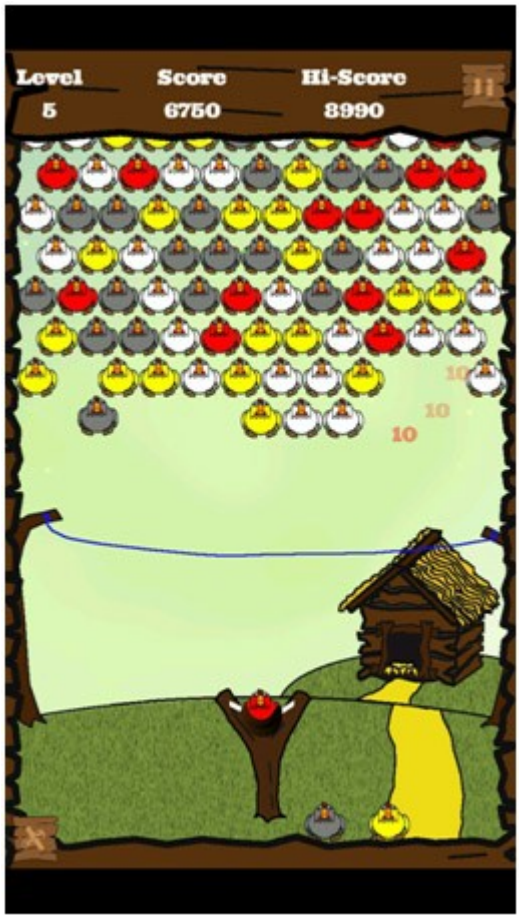
1. Game scene

The game scene is the most complicated. This is the place where the whole game's action takes place. All layers are attached to this scene. It also initializes and manages the game's physics. The appearance of this scene during the game is presented below.



[Image 9] The appearance of the game scene

1.1 Physics

There is a basic physics engine implemented in the HenShooter game. It is responsible for handling collisions of the object's bodies. Additionaly, it applies gravity and wind forces to the moving objects. The AndEngine library provides the physics implementation as a Box2D extension. It has two classes which represent the physical world: PhysicsWorld and FixedStepPhysicsWorld. Both these classes take the gravity vector as their constructor's first parameter. The main difference between them is that FixedStepPhysicsWorld takes another parameter indicating the number of steps per second. There is no guarantee that the number of frames will be exactly the same as the provided one, but AndEngine will try to achieve this FPS count. The HenShooter game uses FixedStepPhysicsWorld with 30 steps per second and a gravity vector with wind strength and earth's gravity values. The physics's world gravity is updated each time the player manually changes wind direction. The collisions of the objects are handled by the ContactListener which can be added to the physics world instance with

the setContactListener(...) method. HenShooter provides its own CollisionListener class which implements methods from the Contact Listener interface.

2. Effects

Effects are a very important part of the game. In our game almost all effects are located on the EffectLayer. To present the various effects, we used different mechanisms such as a particle system, or animations.

2.1 The particle system

The particle system is a mechanism which is used to present effects which can be present as some small particles, for example explosions or fire. In the HenShooter application the particle system is used to display effects occuring after the hens and bullets destruction, and to present the effects of the wind. The way to create this effects, is shown in the following code.

```
public void prepareExplosion(final float pX, final float pY, final Color pColor) {
        final PointParticleEmitter particleEmitter = new PointParticleEmitter(pX, pY);
        final SpriteParticleSystem particleSystem = new
SpriteParticleSystem(particleEmitter, 100, 100, 7,
getGameTextureRegion(GameTextureKeys.EFFECTS_BOOM_PARTICLE),
getVertexBufferObjectManager());

        particleSystem.addParticleInitializer(
                new BlendFunctionParticleInitializer<Sprite>(GLES20.GL_SRC_ALPHA,
                        GLES20.GL_ONE));
        particleSystem.addParticleInitializer(
                new ColorParticleInitializer<Sprite>(pColor));
        particleSystem.addParticleInitializer(
                new RotationParticleInitializer<Sprite>(0.0f, 360.0f));
        particleSystem.addParticleInitializer(
                new GravityParticleInitializer<Sprite>());
        particleSystem.addParticleInitializer(
                new ScaleParticleInitializer<Sprite>(0.5f));
        particleSystem.addParticleModifier(
                new AlphaParticleModifier<Sprite>(EXPLOSION_TIME - 1.0f, 275
        EXPLOSION_TIME- HALF, 1.0f, 0.0f));
        particleSystem.addParticleInitializer(
                new VelocityParticleInitializer<Sprite>(-32, 32, -32, 32));

        particleSystem.setCullingEnabled(true);
        mListOfExplosions.add(particleSystem);
}
```

[Code 21] Body of method which prepare system particle for hens which will be destroy

The SpriteParticleSystem is created using a constructor which takes the following values: ParticleEmiter (object which defines coordinates), minimum rate as float, maximum rate as float,

texture region which will be displayed and VertexBufferObjectManager. In the next steps the particle initializers are added to the particle system. AndEngine Framework provides many particle initializers. In the HenShooter application, some of them are used. BlendFunctionParticleInitializer is created via a constructor which takes two constants from the GLES20 class as parameters. The first of them defines the source function, the seconds one defines the destination function. The ColorParticleInitializer takes color as constructor parameter. This color will be used to color particles. A RotationParticleInitializer is used to rotate elements. The first parameter of its constructor defines the minimum rotation angle. The second parameter defines the maximum rotation angle. The GravityParticleInitializer simulates gravity. The ScaleParticleInitializer sets the scale for one particle. This object's constructor takes in two parameters. The first parameter is the minimum scale, the second is maximum scale. If there is only one parameter, then both these values are the same. The last initializer is a VelocityParticleInitializer. It defines the minimum X velocity, maximum X velocity, minimum Y velocity and maximum Y velocity. It is additionally possible to add modifiers. A AlphaParticleModifier modifies the color transparency based on four parameters: from time, to time, from alpha, to alpha. The action effect is shown in the following picture.

