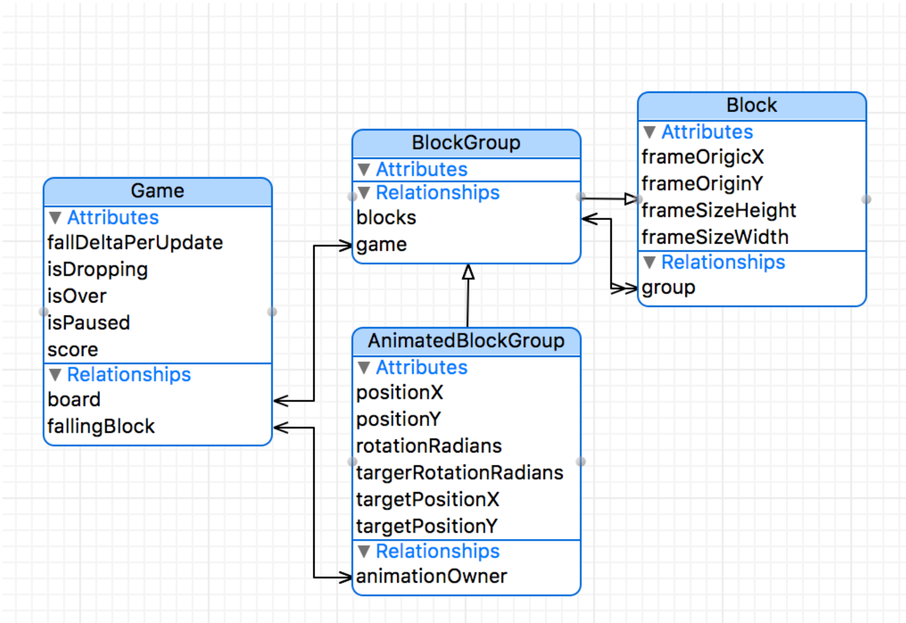

Figure 5 An Object Oriented ER Diagram showing Inheritance Relationships as well as "to-one" and "to-many" Relationships

5.3. Concurrency of Transactions

Within the implementation of this project, database transactions occur within five separate executions threads:

1) The "main" thread handles "user input" and invokes the "*DBTetrisGame*: Move Falling Block Left", "*DBTetrisGame*: Move Falling Block Right", "*DBTetrisGame*: Rotate Falling Block Counterclockwise", and "*DBTetrisGame*: Rotate Falling Block Clockwise" transactions. **Note**: these transactions are invoked whenever the user presses the relevant key. Therefore, these transactions are inherently asynchronous to transactions occurring on other threads of execution.
2) The **GameFallingBlockAction** thread is encapsulated in a Cocos2D ***Action***. ***Actions*** are "co-routines" that the Cocos2D library parallelizes in separate threads when possible. The **GameFallingBlockAction** invokes the "*DBTetrisGame*: Update Falling Block" transaction approximately 25 times per second.
3) The **GameCompleteRowsAction** thread is encapsulated in a Cocos2D ***Action*** and invokes the "*DBTetrisGame*: Remove All Blocks in Complete Rows within the Game Board and Update the Game's Score" transaction approximately 30 times per second.
4) The **GameScoreAction** thread is encapsulated in a Cocos2D ***Action*** and queries the database for the current game's *score* approximately 60 times per second. This thread also invokes the "*DBTetrisGame*: Add Falling Block" transaction whenever the thread detects that the game has no current Falling Block.

5) The **GameAction** thread is encapsulated in a Cocos2D ***Action*** and queries the database approximately 60 time per second to obtain the current game's board state and Falling Block state. This thread updates the display to present the game state to a user.

It is crucial for the implementation that certain transactions occur atomically because interleaving concurrent transactions can corrupt the game state stored in the database. Even some multi-step SELECT queries need to be protected as atomic transactions to avoid the possibility that another concurrent transaction modifies tables between one SELECT and the next in the multi-step query.

All game state is stored in the database. A game can be exited at any time and resumed from the state encapsulated by the database at a later time.

Note: Using multiple threads in this project adds complexity that is not suitable or justifiable for a "real" game as trivial as the one produced by this project. The multiple threads are only used to demonstrate concurrent transactions for the sake of meeting the presumed but never explicitly stated or documented course/project goals.

Note: It is possible to run multiple game instances simultaneously using the same database. This is yet another form of concurrency that is supported. All concurrent game instances display the same game state. However, there is typically a multi-second delay between when one game instance performs a database transaction and the others read the result. The delay is the result of infrequent "checkpoints". The data on disk is typically not frequently updated to match the in-memory database content. Interestingly, exiting (or crashing) a game instance results in a prompt checkpoint, and the other game instances almost instantly display the game state captured in the checkpoint.

5.4. Performance

The SQLite database used for this project provides excellent performance. It is an "in-memory" database that periodically creates checkpoints in a persistent file on disk. The database for this project is small even when storing the boards, scores, and blocks for every instance of the game ever played. The entire database and all internal indexes fit easily in memory. Furthermore, the test system used has a fast solid-state drive that means disk performance is unlikely to limit database performance.

Even though SQLite is fast, updating game state via multiple SQL transactions achieves approximately 1/1000th the performance of direct instance variable access. Storing Tetris games in an SQL database has the advantage that the games can be exited and resumed later without any loss of game state/data. Finding the score of the highest scoring game in the database is a simple SQL query. In fact, obtaining the scores and dates of the top 10 high scoring games is a trivial transaction and could be used to present a leaderboard/high score report.

