(с.126) добавить в AppModel

```
private fun moveValid(position: Point, frameNumber: Int?): Boolean {
    val shape: Array<ByteArray>? = currentBlock?.getShape(frameNumber
as Int)
    return validTranslation(position, shape as Array<ByteArray>)
}

fun generateField(action: String) {
    if (isGameActive()) {
        resetField()
        var frameNumber: Int? = currentBlock?.frameNumber
        val coordinate: Point? = Point()
        coordinate?.x = currentBlock?.position?.x
        coordinate?.y = currentBlock?.position?.y

        when (action) {
            Motions.LEFT.name -> {
                coordinate?.x = currentBlock?.position?.x?.minus(1)
            }
            Motions.RIGHT.name -> {
                coordinate?.x = currentBlock?.position?.x?.plus(1)
            }
            Motions.DOWN.name -> {
                coordinate?.y = currentBlock?.position?.y?.plus(1)
            }
            Motions.ROTATE.name -> {
                frameNumber = frameNumber?.plus(1)
                if (frameNumber != null) {
                    if (frameNumber >= currentBlock?.frameCount as
Int) {
                        frameNumber = 0
                    }
                }
            }
        }

        if (!moveValid(coordinate as Point, frameNumber)) {
            translateBlock(
                currentBlock?.position as Point,
                currentBlock?.frameNumber as Int
            )
            if (Motions.DOWN.name == action) {
                boostScore()
                persistCellData()
                assessField()
                generateNextBlock()
                if (!blockAdditionPossible()) {
                    currentState = Statuses.OVER.name
                    currentBlock = null
                    resetField(false)
                }
```

```
                }
            } else {
                if (frameNumber != null) {
                    translateBlock(coordinate, frameNumber)
                    currentBlock?.setState(frameNumber, coordinate)
                }
            }
        }
    }

    private fun resetField(ephemeralCellsOnly: Boolean = true) {
        for (i in 0 until FieldConstants.ROW_COUNT.value) {
            (0 until FieldConstants.COLUMN_COUNT.value)
                .filter {
                    !ephemeralCellsOnly || field[i][it] ==
                            CellConstants.EPHEMERAL.value
                }
                .forEach { field[i][it] = CellConstants.EMPTY.value }
        }
    }

    private fun persistCellData() {
        for (i in 0 until field.size) {
            for (j in 0 until field[i].size) {
                var status = getCellStatus(i, j)
                if (status == CellConstants.EPHEMERAL.value) {
                    status = currentBlock?.staticValue
                    setCellStatus(i, j, status)
                }
            }
        }
    }

    private fun assessField() {
        for (i in 0 until field.size) {
            var emptyCells = 0
            for (j in 0 until field[i].size) {
                val status = getCellStatus(i, j)
                val isEmpty = CellConstants.EMPTY.value == status
                if (isEmpty)
                    emptyCells++
            }
            if (emptyCells == 0)
                shiftRows(i)
        }
    }

    private fun translateBlock(position: Point, frameNumber: Int) {
        synchronized(field) {
            val shape: Array<ByteArray>? =
    currentBlock?.getShape(frameNumber)
```

```
            if (shape != null) {
                for (i in shape.indices) {
                    for (j in 0 until shape[i].size) {
                        val y = position.y + i
                        val x = position.x + j
                        if (CellConstants.EMPTY.value != shape[i][j]) {
                            field[y][x] = shape[i][j]
                        }
                    }
                }
            }
        }
    }

    private fun blockAdditionPossible(): Boolean {
        if (!moveValid(
                currentBlock?.position as Point,
                currentBlock?.frameNumber
            )
        ) {
            return false
        }
        return true
    }

    private fun shiftRows(nToRow: Int) {
        if (nToRow > 0) {
            for (j in nToRow - 1 downTo 0) {
                for (m in 0 until field[j].size) {
                    setCellStatus(j + 1, m, getCellStatus(j, m))
                }
            }
        }

        for (j in 0 until field[0].size) {
            setCellStatus(0, j, CellConstants.EMPTY.value)
        }
    }

    fun startGame() {
        if (!isGameActive()) {
            currentState = Statuses.ACTIVE.name
            generateNextBlock()
        }
    }

    fun restartGame() {
        resetModel()
        startGame()
    }
```

```
fun endGame() {
    score = 0
    currentState = AppModel.Statuses.OVER.name
}

private fun resetModel() {
    resetField(false)
    currentState = Statuses.AWAITING_START.name
    score = 0
}
```

Создать в исходном проекте пакет view, а в нем файл/класс TetrisView.kt:

```
package com.example.tetris.view

import android.content.Context
import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import android.graphics.RectF
import android.os.Handler
import android.os.Message
import android.util.AttributeSet
import android.view.View
import android.widget.Toast