[Image 10] System particles

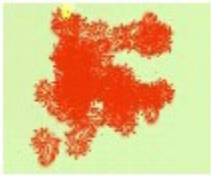
Below is an example of how to use the particles system

```
public void prepareBombExplosion(final float pX, final float pY) {
final PointParticleEmitter particleEmitter = new PointParticleEmitter(pX, pY);
final SpriteParticleSystem particleSystem = new SpriteParticleSystem(particleEmitter, 100,
100, 50, getGameTextureRegion(GameTextureKeys.EFFECTS_BOMB_PARTICLE),
getVertexBufferObjectManager());
particleSystem.addParticleInitializer(new ColorParticleInitializer<Sprite>(0.95f, 0.21f,
0.04f));
particleSystem.addParticleInitializer(new ColorParticleInitializer<Sprite>(0.95f, 0.92f,
0.04f));
particleSystem.addParticleInitializer(new ColorParticleInitializer<Sprite>(0.93f, 0.93f,
0.93f));
particleSystem.addParticleInitializer(new
ExpireParticleInitializer<Sprite>(EXPLOSION_TIME));
particleSystem.addParticleInitializer(new RotationParticleInitializer<Sprite>(0.0f,
360.0f));
particleSystem.addParticleInitializer(new ScaleParticleInitializer<Sprite>(0.5f, 1.5f));

particleSystem.addParticleModifier(new AlphaParticleModifier<Sprite>(EXPLOSION_TIME - 1f,
EXPLOSION_TIME, 1.0f, 0.0f));
```

```
particleSystem.addParticleInitializer(new VelocityParticleInitializer<Sprite>(-26, 58, -26,
58));
particleSystem.addParticleModifier(new ScaleParticleModifier<Sprite>(0, 5, 0.5f, 2.0f));
particleSystem.setCullingEnabled(true);
mBulletExplosion = particleSystem;
}
```

[Code 22] Method preparing the particle system for hens which will be destroyed.

Almost the same initializers and modifiers are used in this code as were used before. The ExpireParticleInitializer is used for the first time. This object's constructor takes a number that defines the time left to extinction. The action's effect is shown in following picture.



[Image 11] Bomb explosion

The function below is responsible for preparing the particle system for the thunderbolt.

```
private void createLightningPart(final float pX, final float pY, final boolean pScaleY) {
final RectangleParticleEmitter particleEmitter = new RectangleParticleEmitter(pX + 32, pY -
32, 64, 64);
final SpriteParticleSystem particleSystem = new SpriteParticleSystem(particleEmitter, 100,
100, 3,
getGameTextureRegion(GameTextureKeys.EFFECTS_LIGHTNING_PARTICLE),
getVertexBufferObjectManager());
particleSystem.addParticleInitializer(new AlphaParticleInitializer<Sprite>(0));
particleSystem.addParticleInitializer(new RotationParticleInitializer<Sprite>(0.0f,
360.0f));
particleSystem.addParticleInitializer(new ExpireParticleInitializer<Sprite>(12));
if (pScaleY) {
        particleSystem.addParticleModifier(new ScaleParticleModifier<Sprite>(0f, 0.1f, 1f,
1f, 1f, 2f));
}
particleSystem.addParticleModifier(new ScaleParticleModifier<Sprite>(0, 5, .7f, .5f));
particleSystem.addParticleModifier(new AlphaParticleModifier<Sprite>(0, 2, 0, 1));
particleSystem.addParticleModifier(new AlphaParticleModifier<Sprite>(10, 12, 1, 0));
particleSystem.setCullingEnabled(true);
mLightnings.add(particleSystem);
}
```

[Code 23] Method preparing the system particle for the thunderbolt

[Image 12] Thunderbolt effect

The picture above shows the thunderbolt effect. The last method which prepares effects is the method which is responsible for the wind particle. The code of this method is shown below.

```
private SpriteParticleSystem createWindParticleGenerator() {
        final RectangleParticleEmitter particleEmitter = new
RectangleParticleEmitter(mGameLayer.getCameraWidth() * HALF, (mGameLayer.getCameraHeight() -
550) * HALF, mGameLayer.getCameraWidth(), mGameLayer.getCameraHeight() - 550 -
HudLayer.CEILING_WIDTH);
        final SpriteParticleSystem particleSystem = new
SpriteParticleSystem(particleEmitter, 10, 10, 20,
getGameTextureRegion(GameTextureKeys.EFFECTS_WIND_PARTICLE),
getVertexBufferObjectManager());
        particleSystem.addParticleInitializer(new AlphaParticleInitializer<Sprite>(0));
        particleSystem.addParticleInitializer(new
BlendFunctionParticleInitializer<Sprite>(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE));
        particleSystem.addParticleInitializer(mVelocityParticleInitializer);
        particleSystem.addParticleInitializer(new RotationParticleInitializer<Sprite>(0.0f,
360.0f));
        particleSystem.addParticleInitializer(new ExpireParticleInitializer<Sprite>(12));
        particleSystem.addParticleModifier(new ScaleParticleModifier<Sprite>(0, 5, .7f, .
3f));
        particleSystem.addParticleModifier(new AlphaParticleModifier<Sprite>(0, 2, 0, 1));
        particleSystem.addParticleModifier(new AlphaParticleModifier<Sprite>(10, 12, 1,
0));
        particleSystem.setCullingEnabled(true);
        return particleSystem;
}
```

[Code 24] Method which prepares the system particle for wind

The SpriteParticleSystem is added to a layer like a normal sprite via the attachChild(IEntity) method.

2.2 Sprite animations

The next element which can enrich effects is the sprite animation. For a sprite to be animated, it has to be an instance of the AnimatedSprite class. Additionally in HenShooter this sprite has to be an instance of AnimatedSpriteWithAreaToTouch. The body of this class is shown in the following code.

```
public class AnimatedSpriteWithAreaToTouch extends AnimatedSprite {
```

```
        private final IAreaToTouch mAreaToTouch;
        public AnimatedSpriteWithAreaToTouch(final float pX, final float pY, final
ITiledTextureRegion pTiledTextureRegion, final VertexBufferObjectManager
pVertexBufferObjectManager, final IAreaToTouch pAreaToTouch) {
                        super(pX, pY, pTiledTextureRegion, pVertexBufferObjectManager);
                        mAreaToTouch = pAreaToTouch;
        }

        public void onAfterAreaTouched(final TouchEvent pSceneTouchEvent, final ITouchArea
pTouchArea, final float pTouchAreaLocalX, final float pTouchAreaLocalY) {
                final MotionEvent me = pSceneTouchEvent.getMotionEvent();
                final int pointerID = pSceneTouchEvent.getPointerID();
                final int index = me.findPointerIndex(pointerID);
                if (index == -1) {
                        return;
                }
                final int toolType = me.getToolType(index);

                if (mAreaToTouch != null) {
                        if (toolType == MotionEvent.TOOL_TYPE_ERASER) {
                                mAreaToTouch.onAfterAreaTouchByEraser(pSceneTouchEvent,
pTouchArea, pTouchAreaLocalX, pTouchAreaLocalY);
                        } else if (toolType == MotionEvent.TOOL_TYPE_STYLUS) {
                                mAreaToTouch.onAfterAreaTouchByPen(pSceneTouchEvent,
pTouchArea, pTouchAreaLocalX, pTouchAreaLocalY);
                        } else if (toolType == MotionEvent.TOOL_TYPE_FINGER) {
                                mAreaToTouch.onAfterAreaTouchByFinger(pSceneTouchEvent,
pTouchArea, pTouchAreaLocalX, pTouchAreaLocalY);
                        }
                }
        }
}
```