BEGIN PYTHON CODE LISTINGS

Erik M. Buck

July 18, 2017

```python
from datetime import date, datetime
import sqlite3

##
connection = sqlite3.connect('tetris.db', isolation_level="EXCLUSIVE")
connection.execute('PRAGMA foreign_keys')
cursor = connection.cursor()

def create_tables(cursor):
    """ """

    with connection:

        cursor.execute('SELECT SQLITE_VERSION()')
        data = cursor.fetchone()
        print "SQLite version: %s" % data

        ## BlockGroups
        cursor.executescript("""
                    CREATE TABLE IF NOT EXISTS BlockGroups(
                       groupId INTEGER PRIMARY KEY AUTOINCREMENT);
                    """)

        ## Blocks
        cursor.executescript("""
                    CREATE TABLE IF NOT EXISTS Blocks(
                       blockId INTEGER PRIMARY KEY AUTOINCREMENT,
                       posX INT,
                       posY INT,
                       groupId INT,
                       red INT,
                       green INT,
                       blue INT,
                       FOREIGN KEY(groupId) REFERENCES BlockGroups(groupId));
                    """)

        ## Falling Blocks
        cursor.executescript("""
                    CREATE TABLE IF NOT EXISTS FallingBlocks(
                       fallingBlockId INTEGER PRIMARY KEY AUTOINCREMENT,
                       posX INT,
                       posY INT,
                       angleDeg INT,
                       contentsId INT,
                       FOREIGN KEY(contentsId) REFERENCES BlockGroups(groupId));
                    """)


        ## Games
        cursor.executescript("""
                    CREATE TABLE IF NOT EXISTS Games(
                       gameId INTEGER PRIMARY KEY AUTOINCREMENT,
                       date TEXT,
                       score INT,
                       isOver INT,
                       boardId INT,
```

```
                        fallingBlockId INT,
                        FOREIGN KEY(boardId) REFERENCES BlockGroups(groupId),
                        FOREIGN KEY(fallingBlockId) REFERENCES FallingBlocks
                           (fallingBlockId));
                     """)

######################################################################
######################################################################
if __name__ == "__main__":
   print 'Please start the game from Game.py'
```

```python
from datetime import date, datetime
import sqlite3
import random
import DBTetris


standard_block_arrangements = (
   ((-1, 0), (0, 0), (1,  0), ( 2, 0)), ## Horizontal bar
   (( 0, 1), (1, 1), (0,  0), ( 1, 0)), ## Box
   (( 0, 1), (1, 1), (1,  0), ( 2, 0)), ## Z one
   (( 2, 1), (1, 1), (1,  0), ( 0, 0)), ## Z two
   ((-1, 0), (0, 0), (1,  0), ( 0, 1)), ## Tee
   ((-1, 0), (0, 0), (1,  0), ( 1, 1)), ## L one
   ((-1, 0), (0, 0), (1,  0), (-1, 1)), ## L two
)


######################################################################
######################################################################
class DBTetrisFallingBlock(object):
   """ """

   ###################################################################
   def __init__(self, cursor, x, y, ownId = None):
      """ """
      self.cursor = cursor

      if None == ownId:
         DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")

         ## Create BlockGroup to store falling block's contents
         self.cursor.execute("""INSERT INTO BlockGroups VALUES(NULL);""")
         self.contentsId = self.cursor.lastrowid

         ## Insert FallingBlock
         self.cursor.execute("""INSERT INTO FallingBlocks(
            posX, posY, angleDeg, contentsId) VALUES(?,?,?,?)""",
            (x, y, 0, self.contentsId))
         self.load(self.cursor.lastrowid)

         ## Populate the content group with Blocks in a randomly selected
         # arrangement and color
         r = random.randint(128,255)
         g = random.randint(128,255)
         b = random.randint(128,255)
         randomIndex = random.randint(0, len(standard_block_arrangements)-1)
         arrangement = standard_block_arrangements[randomIndex]
         for pos in arrangement:
            self.cursor.execute("""INSERT INTO Blocks(
                  posX, posY, groupId, red, green, blue)
                  VALUES(?,?,?,?,?,?)""",
                  (pos[0], pos[1], self.contentsId, r, g, b))

         DBTetris.connection.commit()               ## <-- Commit

      else:
```

```
        self.load(ownId)

    ######################################################################
    def load(self, ownId):
        """ """
        self.ownId = ownId
        self.cursor.execute("""SELECT contentsId FROM FallingBlocks
            WHERE fallingBlockId = ?""", (self.ownId,))
        self.contentsId = list(self.cursor.fetchone())[0]


######################################################################
######################################################################
if __name__ == "__main__":
    print 'Please start the game from Game.py'
```

```
from datetime import date, datetime
import DBTetris
from DBTetrisFallingBlock import DBTetrisFallingBlock as FallingBlock
import sqlite3

GAME_BOARD_WIDTH = 11
GAME_BOARD_HEIGHT = 22


class DBTetrisGame(object):
   """ """

   ###################################################################
   def __init__(self, ownId = None):
      """ This initializer fetches the tuple for the most recent
          game that is not over. If every game in the database is
          over, a new game is created as a transaction involving
          multiple steps. """
      self._fallingBlock = None
      self.__isGameOver = False

      if None == ownId:
         ## See if there is an ongoing Game to be continued
         DBTetris.cursor.execute("""SELECT * FROM Games WHERE isOver = 0;""")
         games = DBTetris.cursor.fetchall()
         if 0 < len(games):
            # Load the game state from the last selected tuple.
            gameTuple = games[-1] ## Grab the last/most recent game
            print 'Resuming Game from ' + gameTuple[1]
            self.ownId = gameTuple[0]
            self.boardId = gameTuple[4]
            self._fallingBlock = FallingBlock(DBTetris.cursor,
               0, 0, gameTuple[5])

         else:
            DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
            ## Create BlockGroup to store board
            DBTetris.cursor.execute("""INSERT INTO BlockGroups VALUES(NULL);"""
               )
            self.boardId = DBTetris.cursor.lastrowid

            gameTuple = (str(date.today()), 0, 0, self.boardId)
            ## Create Game
            DBTetris.cursor.execute("""INSERT INTO Games
               (date, score, isOver, boardId)
               VALUES(?,?,?,?)""", gameTuple)
            self.ownId = DBTetris.cursor.lastrowid

            DBTetris.connection.commit()                ## <-- Commit

      else:
         self.ownId = ownId
         DBTetris.cursor.execute("""SELECT ? FROM Games""", (self.ownId,))
         gameTuple = DBTetris.cursor.fetchone()
         self.boardId = gameTuple[4]
         self._fallingBlock = FallingBlock(DBTetris.cursor,
```

```
            0, 0, gameTuple[5])

    ###################################################################
    def addFallingBlock(self):
        """ """
        self._fallingBlock = FallingBlock(DBTetris.cursor,
              GAME_BOARD_WIDTH//2, GAME_BOARD_HEIGHT)
        DBTetris.cursor.execute("""UPDATE Games SET fallingBlockId = ?
           WHERE gameId = ?""",
           (self._fallingBlock.ownId, self.ownId))

    ###################################################################
    def setGameOver(self):
        """ """
        DBTetris.cursor.execute("""UPDATE Games SET isOver = 1
              WHERE gameId = ?; """, (self.ownId,))


