public class Memory {  
 private static SharedPreferences *appSharedPrefs*;  
 private static Editor *prefsEditor*;  
  
 @SuppressLint("CommitPrefEdits")  
 public static void init(Context context) {  
 *appSharedPrefs* = PreferenceManager.*getDefaultSharedPreferences*(context);  
 *prefsEditor* = *appSharedPrefs*.edit();  
 }  
  
 public static int getInt(Context context, String key, int defValue) {  
 *init*(context);  
  
 return *appSharedPrefs*.getInt(key, defValue);  
 }  
  
 public static void setInt(Context context, String key, int value) {  
 *init*(context);  
  
 *prefsEditor*.putInt(key, value);  
 *prefsEditor*.commit();  
 }

}

Memory

bestint = Memory.*getInt*(mContext, "BEST", 0);

Memory.*setInt*(mContext, "BEST", bestint);

private void init(Context context) {  
 activity = (Activity) context;  
 myGestureListener = new MyGestureListener(context);  
 mContext = context;  
 getHolder().addCallback(this);  
  
 Resources res = getResources();  
  
 bestint = Memory.*getInt*(mContext, "BEST", 0);  
 apple = BitmapFactory.*decodeResource*(res, R.drawable.*apple*);  
 int width = apple.getWidth();  
 int height = apple.getHeight();  
 int newWidth = SIDE\_LENGTH;  
 int newHeight = SIDE\_LENGTH;  
  
 float scaleWidth = ((float) newWidth) / width;  
 float scaleHeight = ((float) newHeight) / height;  
  
 Matrix matrix = new Matrix();  
 matrix.postScale(scaleWidth, scaleHeight);  
 newbm = Bitmap.*createBitmap*(apple, 0, 0, width, height, matrix, true);  
 applePaint = new Paint();  
  
 snakePaint = new Paint();  
 xPaint = new Paint();  
 targets = new Point();  
 targetsPaint = new Paint();  
 backgroundPaint = new Paint();  
 textPaint = new Paint();  
  
 if (mContext instanceof SnakeActivity) {  
 switch (((SnakeActivity) mContext).level) {  
 case "2":  
 speed = 200;  
 snakelevel = 200;  
 break;  
 case "3":  
 speed = 150;  
 snakelevel = 150;  
 break;  
 case "4":  
 speed = 100;  
 snakelevel = 100;  
 break;  
 case "5":  
 speed = 50;  
 snakelevel = 50;  
 break;  
  
 default:  
 speed = 250;  
 snakelevel = 250;  
 }  
 }  
}

@Override  
protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
 super.onSizeChanged(w, h, oldw, oldh);  
  
 screenWidth = w;  
 screenHeight = h;  
 width\_count = w / SIDE\_LENGTH;  
 height\_count = h / SIDE\_LENGTH;  
 backgroundPaint.setColor(Color.*WHITE*);  
 snakePaint.setColor(Color.*BLACK*);  
 xPaint.setColor(Color.*YELLOW*);  
 targetsPaint.setColor(Color.*RED*);  
 textPaint.setTextSize(50);  
  
 width\_offset = (screenWidth - width\_count \* SIDE\_LENGTH) / 2;  
 height\_offset = (screenHeight - height\_count \* SIDE\_LENGTH) / 2;  
  
 if (width\_offset != 0 || height\_offset != 0)  
 newGame();  
  
 RelativeLayout.LayoutParams lp = (RelativeLayout.LayoutParams) getLayoutParams();  
 lp.setMargins((int) width\_offset, (int) height\_offset, (int) width\_offset, (int) height\_offset);  
 setLayoutParams(lp);  
}

public void newGame() {  
  
 time = 0;  
 score = 0;  
 timeLeft = 20;  
 speed = snakelevel;  
 //shotsFired = 0;  
 totalElapsedTime = 0.0;  
 *nowdirection* = *snakeUp*;  
 snakeover = false;  
 targets = new Point(new Random().nextInt(width\_count) \* SIDE\_LENGTH,  
 new Random().nextInt(height\_count) \* SIDE\_LENGTH);  
 startx = (width\_count) / 2 \* SIDE\_LENGTH;  
 starty = (height\_count) / 2 \* SIDE\_LENGTH;  
 arrayList.clear();  
 arrayList.add(new Point(startx, starty));  
 arrayList.add(new Point(startx, starty + SIDE\_LENGTH));  
 arrayList.add(new Point(startx, starty + 2 \* SIDE\_LENGTH));  
 arrayList.add(new Point(startx, starty + 3 \* SIDE\_LENGTH));  
 arrayList.add(new Point(startx, starty + 4 \* SIDE\_LENGTH));  
  
  
 if (gameOver) {  
 gameOver = false;  
 cannonThread = new CannonThread(getHolder());  
 cannonThread.start();  
 }  
}

private void updatePositions(double elapsedTimeMS) {  
  
 double interval = elapsedTimeMS / 1000.0;  
  