[Code 25] Body of AnimatedSpriteWithAreaToTouch class

The sprite is animated via the animate()method as shown in the code below.

```
animatedSpriteWithAreaToTouch.animate(new long[]{100, 100,100}, 3, 5, false, new
OnAnimationFinishedListener(
        new IOnAnimationFinishedListener() {
        @Override
        public void onAnimationFinished(final AnimatedSprite pAnimatedSprite) {
                runOnUpdateThread(new Runnable() {
                        @Override
```

```
                        public void run() {


                        }
                });
        }
}));
```

[Code 26] Sample called of animated method

The first parameter is a table with the durations for frames. The second parameter is the index of the first frame. The third parameter is the index of the last frame. It is important for the number of elements in duration's table to be the same as number of frames. The second and the third parameters can be replaced via a table of frame indexes. The next parameter decides the repetition of the animation – if true, the animation will loop, otherwise it will be played only once. The last argument is an OnAnimationFinishedListener object. The body of this object is shown in the code below.

```
public class OnAnimationFinishedListener implements IAnimationListener {
        private final IOnAnimationFinishedListener mIOnAnimationFinishedListener;
        public interface IOnAnimationFinishedListener {
                void onAnimationFinished(AnimatedSprite pAnimatedSprite);
        }

        public OnAnimationFinishedListener(final IOnAnimationFinishedListener
                pIOnAnimationFinishedListener) {
                mIOnAnimationFinishedListener = pIOnAnimationFinishedListener;
        }

        @Override
        public void onAnimationStarted(final AnimatedSprite pAnimatedSprite, final int
pInitialLoopCount) {
                // Do nothing
        }
        @Override
        public void onAnimationFrameChanged(final AnimatedSprite pAnimatedSprite, final int
pOldFrameIndex, final int pnewFrameIndex) {
                // Do nothing
        }

        @Override
        public void onAnimationLoopFinished(final AnimatedSprite pAnimatedSprite, final int
pRemainingLoopCount, final int pInitialLoopCount) {
                // Do nothing
        }
```

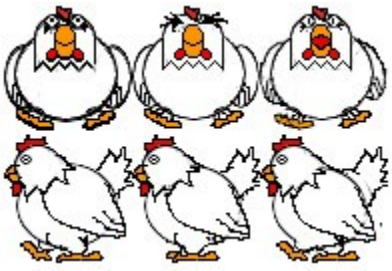
```
        @Override
        public void onAnimationFinished(final AnimatedSprite pAnimatedSprite) {
                mIOnAnimationFinishedListener.onAnimationFinished(pAnimatedSprite);
        }

}
```

[Code 27] Body of OnAnimationFinishListener

The OnAnimationFinishListener implements onAnimationFinshed(...) from an IAnimationListener. It is possible to use the IAnimationListener in the animate() method, and use all the methods, and implement something for every event. The following events are available.

- on animation started;
- on animation frame changed;
- on animation loop finished – available if flag which switch on loop mode is set as true;
- on animation finished.

In order to properly animate a sprite, it has to use the TiledTextureRegion. Below is an example bitmap which can be used in the animation.



[Image 13] Sample bitmap

2.3 Text animations

Apart from the animated sprites there are also animated texts in the HenShooter game. They inform the player about the number of points gained for destroying each hen, reaching the next level and starting a new game. The code responsible for creating and showing animated texts has been put to classes AnimatedText and EffectLayer. The AnimatedText class represents text to be animated and contains definitions of different animation types. Its code is presented below.

```
public class AnimatedText extends Text {

        private static final float START_SCALE_FIRST = 0.0f;
        private static final float END_SCALE_FIRST = 0.7f;
        private static final float START_SCALE_SECOND = 1.0f;
        private static final float END_SCALE_SECOND = 0.7f;

        private static final int Y_VALUE_SHIFT = 10;
```

```
        private final float mDuration;
        private final EffectType mEffectType;

        public enum EffectType {
                DISAPPEAR, DISAPPEAR_UP, LEVEL_UP
        };

        public AnimatedText(final float pX, final float pY, final IFont pFont,  final
CharSequence pText, final TextOptions pTextOptions, final VertexBufferObjectManager
pVertexBufferObjectManager, final EffectType pEffectType, final float pDuration){
                super(pX, pY, pFont, pText, pTextOptions, pVertexBufferObjectManager);

                mEffectType = pEffectType;
                mDuration = pDuration;
        }

        public void initializeEffects(final IEntityModifierListener
pIEntityModifierListener) {
                switch (mEffectType) {
                case DISAPPEAR:
                        setScale(0);
                        registerEntityModifier(
                                new SequenceEntityModifier(pIEntityModifierListener,
                                new ParallelEntityModifier(
                                new ScaleModifier(mDuration, START_SCALE_FIRST,
                                END_SCALE_FIRST),
                                new ScaleModifier(mDuration,
                                START_SCALE_SECOND, END_SCALE_SECOND)),
                                new FadeOutModifier(mDuration)));
                        break;
                case DISAPPEAR_UP:
                        registerEntityModifier(
                                new SequenceEntityModifier(pIEntityModifierListener,
                                new MoveYModifier(mDuration,
                                        mY, mY - Y_VALUE_SHIFT),
                                        new FadeOutModifier(mDuration)));
                        break;
                case LEVEL_UP:
                        setColor(0.17f, 0.54f, 0.34f);
                        setAlpha(0.8f);
                        registerEntityModifier(
                                new SequenceEntityModifier(pIEntityModifierListener,
```

```
                                        new ParallelEntityModifier(
                                        new MoveYModifier(mDuration, mY, mY - 60),
                                        new ScaleModifier(mDuration, 1.0f, 1.3f)),
                                        new FadeOutModifier(mDuration)));
                        break;
                default:
                        break;
                }
        }
}
```