    ###################################################################
    def getIsOver(self):
        """ """
        ## Get the all-time high score
        DBTetris.cursor.execute("""SELECT isOver FROM Games WHERE gameId = ?""",
           (self.ownId,))
        return DBTetris.cursor.fetchone()[0]

    ###################################################################
    def getHighScore(self):
        """ """
        ## Get the all-time high score
        DBTetris.cursor.execute("""SELECT MAX(score) FROM Games;""")
        return DBTetris.cursor.fetchone()[0]

    ###################################################################
    def getHighScoreDate(self):
        """ """
        DBTetris.cursor.execute("""SELECT date FROM Games
              WHERE score = (SELECT MAX(score) FROM Games);""")
        return DBTetris.cursor.fetchone()[0]

    ###################################################################
    def getScore(self):
        """ """
        DBTetris.cursor.execute("""SELECT score FROM Games WHERE gameId = ?""",
              (self.ownId,))
        score = DBTetris.cursor.fetchone()
        return score[0]

    ###################################################################
    def getFallingBlock(self):
        """ """
        if None != self._fallingBlock and None == self._fallingBlock.ownId:
           self._fallingBlock = None
        return self._fallingBlock

    ###################################################################
```

```
    def getTransformedBlockPositions(self, offsetX, offsetY, rotation, blocks):
        """ """
        result = []

        ## Normalize rotation to positive angles < 360
        rotation = rotation % 360
        if 0 > rotation:
            rotation += 360

        if rotation >= 270:
            for block in blocks:
                blockAsList = list(block)
                x = blockAsList[1]
                y = blockAsList[2]
                blockAsList[1] = x *  0 + y * 1 + offsetX
                blockAsList[2] = x * -1 + y * 0 + offsetY
                result.append(blockAsList)
        elif rotation >= 180:
            for block in blocks:
                blockAsList = list(block)
                x = blockAsList[1]
                y = blockAsList[2]
                blockAsList[1] = x * -1 + y * 0 + offsetX
                blockAsList[2] = x * 0 + y * -1 + offsetY
                result.append(blockAsList)
        elif rotation >= 90:
            for block in blocks:
                blockAsList = list(block)
                x = blockAsList[1]
                y = blockAsList[2]
                blockAsList[1] = x * 0 + y * -1 + offsetX
                blockAsList[2] = x * 1 + y *  0 + offsetY
                result.append(blockAsList)
        else:
            for block in blocks:
                blockAsList = list(block)
                blockAsList[1] = blockAsList[1] + offsetX
                blockAsList[2] = blockAsList[2] + offsetY
                result.append(blockAsList)

        return result

    ####################################################################
    def doesFallingBlockCollide(self):
        DBTetris.cursor.execute("""SELECT * FROM Blocks
            WHERE groupId = ? """, (self.boardId,))
        boardTuples = DBTetris.cursor.fetchall()
        maxY = 0
        blocksPosAndRot = self.getFallingBlocksPosistionAndRotation()
        if None != blocksPosAndRot:
            fallingTuples, x, y, r = blocksPosAndRot
            fallingTuples = self.getTransformedBlockPositions(x, y, r,
                fallingTuples)

            for fallingTuple in fallingTuples:
                fx = fallingTuple[1]
```

```
            fy = fallingTuple[2]
            maxY = max(maxY, fy)
            if 0 > fy:
               ## Collided with bottom of board
               return True

            for boardTuple in boardTuples:
               if fx == boardTuple[1] and fy == boardTuple[2]:
                  ## Collided with block already in board
                  ## If falling block is above top of board, the game is over
                  if maxY >= GAME_BOARD_HEIGHT:
                     self.setGameOver()

                  return True
      return False

   ####################################################################
   def doesFallingBlockFit(self):
      """ """
      blocksPosAndRot = self.getFallingBlocksPosistionAndRotation()
      if None != blocksPosAndRot:
         blocks, x, y, r = blocksPosAndRot
         blocks = self.getTransformedBlockPositions(x, y, r, blocks)
         for block in blocks:
            if block[1] < 0 or block[1] >= GAME_BOARD_WIDTH:
               ## Block is off left or right of game board
               return False

      return True

   ####################################################################
   def getBoardBlocks(self):
      """ """
      DBTetris.cursor.execute("""SELECT * FROM Blocks
            WHERE groupId = ? """, (self.boardId,))
      return DBTetris.cursor.fetchall()

   ####################################################################
   def getFallingBlocksPosistionAndRotation(self):
      """ """
      result = None

      if None != self._fallingBlock and None != self._fallingBlock.ownId:
         DBTetris.cursor.execute("""SELECT posX, posY, angleDeg
               FROM FallingBlocks
               WHERE fallingBlockId = ? """, (self._fallingBlock.ownId,))
         posAndRot = DBTetris.cursor.fetchone()
         if None == posAndRot:
            return None  ## EARLY EXIT

         x, y, r = posAndRot
         DBTetris.cursor.execute("""SELECT * FROM Blocks
               WHERE groupId = ? """, (self._fallingBlock.contentsId,))
         fallingBlocks = DBTetris.cursor.fetchall()
         result = (fallingBlocks, x, y, r)
```

```
        return result

    ###################################################################
    def moveFallingBlockLeft(self):
        """ """
        if None != self._fallingBlock:
            DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
            DBTetris.cursor.execute("""UPDATE FallingBlocks SET
                    posX = posX - 1
                    WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
            if (not self.doesFallingBlockFit() or\
                    self.doesFallingBlockCollide()) and not self.getIsOver():
                ## Lateral movement and rotation are not allowed to
                # cause collisions or cause blocks to leave the
                # board/play area, so rollback the change
                DBTetris.connection.rollback()              ## <-- Rollback
            else:
                DBTetris.connection.commit()                ## <-- Commit

    ###################################################################
    def moveFallingBlockRight(self):
        """ """
        if None != self._fallingBlock:
            DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
            DBTetris.cursor.execute("""UPDATE FallingBlocks SET
                    posX = posX + 1
                    WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
            if (not self.doesFallingBlockFit() or\
                    self.doesFallingBlockCollide()) and not self.getIsOver():
                ## Lateral movement and rotation are not allowed to
                # cause collisions or cause blocks to leave the
                # board/play area, so rollback the change
                DBTetris.connection.rollback()              ## <-- Rollback
            else:
                DBTetris.connection.commit()                ## <-- Commit

    ###################################################################
    def rotateFallingBlockCounterclockwise(self):
        """ """
        if None != self._fallingBlock:
            DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
            DBTetris.cursor.execute("""UPDATE FallingBlocks SET
                    angleDeg = angleDeg + 90
                    WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
            if (not self.doesFallingBlockFit() or\
                    self.doesFallingBlockCollide()) and not self.getIsOver():
                ## Lateral movement and rotation are not allowed to
                # cause collisions or cause blocks to leave the
                # board/play area, so rollback the change
                DBTetris.connection.rollback()              ## <-- Rollback
            else:
                DBTetris.connection.commit()                ## <-- Commit

    ###################################################################
    def rotateFallingBlockClockwise(self):
        """ """
```

```
        if None != self._fallingBlock:
           DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
           DBTetris.cursor.execute("""UPDATE FallingBlocks SET
                 angleDeg = angleDeg - 90
                 WHERE fallingBlockId = ?""", (self._fallingBlock.ownId,))
           if (not self.doesFallingBlockFit() or\
                 self.doesFallingBlockCollide()) and not self.getIsOver():
              ## Lateral movement and rotation are not allowed to
              # cause collisions or cause blocks to leave the
              # board/play area, so rollback the change
              DBTetris.connection.rollback()              ## <-- Rollback
           else:
              DBTetris.connection.commit()                ## <-- Commit

     ###################################################################
     def clearCompleteRowsAndUpdateScore(self):
        """ """
        DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
        for i in xrange(GAME_BOARD_HEIGHT-1, 0-1, -1):
           DBTetris.cursor.execute("""SELECT count(*) FROM Blocks
                 WHERE groupId = ? AND posY = ? """, (self.boardId, i))
           count = DBTetris.cursor.fetchone()
           if GAME_BOARD_WIDTH <= count[0]:
              ## Remove row
              DBTetris.cursor.execute("""DELETE FROM Blocks
                    WHERE groupId = ? AND posY = ? """, (self.boardId, i))
              DBTetris.cursor.execute("""UPDATE Blocks SET posY = posY - 1
                    WHERE groupId = ? AND posY > ?;""", (self.boardId, i))
              DBTetris.cursor.execute("""UPDATE Games SET score = score + 1
                    WHERE gameId = ?; """, (self.ownId,))
              DBTetris.connection.commit()                ## <-- Commit
              return

        DBTetris.connection.commit()                      ## <-- Commit

     ###################################################################
     def updateFallingBlock(self):
        """ """
        if None != self.getFallingBlock():
           DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
           ## Move the Falling Block down (may be rolled back)
           DBTetris.cursor.execute("""UPDATE FallingBlocks
                 SET posY = posY - 1 WHERE fallingBlockId = ?""",
                 (self._fallingBlock.ownId,))

           ## Rollback if transaction produces collision with board sides
           if (not self.doesFallingBlockFit() or\
                 self.doesFallingBlockCollide()) and not self.getIsOver():
              DBTetris.connection.rollback() ## <--- Rollback

              DBTetris.cursor.execute("""BEGIN EXCLUSIVE TRANSACTION""")
              ## Move falling block's content blocks to board
              blocks, x, y, r = self.getFallingBlocksPosistionAndRotation()
              blocks = self.getTransformedBlockPositions(x, y, r, blocks)

              for block in blocks:
```

```
                boardX = block[1]
                boardY = block[2]
                DBTetris.cursor.execute("""UPDATE Blocks SET groupId = ?,
                      posX = ?, posY = ? WHERE blockId = ?;""",
                      (self.boardId, boardX, boardY, block[0],))

            ## Remove relation to current falling block
            DBTetris.cursor.execute("""UPDATE Games
                  SET fallingBlockId = NULL
                  WHERE gameId = ? """, (self.ownId,))

            ## Delete the current falling block
            DBTetris.cursor.execute("""DELETE FROM FallingBlocks
                  WHERE fallingBlockId = ? """, (self._fallingBlock.ownId,))

            ## There is no falling block anymore
            self._fallingBlock = None

            DBTetris.connection.commit()              ## <-- Commit

         else:
            ## No collision: Commit move
            DBTetris.connection.commit()                 ## <-- Commit


DBTetris.create_tables(DBTetris.cursor)

######################################################################
######################################################################
if __name__ == "__main__":
   print 'Please start the game from Game.py'
```

```python
import cocos
import pyglet
import DBTetrisGame

BLOCK_WIDTH = 40

######################################################################
######################################################################
class GameBaseAction(cocos.actions.Action):
   """ Asynchronously ask the model to both clear a complete rows
       blocks in the game board and update. The operations are
       performed together as a single transaction.
   """

   def start(self):
      """ """
      self.current_update_number = 0

   def step(self, dt):
      """ """
      if not self.target.game_model.getIsOver():
         self.current_update_number += 1

######################################################################
######################################################################
class GameFallingBlockAction(GameBaseAction):
   """ Asynchronously ask the model to both clear a complete rows
       blocks in the game board and update. The operations are
       performed together as a single transaction.
   """

   def step(self, dt):
      """ """
      super( GameFallingBlockAction, self ).step(dt)
      if not self.target.game_model.getIsOver():
         self.current_update_number += 1
         time_divisor = self.target.getTimeDivisor()

         if 0 == (self.current_update_number % time_divisor):
            self.target.game_model.updateFallingBlock()

######################################################################
######################################################################
class GameCompleteRowsAction(GameBaseAction):
   """ Asynchronously ask the model to both clear a complete rows
       blocks in the game board and update. The operations are
       performed together as a single transaction.
   """

   def step(self, dt):
      """ """
      super( GameCompleteRowsAction, self ).step(dt)
      if not self.target.game_model.getIsOver():
         self.current_update_number += 1
         if 0 == self.current_update_number % 30:
            ## Clear any complete rows
```

```
            self.target.game_model.clearCompleteRowsAndUpdateScore()


####################################################################
####################################################################
class GameScoreAction(GameBaseAction):
   """ Asynchronously ask the model for the current score and current
       high score. Then update the display to match.
   """

   def step(self, dt):
      """ """
      if None == self.target.game_model.getFallingBlock():
         ## All complete rows have been removed so start new falling block
         self.target.game_model.addFallingBlock()
      self.target.updateScoreDisplay()

####################################################################
####################################################################