 timeLeft -= interval;  
  
 if (screenWidth - arrayList.get(0).x < SIDE\_LENGTH || screenHeight - arrayList.get(0).y < SIDE\_LENGTH ||  
 arrayList.get(0).x < 0 || arrayList.get(0).y < 0) {  
 snakeover = true;  
 }  
  
 if ((Math.*abs*(timeLeft - time) \* 1000 > speed || time == 0) && !snakeover) {  
  
 time = timeLeft;  
 for (int i = arrayList.size() - 1; i > 0; i--) {  
 arrayList.set(i, arrayList.get(i - 1));  
 }  
 if (*nowdirection* == 0) {  
 arrayList.set(0, new Point(arrayList.get(0).x, arrayList.get(0).y - SIDE\_LENGTH));  
 } else if (*nowdirection* == 1) {  
 arrayList.set(0, new Point(arrayList.get(0).x, arrayList.get(0).y + SIDE\_LENGTH));  
 } else if (*nowdirection* == 2) {  
 arrayList.set(0, new Point(arrayList.get(0).x - SIDE\_LENGTH, arrayList.get(0).y));  
 } else if (*nowdirection* == 3) {  
 arrayList.set(0, new Point(arrayList.get(0).x + SIDE\_LENGTH, arrayList.get(0).y));  
 }  
  
 ArrayList<Point> tmpList = new ArrayList<>();  
 tmpList.addAll(arrayList);  
 Point tmpPoint = tmpList.remove(0);  
  
 if (tmpList.contains(tmpPoint))  
 snakeover = true;  
  
 }  
  
 if (snakeover) {  
 timeLeft = 0.0;  
 gameOver = true;  
 cannonThread.setRunning(false);  
 showGameOverDialog(R.string.*lose*);  
 }  
  
 if (arrayList.contains(targets)) {  
 arrayList.add(arrayList.get(arrayList.size() - 1));  
 for (int i = arrayList.size() - 2; i > 0; i--) {  
 arrayList.set(i, arrayList.get(i - 1));  
 }  
 if (*nowdirection* == 0) {  
 arrayList.set(0, new Point(arrayList.get(1).x, arrayList.get(1).y - SIDE\_LENGTH));  
 } else if (*nowdirection* == 1) {  
 arrayList.set(0, new Point(arrayList.get(1).x, arrayList.get(1).y + SIDE\_LENGTH));  
 } else if (*nowdirection* == 2) {  
 arrayList.set(0, new Point(arrayList.get(1).x - SIDE\_LENGTH, arrayList.get(1).y));  
 } else if (*nowdirection* == 3) {  
 arrayList.set(0, new Point(arrayList.get(1).x + SIDE\_LENGTH, arrayList.get(1).y));  
 }  
 targets = new Point(new Random().nextInt(width\_count) \* SIDE\_LENGTH,  
 new Random().nextInt(height\_count) \* SIDE\_LENGTH);  
 score++;  
 speed = speed - 10;  
 if (score > bestint) {  
 bestint = score;  
 }  
 }  
}

public void drawGameElements(Canvas canvas) {  
 if (canvas == null)  
 return;  
  
 canvas.drawRect(0, 0, screenWidth, screenHeight, backgroundPaint);  
  
 for (Point point : arrayList) {  
 canvas.drawRect(point.x, point.y, point.x + SIDE\_LENGTH, point.y + SIDE\_LENGTH, snakePaint);  
 canvas.drawRect(point.x + 5, point.y + 5, point.x + SIDE\_LENGTH - 5, point.y + SIDE\_LENGTH - 5, xPaint);  
 }  
  
 canvas.drawBitmap(newbm, targets.x, targets.y, applePaint);  
 canvas.drawText(getResources().getString(R.string.*score*, score), screenWidth - 250, 60, textPaint);  
 canvas.drawText(getResources().getString(R.string.*best*, bestint), 40, 60, textPaint);  
}

private void showGameOverDialog(final int messageId) {  
  
 final DialogFragment gameResult =  
 new DialogFragment() {  
 @Override  
 public Dialog onCreateDialog(Bundle bundle) {  
 AlertDialog.Builder builder =  
 new AlertDialog.Builder(getActivity());  
 builder.setTitle(getResources().getString(messageId));  
 builder.setMessage(getResources().getString(  
 R.string.*results\_format*, score, bestint));  
 Memory.*setInt*(mContext, "BEST", bestint);  
  
 builder.setNegativeButton(R.string.*back*,  
 new DialogInterface.OnClickListener() {  
 @Override  
 public void onClick(DialogInterface dialog, int which) {  
 if (mContext instanceof SnakeActivity)  
 ((SnakeActivity) mContext).onBackPressed();  
 }  
 }  
 );  
  
 builder.setPositiveButton(R.string.*reset\_game*,  
 new DialogInterface.OnClickListener() {  
 @Override  
 public void onClick(DialogInterface dialog, int which) {  
 dialogIsDisplayed = false;  
 newGame();  
 }  
 }  
 );  
 return builder.create();  
 }  
 };  
  
  
 activity.runOnUiThread(  
 new Runnable() {  
 public void run() {  
 dialogIsDisplayed = true;  
 gameResult.setCancelable(false);  
 gameResult.show(activity.getFragmentManager(), "results");  
 }  
 }  
 );  
}