[Code 28] Code of AnimatedText class

AnimatedText inherits behavior from the Text class. Its constructor takes all parameters needed to call the Text's super constructor. The most important ones are pX and pY which define the location of the text and pText which is the text to be presented. The constructor also requires the pEffectType which determines the effect to be applied and pDuration, which defines the duration of the animation. There is also the initializeEffects (final IEntityModifierListener pIEntityModifierListener) method which ensures the initialization of different animation effects. Animation is performed by means of entity modifiers. There are many modifiers available in AndEngine. Each modifier changes some of the entity's properties e.g. size, alpha, position, and color. There are two modifiers which can be used to group other modifiers. One of them is a SequenceEntityModifier whose constructor takes variable amount of modifier objects as parameters. Action defined by means of modifiers passed to it constructor are performed in sequence. The other is of ParallelEntityModifier type which performs actions in parallel. HenShooter makes use of a ScaleModifier to change the object's size. It also uses a FadeOutModifier which simply changes the object's alpha from 1 to 0 in a specified amount of time. All modifiers are registered with a registerEntityModifier(...) method to be used for a given object.

Instances of the AnimatedText class are used in the EffectLayer to prepare and trigger animations. Two methods for animating the labels giving out points are in that class. Their code is presented below.

```
public void prepareAnimatedText(final float pX, final float pY, final Color pColor, final
String pText, final float pDuration, final EffectType pEffectType) {
        final AnimatedText text = new AnimatedText(pX, pY, mBitmapFont,
        pText, new TextOptions(HorizontalAlign.CENTER),
        getVertexBufferObjectManager(), pEffectType, pDuration);
        text.setColor(pColor);
        mAnimatedTexts.add(text);
}

public synchronized void showTexts() {
        if (!mAnimatedTexts.isEmpty()) {
                for (final AnimatedText animatedText : mAnimatedTexts) {
                        animatedText.initializeEffects(mModifierListener);
                        attachChild(animatedText);
                }
```

```
                mAnimatedTexts.clear();
        }
}
```

[Code 29] Methods from EffectLayer which are responsible for text animations

The prepareAnimatedText (...) method simply creates instances of the AnimatedText class with specified parameters (position, font, text, text options, effect type and animation duration) and adds them to the list containing all text animation to be performed. The showTexts() method iterates through all AnimateText objects which were previously added to the aforementioned list and invokes the initalizeEffect() method on them. AndEngine takes care of handling the duration of animation by itself and provides an IEntityModifierListener () which has the onModifierStarted (...) and onModifierFinished () methods. Animation of texts showing information about reaching the next level is implemented in HenShooter in a similar way as described in this section.

3. Gun

HenShooter uses a special gun to shoot hens. Its appearance and mechanics resemble a slingshot. Different hens are used as bullets.

3.1 Gun

The gun implemented in HenShooter imitates the mechanics of a slingshot. The player can shoot bullets by stretching the slingshot's rubber bands while holding the basket by moving the finger down and then releasing it. Depending on the distance from the gun's center, a different force is applied to the bullet's body and it moves with velocity corresponding to the stretch force. There is a special layer - GunLayer – which is intended for holding the gun object with a bullet and queue of three next bullets which will be shot. The Gun object itself is represented as an instance of the Gun class which contains a sprite with gun texture and implements an IAreaToTouch for handling the shots. There is also the GunManager class which manages the GunLayer and passes information about shots to the GameLayer and invokes different animation effects which occur when the bullet collides with the HenTop. The GunLayer's constructor is presented below.

```
public GunLayer(final GameLayer pParentLayer) {
        super();
        mGameLayer = pParentLayer;
        mDefaultBulletXCoord = mGameLayer.getCameraWidth() / 2 - Hen.HALF_HEN_SIZE;
        mDefaultBulletYCoord = mGameLayer.getCameraHeight() - Gun.GUN_SIZE - 25.0f;
        mDefaultWaitingHenYCoord = mGameLayer.getCameraHeight() - Hen.HEN_SIZE - 40.0f;

        mGun = createGun();
        mPocketBody = createGunPocket(mGun);
        mGunManager = new GunManager(mGameLayer);
        mTrajectoryCurve = createTrajectoryCurve();
        mWaitingHens = new IBullet[3];
        createInitialBullets();
}
```

[Code 30] GunLayer's constructor

The presented constructor performs initialization of the gun layer. First it calculates the initial

positions of the bullet in the gun pocket and of the waiting bullets queue. Then it creates an instance of the Gun object class and passes it to the createGunPocket(…) method which is responsible for the gun's mechanics and will be described in greater detail later. Finally it calls createInitialBullets() which creates the first three bullets in the queue.

As mentioned before, the gun mechanics are implemented in the createGunPocket(…) method. Its implementation uses the concept of joints which are provided by AndEngine. Joints simulate physical connections between two bodies. There are many types of joints available in the AndEngine Box2D extension. Each of them behaves in a different way. The Gun in HenShooter is implemented using distance joints, which are one of the simpler joint types. It says that the distance between two bodies must be constant. Additionally the concept of damping is utilized, which allows creating stretchable joints which return to its initial length after being released. Box2D also provides more sophisticated joints e.g. revolute joints which force two bodies to share a common anchor point. Revolute joints are usually used for creating connections where a single degree of freedom is needed. It is also possible to create moving constructions by enabling the joint's motor. Descriptions of other joints can be found in the Box2D manual. A fragment of the createGunPocket(…) method followed by an explanation of used concepts is attached below.

```
private Body createGunPocket(final Gun pGun) {
        final FixtureDef objectFixtureDef =
                PhysicsFactory.createFixtureDef(1, 0.2f, 0f);
        final AnimatedSprite mPocket;
        final Body mLeftLinePin;
        final TiledTextureRegion pocketTextureRegion = (TiledTextureRegion)
                getGameTextureRegion(GameTextureKeys.GUN_POCKET);
        mPocket = new AnimatedSprite(pGun.getGunShotPointX() - 32,
                pGun.getGunShotPointY() + 12, pocketTextureRegion,
                getVertexBufferObjectManager());
        final PhysicsWorld physicsWorld = getPhysicsWorld();
        final Body pocketBody = PhysicsFactory.createBoxBody(physicsWorld,
                mPocket, BodyType.StaticBody,   objectFixtureDef);

        mLeftLinePin = PhysicsFactory.createCircleBody(physicsWorld,
                mPocket.getX() - 18, mPocket.getY() + 4, 0,
                BodyType.StaticBody, objectFixtureDef);

        final Line leftConnectionLine = new Line(mPocket.getX() - 18,
                mPocket.getY() + 4, mPocket.getX(),
                mPocket.getY(), getVertexBufferObjectManager());

        leftConnectionLine.setLinewidth(8.0f);


        attachChild(mPocket);
        attachChild(leftConnectionLine);
```