class GameAction(GameBaseAction):
   """
   """

   def step(self, dt):
      """ """
      super( GameAction, self ).step(dt)
      if self.target.game_model.getIsOver():
         self.target.presentGameOverNotice()
         return

      time_divisor = self.target.getTimeDivisor()
      if 0 == (self.current_update_number % time_divisor):
         self.target.synchronizeDisplayWithModel(time_divisor)

####################################################################
####################################################################
class Game(cocos.layer.ColorLayer):
   """
   """

   # Tell cocos that this layer is for handling input!
   is_event_handler = True

   ##################################################################
   def __init__(self):
      """ """
      self.game_model = DBTetrisGame.DBTetrisGame()
      self.keys_being_pressed = set()
      self.isDropping = False
      self.model_block_to_layer_map = {}
      self.updates_per_move = 25
      self.current_update_number = 0
      self.last_score = 0
      self.last_high_score = 0

      self.director_width = DBTetrisGame.GAME_BOARD_WIDTH * BLOCK_WIDTH
```

```
        self.director_height = DBTetrisGame.GAME_BOARD_HEIGHT * BLOCK_WIDTH
        cocos.director.director.init(self.director_width, self.director_height,
           caption = "DBTetris", fullscreen=False)

        super( Game, self ).__init__(64, 64, 64, 255)

        self.intro_scene = cocos.scene.Scene(self)
        self.score_label = cocos.text.Label(text="Score: 0",
              font_size = 24, position = (10, self.director_height - 40))
        self.add(self.score_label)
        self.high_score_label = cocos.text.Label(text="High: 0", font_size = 24,
              position = (self.director_width-10, self.director_height - 40),
              anchor_x = "right")
        self.add(self.high_score_label)
        self.game_over_label = None

        self.do(GameAction())
        self.do(GameScoreAction())
        self.do(GameCompleteRowsAction())
        self.do(GameFallingBlockAction())

    ###################################################################
    def on_key_press(self, key, modifiers):
        """ """
        self.keys_being_pressed.add(key)
        if pyglet.window.key.LEFT == key:
           self.game_model.moveFallingBlockLeft()
        elif pyglet.window.key.RIGHT == key:
           self.game_model.moveFallingBlockRight()
        elif pyglet.window.key.SPACE == key:
           self.isDropping = True
        elif pyglet.window.key.A == key:
           self.game_model.rotateFallingBlockCounterclockwise()
        elif pyglet.window.key.D == key:
           self.game_model.rotateFallingBlockClockwise()

    ###################################################################
    def presentGameOverNotice(self):
        """ """
        if None == self.game_over_label:
           self.game_over_label = cocos.layer.Layer()
           label = cocos.text.Label(text="GAME OVER",
              font_size = 54,
              position = (self.director_width//2, self.director_height//2),
              anchor_x = "center")
           self.game_over_label.add(label)

           date = self.game_model.getHighScoreDate()
           message = 'High Score Achived ' + date
           message_layer = cocos.text.Label(text=message,
              font_size = 20,
              position = (self.director_width//2, self.director_height//2 - 40),
              anchor_x = "center")
           self.game_over_label.add(message_layer)

           self.add(self.game_over_label)
```

```
        self.game_over_label.opacity = 0
        self.game_over_label.do(cocos.actions.RotateTo(-30, duration = 1))
        self.game_over_label.do(cocos.actions.FadeIn(1))

    ###################################################################
    def updateScoreDisplay(self):
        """ """
        score = self.game_model.getScore()
        if self.last_score != score:
            self.last_score = score
            self.score_label.element.text = "Score: "+str(self.last_score)

        high_score = self.game_model.getHighScore()
        if self.last_high_score != high_score:
            self.last_high_score = high_score
            self.high_score_label.element.text = "High: "+str(self.last_high_score
                )

    ###################################################################
    def getTimeDivisor(self):
        """ """
        time_divisor = max(1, self.updates_per_move)

        if self.isDropping:
            time_divisor = 1 ## Drop as fast as possible

        return time_divisor

    ###################################################################
    def synchronizeDisplayWithModel(self, time_divisor):
        """ """
        blocks = self.game_model.getBoardBlocks()

        # Build map by adding blocks that are still in the model
        new_model_block_to_layer_map = {}

        ## Update positions of layers to match the blocks on the board
        for block in blocks:
            newX = block[1] * BLOCK_WIDTH
            newY = block[2] * BLOCK_WIDTH
            if not block[0] in self.model_block_to_layer_map:
                ## Create layer to represent block
                newBlockLayer = cocos.layer.ColorLayer(
                      block[4], block[5], block[6], 255, BLOCK_WIDTH, BLOCK_WIDTH)
                newBlockLayer.position = (newX, newY)
                self.add(newBlockLayer)
                self.model_block_to_layer_map[block[0]] = newBlockLayer

            layer = self.model_block_to_layer_map[block[0]]
            layer.position = (newX, newY)
            new_model_block_to_layer_map[block[0]] = layer

        ## Update positions of layers to match the blocks that are falling
        blocksPosAndRot = self.game_model.getFallingBlocksPosistionAndRotation()
        if None != blocksPosAndRot:
            fallingBlocks, x,y, rotation = blocksPosAndRot
```

```
        fallingBlocks = self.game_model.getTransformedBlockPositions(
           x, y, rotation, fallingBlocks)

        ## Reposition each layer to match its corresponding block's position
        for block in fallingBlocks:
           newX = block[1] * BLOCK_WIDTH
           newY = block[2] * BLOCK_WIDTH
           if not block[0] in self.model_block_to_layer_map:
              ## Create a layer to represent block
              newBlockLayer = cocos.layer.ColorLayer(
                    block[4], block[5], block[6], 255, BLOCK_WIDTH,
                       BLOCK_WIDTH)
              newBlockLayer.position = (newX, newY)
              self.add(newBlockLayer)
              self.model_block_to_layer_map[block[0]] = newBlockLayer
              self.isDropping = False

           layer = self.model_block_to_layer_map[block[0]]
           layer.stop()
           layer.do(cocos.actions.MoveTo((newX, newY),
                 duration=1.0/60.0 * time_divisor))

           new_model_block_to_layer_map[block[0]] = layer

     ## Remove layers that no longer have corresponding blocks in the model
     # for each block in self.model_block_to_layer_map that isn't in
     # new_model_block_to_layer_map, remove the corresponding layer
     for id, layer in self.model_block_to_layer_map.iteritems():
        if not id in new_model_block_to_layer_map:
           layer.do(cocos.actions.FadeOut(0.5)+\
              cocos.actions.CallFuncS(cocos.layer.Layer.kill))

     self.model_block_to_layer_map = new_model_block_to_layer_map

  def run(self, host=None, port=None):
     """ """
     cocos.director.director.set_show_FPS(True)
     cocos.director.director.run (self.intro_scene)

######################################################################
######################################################################
if __name__ == "__main__":
  game = Game()
  game.run()
```