@Override  
public void surfaceCreated(SurfaceHolder holder) {  
 if (!dialogIsDisplayed) {  
 cannonThread = new CannonThread(holder);  
 cannonThread.setRunning(true);  
 cannonThread.start();  
 }  
}

@Override  
public void surfaceDestroyed(SurfaceHolder holder) {  
 boolean retry = true;  
 cannonThread.setRunning(false);  
 while (retry) {  
 try {  
 cannonThread.join();  
 retry = false;  
 } catch (InterruptedException e) {  
 Log.*e*(*TAG*, "Thread interrupted", e);  
 }  
 }  
}

private class CannonThread extends Thread {  
 private final SurfaceHolder surfaceHolder;  
 private boolean threadIsRunning = true;  
  
 public CannonThread(SurfaceHolder holder) {  
 surfaceHolder = holder;  
 setName("CannonThread");  
 }  
  
 public void setRunning(boolean running) {  
 threadIsRunning = running;  
 }  
  
 @Override  
 public void run() {  
 Canvas canvas = null;  
 long previousFrameTime = System.*currentTimeMillis*();  
  
 while (threadIsRunning) {  
 try {  
 canvas = surfaceHolder.lockCanvas();  
  
 synchronized (surfaceHolder) {  
 long currentTime = System.*currentTimeMillis*();  
 double elapsedTimeMS = currentTime - previousFrameTime;  
 totalElapsedTime += elapsedTimeMS / 1000.0;  
 updatePositions(elapsedTimeMS);  
 if (!snakeover) {  
 drawGameElements(canvas);  
 }  
 previousFrameTime = currentTime;  
 }  
 } finally {  
 if (canvas != null)  
 surfaceHolder.unlockCanvasAndPost(canvas);  
 }  
 }  
 }  
}

@Override  
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {  
 float x1 = e1.getX();  
 float y1 = e1.getY();  
 float x2 = e2.getX();  
 float y2 = e2.getY();  
  
 Direction direction = getDirection(x1, y1, x2, y2);  
 if (direction == Direction.*left*) {  
 if (*nowdirection* != *snakeRight*)  
 *nowdirection* = *snakeLeft*;  
 } else if (direction == Direction.*right*) {  
 if (*nowdirection* != *snakeLeft*)  
 *nowdirection* = *snakeRight*;  
 } else if (direction == Direction.*up*) {  
 if (*nowdirection* != *snakeDown*)  
 *nowdirection* = *snakeUp*;  
 } else if (direction == Direction.*down*) {  
 if (*nowdirection* != *snakeUp*)  
 *nowdirection* = *snakeDown*;  
 }  
 return super.onFling(e1, e2, velocityX, velocityY);  
}

public Direction getDirection(float x1, float y1, float x2, float y2) {  
 double angle = getAngle(x1, y1, x2, y2);  
 return Direction.*get*(angle);  
}

public double getAngle(float x1, float y1, float x2, float y2) {  
  
 double rad = Math.*atan2*(y1 - y2, x2 - x1) + Math.*PI*;  
 return (rad \* 180 / Math.*PI* + 180) % 360;  
}

public enum Direction {  
 *up*,  
 *down*,  
 *left*,  
 *right*;  
  
 */\*\*  
 \* Returns a direction given an angle.  
 \* Directions are defined as follows:  
 \* <p/>  
 \* Up: [45, 135]  
 \* Right: [0,45] and [315, 360]  
 \* Down: [225, 315]  
 \* Left: [135, 225]  
 \*  
 \** ***@param*** *angle an angle from 0 to 360 - e  
 \** ***@return*** *the direction of an angle  
 \*/* public static Direction get(double angle) {  
 if (*inRange*(angle, 45, 135)) {  
 return Direction.*up*;  
 } else if (*inRange*(angle, 0, 45) || *inRange*(angle, 315, 360)) {  
 return Direction.*right*;  
 } else if (*inRange*(angle, 225, 315)) {  
 return Direction.*down*;  
 } else {  
 return Direction.*left*;  
 }  
  
 }  
  
 */\*\*  
 \** ***@param*** *angle an angle  
 \** ***@param*** *init the initial bound  
 \** ***@param*** *end the final bound  
 \** ***@return*** *returns true if the given angle is in the interval [init, end).  
 \*/* private static boolean inRange(double angle, float init, float end) {  
 return (angle >= init) && (angle < end);  
 }  
}