import androidx.annotation.Dimension
import com.example.tetris.constants.CellConstants
import com.example.tetris.GameActivity
import com.example.tetris.constants.FieldConstants
import com.example.tetris.models.AppModel
import com.example.tetris.models.Block
import java.security.acl.Owner
import java.text.AttributedCharacterIterator

class TetrisView : View {
    private val paint = Paint()
    private var lastMove: Long = 0
    private var model: AppModel? = null
    private var activity: GameActivity? = null
    private val viewHandler = ViewHandler(this)
    private var cellSize: Dimension = Dimension(0, 0)
    private var frameOffset: Dimension = Dimension(0, 0)

    constructor(context: Context, attrs: AttributeSet) :
            super(context, attrs)

    constructor(context: Context, attrs: AttributeSet, defStyle: Int)
:
            super(context, attrs, defStyle)

    companion object {
```

```
        private val DELAY = 500
        private val BLOCK_OFFSET = 2
        private val FRAME_OFFSET_BASE = 10
    }

    fun setModel(model: AppModel) {
        this.model = model
    }

    fun setActivity(gameActivity: GameActivity) {
        this.activity = gameActivity
    }

    fun setGameCommand(move: AppModel.Motions) {
        if (model != null && (model?.currentState ==
AppModel.Statuses.ACTIVE.name)) {
            if (AppModel.Motions.DOWN == move) {
                model?.generateField(move.name)
                invalidate()
                return
            }
            setGameCommandWithDelay(move)
        }
    }

    fun setGameCommandWithDelay(move: AppModel.Motions) {
        val now = System.currentTimeMillis()
        if (now - lastMove > DELAY) {
            model?.generateField(move.name)
            invalidate()
            lastMove = now
        }
        updateScores()
        viewHandler.sleep(DELAY.toLong())
    }

    private fun updateScores() {
        activity?.tvCurrentScore?.text = "${model?.score}"
        activity?.tvHighScore?.text =
"${activity?.appPreferences?.getHighScore()}"
    }

    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        drawFrame(canvas)
        if (model != null) {
            for (i in 0 until FieldConstants.ROW_COUNT.value) {
                for (j in 0 until FieldConstants.COLUMN_COUNT.value) {
                    drawCell(canvas, i, j)
                }
            }
```

```
        }
    }

    private fun drawFrame(canvas: Canvas) {
        paint.color = Color.LTGRAY
        canvas.drawRect(
            frameOffset.width.toFloat(),
            frameOffset.height.toFloat(),
            width - frameOffset.width.toFloat(),
            height - frameOffset.height.toFloat(), paint
        )
    }

    private fun drawCell(canvas: Canvas, row: Int, col: Int) {
        val cellStatus = model?.getCellStatus(row, col)
        if (CellConstants.EMPTY.value != cellStatus) {
            val color = if (CellConstants.EPHEMERAL.value ==
cellStatus) {
                model?.currentBlock?.color
            } else {
                Block.getColor(cellStatus as Byte)
            }
            drawCell(canvas, col, row, color as Int)
        }
    }

    private fun drawCell(canvas: Canvas, x: Int, y: Int, rgbColor:
Int) {
        paint.color = rgbColor
        val top: Float = (frameOffset.height + y * cellSize.height +
BLOCK_OFFSET).toFloat()
        val left: Float = (frameOffset.width + x * cellSize.width +
BLOCK_OFFSET).toFloat()
        val bottom: Float = (frameOffset.height + (y + 1) *
cellSize.height -
                BLOCK_OFFSET).toFloat()
        val right: Float = (frameOffset.width + (x + 1) *
cellSize.width -
                BLOCK_OFFSET).toFloat()
        val rectangle = RectF(left, top, right, bottom)
        canvas.drawRoundRect(rectangle, 4F, 4F, paint)
    }

    override fun onSizeChanged(width: Int, height: Int, previousWidth:
Int, previousHeight: Int) {
        super.onSizeChanged(width, height, previousWidth,
previousHeight)
        val cellWidth = (width - 2 * FRAME_OFFSET_BASE) /
FieldConstants.COLUMN_COUNT.value
        val cellHeight = (height - 2 * FRAME_OFFSET_BASE) /
FieldConstants.ROW_COUNT.value
```

```
        val n = Math.min(cellWidth, cellHeight)
        this.cellSize = TetrisView.Dimension(n, n)
        val offsetX = (width - FieldConstants.COLUMN_COUNT.value * n)
/ 2
        val offsetY = (height - FieldConstants.ROW_COUNT.value * n) /
2
        this.frameOffset = TetrisView.Dimension(offsetX, offsetY)
    }

    private class ViewHandler(private val owner: TetrisView) :
Handler() {
        override fun handleMessage(message: Message) {
            if (message.what == 0) {
                if (owner.model != null) {
                    if (owner.model!!.isGameOver()) {
                        owner.model?.endGame()
                        Toast.makeText(
                            owner.activity, "Game over",
                            Toast.LENGTH_LONG
                        ).show()
                    }
                    if (owner.model!!.isGameActive()) {

owner.setGameCommandWithDelay(AppModel.Motions.DOWN)
                    }
                }
            }
        }

        fun sleep(delay: Long) {
            this.removeMessages(0)
            sendMessageDelayed(obtainMessage(0), delay)
        }
    }

    private data class Dimension(val width: Int, val height: Int)

}
```