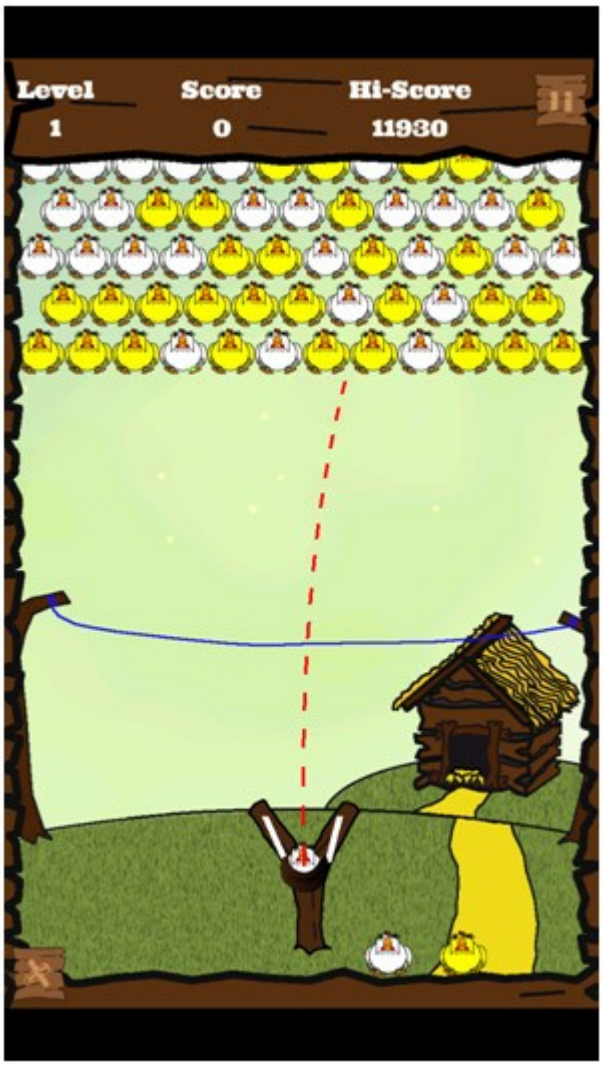
```
        physicsWorld.registerPhysicsConnector(new PhysicsConnector(mPocket,
                pocketBody, true, true) {

                @Override
                public void onUpdate(final float pSecondsElapsed) {
                        super.onUpdate(pSecondsElapsed);
                        final Vector2 movingBodyWorldCenter =
                                pocketBody.getWorldCenter();

        leftConnectionLine.setPosition(leftConnectionLine.getX1(),
                leftConnectionLine.getY1(),
                movingBodyWorldCenter.x *
                PhysicsConstants.PIXEL_TO_METER_RATIO_DEFAULT - 24,
                movingBodyWorldCenter.y *
                PhysicsConstants.PIXEL_TO_METER_RATIO_DEFAULT - 16);

                leftConnectionLine.setZIndex(mPocket.getZIndex() - 1);

                if (mBulletHen != null) {

                mBulletHen.getAnimatedSpriteWithAreaToTouch()
                        .setZIndex(mPocket.getZIndex() - 1);
                        }

                mGun.getGunAnimatedSprite().setZIndex(
                        mPocket.getZIndex() - 2);

                sortChildren();
                }
        });

        final DistanceJointDef leftDistanceJoint = new DistanceJointDef();
        leftDistanceJoint.bodyA = mLeftLinePin;
        leftDistanceJoint.bodyB = pocketBody;
        leftDistanceJoint.frequencyHz = 6f;
        leftDistanceJoint.dampingRatio = 0.01f;

        physicsWorld.createJoint(leftDistanceJoint);

        return pocketBody;
}
```

[Code 31] Fragment of createGunPocket from GunLayer

HenShooter’s gun is implemented in such a way that there are two static points on the sling and one dynamic point on the pocket. Static and dynamic points are connected with each other using distance joints. Left and right joints are implemented in a similar way, therefore the fragment presented above contains only the code responsible for creating the gun and left joint. At the beginning, the sprite for the pocket and body is created, which will represent the gun pocket in the physical world. The sprite’s body is attached to it using the registerPhysicsConnector method. The PhysicsConnector has an onUpdate method which is called each time the body’s position changes. It was used in HenShooter to draw a line connecting the pocket with the gun anchors. The few last lines of the createGunPocket method create a DistanceJoint and set its parameters. First of all the DistanceJointDef is created. It specifies which two bodies are connected by this particular joint and how this joint should behave. Then the joint’s definition is passed to the createJoint method from the PhysicsWorld class which registers that joint in the game’s physics world.

3.2 Trajectory

The trajectory curve shows what will be the trajectory of the bullet shot from the gun. Its purpose is to help the player with aiming. The trajectory curve is implemented in the TrajectoryCurve class which extends the PolyLine from AndEngine. It can be of two colors: red or blue. A red curve indicates that bullet shot from the gun will collide with the HenTop. A blue curve means that bullet will collide with the wall or the floor. While the player is aiming with the gun, the trajectory curve is constantly updated. It takes into account the gravity and wind forces. The process of drawing the curve is pretty simple. It draws a curve point by point, calculating the position of each consecutive point. Points are drawn until the curve reaches the collision point. An image which shows trajectory curve while aiming at the HenTop has been attached below.