BEGIN CONTENT OF DATA TABLES

Erik M. Buck

July 18, 2017

```
SQLite version: 3.13.0
BlockGroups:
groupId
      1
      2
      3
      4
      5
[... Truncated for brevity ...]
     69
     70
     71
     72

Blocks:
blockId posX posY groupId red green blue
4       10   0    1       150 150   150
6       7    0    1       132 132   132
7       8    0    1       132 132   132
8       8    1    1       132 132   132
12      6    0    1       175 175   175
16      0    0    1       221 221   221
17      6    2    1       200 200   200
18      7    2    1       200 200   200
19      6    1    1       200 200   200
20      7    1    1       200 200   200
[... Truncated for brevity ...]
250     6    1    65      173 173   173
251     6    0    65      173 173   173
252     7    0    65      173 173   173
253     4    2    65      240 240   240
254     5    2    65      240 240   240
255     6    2    65      240 240   240
256     5    3    65      240 240   240
257     5    4    65      196 196   196
258     6    4    65      196 196   196
259     6    3    65      196 196   196
260     7    3    65      196 196   196
261     4    5    65      227 227   227
262     5    5    65      227 227   227
263     6    5    65      227 227   227
264     5    6    65      227 227   227
265     5    7    65      247 247   247
266     6    7    65      247 247   247
267     6    6    65      247 247   247
268     7    6    65      247 247   247
269     5    9    65      244 244   244
270     6    9    65      244 244   244
271     6    8    65      244 244   244
272     7    8    65      244 244   244
```

```
273      0    1    72      203 203   203
274      1    1    72      203 203   203
275      1    0    72      203 203   203
276      2    0    72      203 203   203

FallingBlocks:
fallingBlockId posX posY angleDeg contentsId
19             5    21   0        20
62             5    21   0        64
69             5    20   0        72

Games:
gameId score isOver boardId fallingBlockId date
1      1     1      1       19             2017-07-18
2      8     1      21      62             2017-07-18
3      0     0      65      69             2017-07-18
Resuming Game from 2017-07-18
```