[Image 14] Screen with the trajectory curve

3.3 Bullets

As mentioned above, the gun is used for shooting bullets at the HenTop. There can be more than one type of bullet shot by the gun thanks to utilizing the IBullet interface which defines behavior common to all bullets. IBullet's code is presented below. Currently there are 3 types of bullets implemented in the HenShooter game: Hen, BombBullet and ThunderboltBullet.

```
public interface IBullet {

        AnimatedSpriteWithAreaToTouch getAnimatedSpriteWithAreaToTouch();

        Body getBody();

        void setState(BulletState pHenState);

        void setBodyType(BodyType pBodyType);
```

```
        BulletType getBulletType();

        BulletState getState();
}
```

[Code 32] Code of the IBullet interface

Each class which represents the bullet has to implement methods from the IBullet interface or extend AbstractBullet which provides implementation of that interface. The most important method from IBullet is getAnimatedSpriteWithAreaToTouch() which returns the sprite with the appropriate texture depending on the bullet's type. In case of classes like BombBullet and ThunderboltBullet which inherit from AbstractBullet, the creation of an animated sprite is handled by the super class constructor as the following code from BombBullet shows.

```
public class BombBullet extends AbstractBullet {

        public BombBullet(final ILayer pParent, final float pX, final float pY) {
                super(pParent, pX, pY, GameTextureKeys.GUN_BOMB);
        }

        @Override
        public BulletType getBulletType() {
                return BulletType.BOMB;
        }

        @Override
        public Names getObjectName() {
                switch (getState()) {
                case BULLET:
                        return Names.BulletName;
                case FLYING:
                        return Names.FlyingBombName;
                default:
                        return Names.UnKnow;
                }
        }

}
```

[Code 33] Code of BombBullet class

The BombBullet's constructor calls the super constructor and passes a parameter of GameTextureKeys type which determines what texture should be used to create an animated sprite representing that bullet. Additionally two methods from AbstractBullet are overriden. The getBulletType() method returns BulletType.BOMB which indicates the bullet type and is used in the code to implement behavior connected to this particular bullet type e.g. an explosion is shown when a bomb bullet collides with the HenTop. The getObjectName() method returns the object's name

depending on the state the object is currently in.

A hen is special type of bullet, because after collision it changes its state and connects to the HenTop. Due to this its implementation differs from the implementation of other bullet types and therefore is described in a separate section.

4. Hens

Hens are the part of the game which creates its essential sense. During the game, they are constantly descending along with the HenTop. The player's goal is to shoot a hen using the gun in order to destroy hens from the HenTop.

4.1 Hen

A hen is a logical object, which enables management of a single hen object. The hen class implements the IHen interface. The code of this interface is shown below.

```
public interface IHen extends IBullet {

	boolean isConnectedWithTop();
	void moveHenDown(int pY, float pDuration);
	HenColor getHenColor();
	void setKey(int pIncorrectKey);
	void destroyHen();
	AnimatedSpriteWithAreaToTouch getHenAnimatedSprite();
	void correctHenXYPosition(int y, float xPositionF, float yPositionF, float f);
	void setParentLayer(ILayer mHenLayer);
	int getKey();
}
```

[Code 34] Code of IHen interface

A hen object contains the graphical representation of a hen in the form of an AnimatedSprite, which is displayed on the screen. It also has the hen's physical representation i.e. a body object used for calculations in the physics world. The hen's constructor takes the parent layer, coordinates and hens color as parameters as shown in the code below.

```
public Hen(final ILayer pParent, final float pX, final float pY, final HenColor pColor) {
	mParentLayer = pParent;
	mHenColor = pColor;
	final VertexBufferObjectManager vbo = mParentLayer.getVertexBufferObjectManager();
	mHenAnimatedSprite = new AnimatedSpriteWithAreaToTouch(pX + HudLayer.WALL_WIDTH,
pY, getHenTextureRegion(), vbo, this);
	mHenAnimatedSprite.setUserData(this);
	final PhysicsWorld physicsWorld = mParentLayer.getPhysicsWorld();
	mBody = PhysicsFactory.createCircleBody(physicsWorld, mHenAnimatedSprite,
BodyType.StaticBody,  FIXTURE_DEF);
	mBody.setUserData(this);
	physicsWorld.registerPhysicsConnector(new PhysicsConnector(mHenAnimatedSprite,
```

```
mBody, true, true));
        mState = BulletState.NONE;
}
```

[Code 35]Hen’s constructor

Additionally, the hen object contains information about the state of the hen (defines what is currently happening with the hen object), a key (defines the hen’s position on the HenTop) and the hen color. Available colors are defined via the HenColor enum. This enum is shown in the code below.

```
public enum HenColor {
BLACK(Color.BLACK), WHITE(Color.WHITE), GRAY(new Color(0.5f, 0.5f, 0.5f)), RED(Color.RED),
ORANGE(new Color(1.0f,0.6f, 0.0f)), YELLOW(Color.YELLOW);

private Color mColor;

private HenColor(final Color pColor) {
        mColor = pColor;
}

public Color getColor() {
        return mColor;
}

public static HenColor randomStandardHenColor() {
        final EnumSet standardHens;
        final int level = AppCache.getInstance().getInt(AppCacheKey.CURRENT_LEVEL);
        switch (level) {
        case  1:
        case  2: {
                standardHens = EnumSet.of(HenColor.YELLOW, HenColor.WHITE);
                break;
        }
        case  3:
        case  4: {
                standardHens = EnumSet.of(HenColor.YELLOW, HenColor.WHITE, HenColor.GRAY);
                break;
        }
        case  5:
        case  6: {
                standardHens = EnumSet.of(HenColor.YELLOW, HenColor.WHITE, HenColor.GRAY,
HenColor.RED);
                break;
        }
```

```
        case  7:
        case  8: {
                standardHens = EnumSet.of(HenColor.YELLOW, HenColor.WHITE, HenColor.GRAY,
HenColor.RED, HenColor.ORANGE);
                break;
        }
        default: {
                standardHens = EnumSet.of(HenColor.YELLOW, HenColor.WHITE, HenColor.GRAY,
HenColor.RED, HenColor.ORANGE, HenColor.BLACK);
        }
}

        final HenColor[] values = standardHens.toArray(new HenColor[standardHens.size()]);
        return pickColor(values);
}
private static HenColor pickColor(final HenColor[] values) {
        final int size = values.length;
        final int pick = new SecureRandom().nextInt(size);
        return values[pick];
}
}
```

[Code 36]HenColor enum

As presented above, HenColor is dependent of the game level. On each level there is a bigger amount of hen's colors available. A Hen object is constructed via the HenFactory. The HenFactory class has two static methods. The first method is used to create a Hen on the HenTop. The second method is used to create the Hen as a bullet. The HenFactory class is shown in the code below

```
public final class HenFactory {
        private HenFactory() {

        }
        public static IHen createHenToConnectWithHenTop(final HenLayer pHenLayer, final
float pX, final float pY, final HenColor pColor) {
                final Hen result = new Hen(pHenLayer, pX, pY, pColor);
                result.setState(BulletState.TO_CONNECT);
                result.setBodyType(BodyType.StaticBody);
                return result;
        }
        public static IHen createHenToBullet(final GunLayer pGunLayer, final float pX,
final float pY, final HenColor pColor) {
                final IHen result = new Hen(pGunLayer, pX, pY, pColor);
                result.setBodyType(BodyType.DynamicBody);
                result.setState(BulletState.BULLET);
```

```
                return result;
        }
}
```