BEGIN TABLES BEFORE AND AFTER TRANSACTIONS

Erik M. Buck

July 18, 2017

A.txt vs. B.txt

A.txt - /Users/erik/Desktop/CS7700

Blocks: (Before Transaction "DBTetrisGame: Update Falling Block")

blockId	posX	posY	groupId	red	green	blue
17	8	1	1	253	253	253
18	8	0	1	253	253	253
21	6	0	1	215	215	215
25	9	2	1	191	191	191
26	9	1	1	191	191	191
27	9	0	1	191	191	191
28	10	0	1	191	191	191
30	4	0	1	144	144	144
31	5	0	1	144	144	144
32	5	1	1	144	144	144
33	4	2	1	207	207	207
34	5	2	1	207	207	207
35	6	2	1	207	207	207
36	5	3	1	207	207	207
37	7	5	1	248	248	248
38	6	5	1	248	248	248
39	6	4	1	248	248	248
40	5	4	1	248	248	248
41	4	6	1	189	189	189
42	5	6	1	189	189	189
43	6	6	1	189	189	189
44	7	6	1	189	189	189
45	7	8	1	211	211	211
46	6	8	1	211	211	211
47	6	7	1	211	211	211
48	5	7	1	211	211	211
49	7	10	1	190	190	190
50	6	10	1	190	190	190
51	6	9	1	190	190	190
52	5	9	1	190	190	190
53	4	11	1	188	188	188
54	5	11	1	188	188	188
55	6	11	1	188	188	188
56	6	12	1	188	188	188
57	5	14	1	238	238	238
58	6	14	1	238	238	238
59	6	13	1	238	238	238
60	7	13	1	238	238	238
61	4	15	1	204	204	204
62	5	15	1	204	204	204
63	6	15	1	204	204	204
64	6	16	1	204	204	204
65	4	17	1	254	254	254
66	5	17	1	254	254	254
67	6	17	1	254	254	254
68	6	18	1	254	254	254
69	5	20	1	188	188	188
70	6	20	1	188	188	188
71	5	19	1	188	188	188
72	6	19	1	188	188	188
73	7	22	1	203	203	203
74	6	22	1	203	203	203
75	6	21	1	203	203	203
76	5	21	1	203	203	203
77	2	1	21	147	147	147
78	1	1	21	147	147	147
79	1	0	21	147	147	147
80	0	0	21	147	147	147
104	9	0	22	224	224	224
105	4	0	22	140	140	140
109	10	0	22	185	185	185
110	10	1	22	185	185	185
111	9	1	22	185	185	185
112	9	2	22	185	185	185
116	6	0	22	152	152	152
121	-1	0	33	243	243	243
122	0	0	33	243	243	243
123	1	0	33	243	243	243
124	0	1	33	243	243	243

Games: (Before Transaction "DBTetrisGame: Update Falling Block")

1	2	1	1	20	2017-07-18
2	3	0	22	31	2017-07-18

FallingBlocks: (Before Transaction "DBTetrisGame: Update Falling Block")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
31	1	0	0	33

B.txt - /Users/erik/Desktop/CS7700

Blocks: (After Transaction A "DBTetrisGame: Update Falling Block")

blockId	posX	posY	groupId	red	green	blue
17	8	1	1	253	253	253
18	8	0	1	253	253	253
21	6	0	1	215	215	215
25	9	2	1	191	191	191
26	9	1	1	191	191	191
27	9	0	1	191	191	191
28	10	0	1	191	191	191
30	4	0	1	144	144	144
31	5	0	1	144	144	144
32	5	1	1	144	144	144
33	4	2	1	207	207	207
34	5	2	1	207	207	207
35	6	2	1	207	207	207
36	5	3	1	207	207	207
37	7	5	1	248	248	248
38	6	5	1	248	248	248
39	6	4	1	248	248	248
40	5	4	1	248	248	248
41	4	6	1	189	189	189
42	5	6	1	189	189	189
43	6	6	1	189	189	189
44	7	6	1	189	189	189
45	7	8	1	211	211	211
46	6	8	1	211	211	211
47	6	7	1	211	211	211
48	5	7	1	211	211	211
49	7	10	1	190	190	190
50	6	10	1	190	190	190
51	6	9	1	190	190	190
52	5	9	1	190	190	190
53	4	11	1	188	188	188
54	5	11	1	188	188	188
55	6	11	1	188	188	188
56	6	12	1	188	188	188
57	5	14	1	238	238	238
58	6	14	1	238	238	238
59	6	13	1	238	238	238
60	7	13	1	238	238	238
61	4	15	1	204	204	204
62	5	15	1	204	204	204
63	6	15	1	204	204	204
64	6	16	1	204	204	204
65	4	17	1	254	254	254
66	5	17	1	254	254	254
67	6	17	1	254	254	254
68	6	18	1	254	254	254
69	5	20	1	188	188	188
70	6	20	1	188	188	188
71	5	19	1	188	188	188
72	6	19	1	188	188	188
73	7	22	1	203	203	203
74	6	22	1	203	203	203
75	6	21	1	203	203	203
76	5	21	1	203	203	203
77	2	1	21	147	147	147
78	1	1	21	147	147	147
79	1	0	21	147	147	147
80	0	0	21	147	147	147
104	9	0	22	224	224	224
105	4	0	22	140	140	140
109	10	0	22	185	185	185
110	10	1	22	185	185	185
111	9	1	22	185	185	185
112	9	2	22	185	185	185
116	6	0	22	152	152	152
121	0	0	22	243	243	243
122	1	0	22	243	243	243
123	2	0	22	243	243	243
124	1	1	22	243	243	243