[Code 37]HenFactory class

4.2 HenTop

The HenTop is an object which collects hens and allows their management. The HenTop class implements the IHenTop interface. The code of this interface is shown below.

```
public interface IHenTop {
        IHen getHen( int pX, int pY);
        void putHen(int pX, int pY, IHen pHen);
        void removeHen(int pX, int pY);
        void clearHenTop();
        IHen getHenByKey( int pKey);
        void removeHenByKey(int pKey);
        void showInLog();
}
```

[Code 38] IHenTop interface

As presented, it is possible to add, get and remove hens via the HenTop API. The HenTop can have a maximum of eleven rows and twelve columns. Hens are collected in a SparseArray. In one row there are eleven or twelve hens. If there is no shift in the row, there are twelve hens, otherwise there are eleven (the last hen is null). Every Hen on the HenTop has a key which defines its position.

4.3 HenTopManager

The HenTopManager is an object which manages the HenTop. In it all actions are handled, such as collisions and calculating which hens should be removed from the HenTop. The main task of the HenTopManager is filling the HenTop on game start. Part of the code responsible for this functionality is presented below.

```
public synchronized voidfillHenTop(final int pCount) {
        for (int y = 0; y < pCount; y++) {
                final int shift = y % 2 == 0 ? 0 : Hen.HALF_HEN_SIZE;
                final int shiftCol = y % 2 == 0 ? 0 : 1;
                for (int x = 0; x < HenTop.COL_COUNT - shiftCol; x++) {
                    final IHen hen = HenFactory.createHenToConnectWithHenTop(mHenLayer, x
* Hen.HEN_SIZE + shift,HudLayer.CEILING_WIDTH + (y - 1) * Hen.HEN_SIZE,
HenColor.randomStandardHenColor());
                    mHenTop.putHen(x, y, hen);
                    mHenLayer.registerTouchArea(hen.getHenAnimatedSprite());
                    mHenLayer.attachChild(hen.getHenAnimatedSprite());
                    hen.setState(BulletState.CONNECTED);
                    // Animations
                    final long firstFrameDuration = 900;
```

```
                    final long secondFrameDuration = 100;
                    final long[] tabOfDuration = new long[] { firstFrameDuration,
secondFrameDuration };
                    hen.getHenAnimatedSprite().animate(tabOfDuration, new int[] { 0, 1 },
true);
                    final int currentLevel =
AppCache.getInstance().getInt(AppCacheKey.CURRENT_LEVEL);
                    final float gameSpeed = GameUtils.getSpeedForLevel(currentLevel);
                        hen.moveHenDown(y, gameSpeed);
                }
        }

        mShift = pCount % 2 == 0;
}
```

[Code 39] Method which creates a few first rows of HenTop

Next, the HenTopManager adds new rows of hens. The following method performs adding.

```
public synchronized boolean addNewRowOfHens(final float pDuration) {
        for (int y = HenTop.ROW_COUNT - 1; y >= 0; y--) {
                for (int x = HenTop.COL_COUNT - 1; x >= 0; x--) {
                        final IHen hen = mHenTop.getHen(x, y);
                        if (hen != null) {
                                if (y == HenTop.ROW_COUNT - 1) {//Collection is already
full - can no longer add anything.
                                        return false;
                                }
                                hen.setBodyType(BodyType.StaticBody);
                                hen.moveHenDown(y, pDuration);
                                mHenTop.putHen(x, y + 1, hen);
                                mHenTop.removeHen(x, y);
                        }
                }
        }
        final int shift = mShift ? 0 : Hen.HALF_HEN_SIZE;
        final int shiftCol = mShift ? 0 : 1;
        for (int x = 0; x < HenTop.COL_COUNT - shiftCol; x++) {
                final int xPosition = x * Hen.HEN_SIZE + shift;
                final int yPosition = HudLayer.CEILING_WIDTH - Hen.HEN_SIZE;
                final HenColor henColor = HenColor.randomStandardHenColor();
                final IHen hen = HenFactory.createHenToConnectWithHenTop(mHenLayer,
xPosition, yPosition, henColor);
```

```
                mHenTop.putHen(x, 0, hen);
        }
        mShift = !mShift;
        return true;
}
```

[Code 40] Method which adds a new row of hens

The HenTopManager also supports actions performed after collision. This operation is very difficult and requires using methods from the HenUtils class. When fired, a Hen collides with the HenTop, and the addHenAfterShot method is called. The body of this method is shown in the code below.

```
public synchronized int addHenAfterShot(final Contact pContact) {
        final IHen henA = (IHen) pContact.getFixtureA().getBody().getUserData();
        final IHen henB = (IHen) pContact.getFixtureB().getBody().getUserData();

        if (!henB.isConnectedWithTop() && !henA.isConnectedWithTop()) {
                return Hen.INCORRECT_KEY;
        }
        final Vector2[] pointsOfContact = pContact.getWorldManifold().getPoints();
        final IHen henOnTop = henA.isConnectedWithTop() ? henA : henB;
        final IHen bulletHen = henA.isConnectedWithTop() ? henB : henA;

        if (Float.isNaN(pointsOfContact[0].x) || Float.isNaN(pointsOfContact[0].y)) {
                henOnTop.setState(BulletState.NOT_CONNECTED);
                return Hen.INCORRECT_KEY;
        }
        final int key = henOnTop.getKey();

        final HenNeighborDirection neighborDirection =
                HenUtils.findNeighborDirectionByCollisionPoints(pointsOfContact,henOnTop);
        final int neighborKeyByDirection = HenUtils.getNeighborKeyByDirection(key,
neighborDirection, mHenTop);
        if (neighborKeyByDirection == Hen.INCORRECT_KEY) {
                return neighborKeyByDirection;
        }
        final IHen checkedHen = mHenTop.getHenByKey(neighborKeyByDirection);
        if (checkedHen != null) {
                return Hen.INCORRECT_KEY;
        }
        final PointF point = HenUtils.getNeighborCoordinatesByDirection(neighborDirection,
henOnTop);
        final int x = neighborKeyByDirection % HenTop.COL_COUNT;
        final int y = (neighborKeyByDirection - x) / HenTop.COL_COUNT;
```

```
        final float xPositionF = point.x;
        final float yPositionF = point.y;
        bulletHen.setBodyType(BodyType.StaticBody);
        bulletHen.setState(BulletState.CONNECTED);
        mHenTop.putHen(x, y, bulletHen);
        mHenLayer.runOnUpdateThread(new Runnable() {
                // This is the main update method moving the henTop.
                @Override
                public void run() {
                        final IHen hen = getHenFromHenTop(x, y);
                        final AnimatedSpriteWithAreaToTouch henSprite =
hen.getHenAnimatedSprite();
                        henSprite.detachSelf();
                        hen.setParentLayer(mHenLayer);
                        mHenLayer.attachChild(henSprite);
                        final long[] tabOfDuration = new long[] { 900, 100 };
                        henSprite.animate(tabOfDuration, new int[] { 0, 1 }, true);
                        hen.correctHenXYPosition(y, xPositionF, yPositionF, 0.01f);
                        mHenLayer.registerTouchArea(hen.getHenAnimatedSprite());
                        deleteHenAfterConnecting(x, y);
                        mHenLayer.removeUnconnectedHens();
                }
        });
        return neighborKeyByDirection;
}
```