Games: (After Transaction A "DBTetrisGame: Update Falling Block")

1	2	1	1	20	2017-07-18
2	3	0	22	None	2017-07-18

FallingBlocks: (After Transaction A "DBTetrisGame: Update Falling Block")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21

These are the before and after tables for transaction "DBTetrisGame: Update Falling Block"

status: 4 differences

Actions

A.txt vs. B.txt

A.txt - /Users/erik/Desktop/CS7700

```
Blocks: (Before Transaction "DBTetrisGame: Remove All Blocks in Complete Row
blockId posX posY groupId red green blue
17      8    1    1       253 253   253
18      8    0    1       253 253   253

[ ... Truncated for brevity ...]

187     6    20   22      173 173   173
188     7    20   22      173 173   173
189     0    1    50      138 138   138
190     1    1    50      138 138   138
191     0    0    50      138 138   138
192     1    0    50      138 138   138
193     0    0    51      160 160   160
194     1    0    51      160 160   160
195     2    0    51      160 160   160
196     1    1    51      160 160   160
197     3    0    51      178 178   178
198     4    0    51      178 178   178
199     5    0    51      178 178   178
200     6    0    51      178 178   178
201     8    0    51      133 133   133
202     9    0    51      133 133   133
203     10   0    51      133 133   133
204     10   1    51      133 133   133
205     6    2    51      183 183   183
206     6    1    51      183 183   183
207     7    1    51      183 183   183
208     7    0    51      183 183   183
209     -1   0    56      182 182   182
210     0    0    56      182 182   182
211     1    0    56      182 182   182
212     -1   1    56      182 182   182
Games: (Before Transaction "DBTetrisGame: Remove All Blocks in Complete Rows
1       2    1     1       20              2017-07-18
2       5    1     22      48              2017-07-18
3       0    0     51      53              2017-07-18
```

B.txt - /Users/erik/Desktop/CS7700

```
Blocks: (After Transaction A "DBTetrisGame: Remove All Blocks in Complete Ro
blockId posX posY groupId red green blue
17      8    1    1       253 253   253
18      8    0    1       253 253   253

[ ... Truncated for brevity ...]

187     6    20   22      173 173   173
188     7    20   22      173 173   173
189     0    1    50      138 138   138
190     1    1    50      138 138   138
191     0    0    50      138 138   138
192     1    0    50      138 138   138
196     1    0    51      160 160   160
204     10   0    51      133 133   133
205     6    1    51      183 183   183
206     6    0    51      183 183   183
207     7    0    51      183 183   183
209     -1   0    56      182 182   182
210     0    0    56      182 182   182
211     1    0    56      182 182   182
212     -1   1    56      182 182   182
Games: (After Transaction A "DBTetrisGame: Remove All Blocks in Complete Row
1       2    1     1       20              2017-07-18
2       5    1     22      48              2017-07-18
3       1    0     51      53              2017-07-18
```

status: 4 differences

Actions

These are the before and after tables for transaction "DBTetrisGame: Remove All Blocks in Complete Rows within the Game Board and Update the Game's Score").

A.txt vs. B.txt

A.txt - /Users/erik/Desktop/CS7700

Games: (Before Transaction "DBTetrisGame: Create Game and Board")

gameId	score	isOver	boardId	fallingBlockId	date
1	2	1	1	20	2017-07-18

FallingBlocks: (Before Transaction "DBTetrisGame: Add Falling Block")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21

FallingBlocks: (Before Transaction "DBTetrisFallingBlock: Create Falling Blo

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21

FallingBlocks: (Before Transaction "DBTetrisGame: Move Falling Block Left")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
21	5	18	0	23

FallingBlocks: (Before Transaction "DBTetrisGame: Move Falling Block Right")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
22	5	20	0	24

FallingBlocks: (Before Transaction "DBTetrisGame: Rotate Falling Block Count

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
27	5	18	0	29

FallingBlocks: (Before Transaction "DBTetrisGame: Rotate Falling Block Clock

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
28	5	17	0	30

B.txt - /Users/erik/Desktop/CS7700

Games: (After Transaction "DBTetrisGame: Create Game and Board")

gameId	score	isOver	boardId	fallingBlockId	date
1	2	1	1	20	2017-07-18
2	0	0	22	None	2017-07-18

FallingBlocks: (After Transaction "DBTetrisFallingBlock: Create Falling Bloc

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
21	5	22	0	23

FallingBlocks: (After Transaction "DBTetrisGame: Add Falling Block")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
21	5	22	0	23

FallingBlocks: (After Transaction "DBTetrisGame: Move Falling Block Left")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
21	4	18	0	23

FallingBlocks: (After Transaction "DBTetrisGame: Move Falling Block Right")

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
22	6	20	0	24

FallingBlocks: (After Transaction "DBTetrisGame: Rotate Falling Block Counte

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
27	5	18	90	29

FallingBlocks: (After Transaction "DBTetrisGame: Rotate Falling Block Clockw

fallingBlockId	posX	posY	angleDeg	contentsId
20	5	21	0	21
28	5	17	-90	30

These are the before and after tables for transactions "DBTetrisGame: Create Game and Board", "DBTetrisGame: Add Falling Block", "DBTetrisFallingBlock: Create Falling Block", "DBTetrisGame: Move Falling Block Left", "DBTetrisGame: Move Falling Block Right", "DBTetrisGame: Rotate Falling Block Counterclockwise", and "DBTetrisGame: Rotate Falling Block Clockwise".

status: 14 differences

Actions