[Code 41] Method called after collision

The method presented above uses the following method to get information about the point where the collision between the fired bullet and the HenTop occurred.

```
public static HenNeighborDirection findNeighborDirectionByCollisionPoints(final Vector2[]
pPoints, final IHen pHenOnTop) {
                final float xHenOnHenTop = pHenOnTop.getBody().getPosition().x;
                final float yHenOnHenTop = -pHenOnTop.getBody().getPosition().y;
                float xCollisionPoint = -1f;
                float yCollisionPoint = -1f;
                for (final Vector2 p : pPoints) {
                        if (Math.abs(p.x) > 0 && Math.abs(p.y) > 0 && !
Float.valueOf(p.x).isNaN() && !Float.valueOf(p.y).isNaN()) {
                                xCollisionPoint = p.x;
                                yCollisionPoint = -p.y;
                                break;
                        }
```

```
                }
                if (Float.compare(xCollisionPoint, -1f) == 0) {
                        throw new IllegalStateException("Illegal state of cooridinates
[x].");
                }
                if (Float.compare(yCollisionPoint, -1f) == 0) {
                        throw new IllegalStateException("Illegal state of cooridinates
[y].");
                }
                final float xCenter = 0.0f;
                final float yCenter = 0.0f;
                final float xBullet = xCollisionPoint - xHenOnHenTop;
                final float yBullet = yCollisionPoint - yHenOnHenTop;
                if (Float.compare(yBullet, 0.0f) == 0) {
                        // 0 or 180
                        return xBullet > 0 ? HenNeighborDirection.E :
HenNeighborDirection.W;
                }
                final float ctgFun = (xCenter - xBullet) / (yCenter - yBullet);
                final float ctg30 = 1.732050808f;
                final float tg90 = 0.0f;
                final float tg150 = -1.732050808f;
                if (ctgFun > ctg30) {
                        return xBullet > 0 ? HenNeighborDirection.E :
HenNeighborDirection.W;
                }
                if (ctgFun >= tg90) {
                        return yBullet > 0 ? HenNeighborDirection.NE :
HenNeighborDirection.SW;
                }
                if (ctgFun >= tg150) {
                        return yBullet > 0 ? HenNeighborDirection.NW :
HenNeighborDirection.SE;
                }
                return xBullet > 0 ? HenNeighborDirection.E : HenNeighborDirection.W;
        }
```

[Code 42] Method for finding direction from collision point

As presented above, this method analyzes the point of contact and returns one of the directions which are defined in the HenNeighbor enum. This enum is shown in the following code.

```
public enum HenNeighborDirection {
        NW, NE, E, W, SE, SW,;
```

```
        }
}
```

[Code 43] Enum HenNeighborDirection.

In the next step the key from the direction is calculated with the help of the following method.

```
public static int getNeighborKeyByDirection(final int pKey, final HenNeighborDirection
pNeighborDirection, final IHenTop henTop) {
                final int x = pKey % HenTop.COL_COUNT;
                final int y = (pKey - x) / HenTop.COL_COUNT;
                final boolean isRowShifted = HenUtils.isRowShiftedAt(henTop, y);
                if (y == 0) {
                        return Hen.INCORRECT_KEY;
                }
                final int keyNeigbort;
                switch (pNeighborDirection) {
                case  NW: {
                        keyNeigbort = (y - 1) * HenTop.COL_COUNT + x - (isRowShifted ?
0 : 1);
                        return keyNeigbort;
                }
                case  NE: {
                        keyNeigbort = (y - 1) * HenTop.COL_COUNT + x + (isRowShifted ?
1 : 0);
                        return keyNeigbort;
                }
                case  E: {

                        if (isRowShifted && x == HenTop.COL_COUNT - 2 || !isRowShifted &&
x == HenTop.COL_COUNT - 1) {
                                return Hen.INCORRECT_KEY;
                        }
                        keyNeigbort = y * HenTop.COL_COUNT + x + 1;
                        return keyNeigbort;
                }

                case  SE: {
                        if (!isRowShifted && x == HenTop.COL_COUNT - 1) {
                                return Hen.INCORRECT_KEY;
                        }
                        if (y == HenTop.ROW_COUNT - 1) {
                                return Hen.INCORRECT_KEY;
                        }
```

```
                        keyNeigbort = (y + 1) * HenTop.COL_COUNT + x + (isRowShifted ?
1 : 0);
                        return keyNeigbort;
                }
                case  SW: {
                        if (x == 0 && !isRowShifted) {
                                return Hen.INCORRECT_KEY;
                        }
                        if (y == HenTop.ROW_COUNT - 1) {
                                return Hen.INCORRECT_KEY;
                        }
                        keyNeigbort = (y + 1) * HenTop.COL_COUNT + x - (isRowShifted ?
0 : 1);
                        return keyNeigbort;
                }
                case  W: {
                        if (x == 0) {
                                return Hen.INCORRECT_KEY;
                        }
                        keyNeigbort = y * HenTop.COL_COUNT + x - 1;
                        return keyNeigbort;
                }
                default: {
                        throw new IllegalStateException(UNSUPPORTED_STATE_EXCEPTION);
                }
        }
}
```

[Code 44] Body of the method which returns neighbor key by direction

In the next step the coordinates of neighbors by direction are calculated with the help of the following method.

```
public static PointF getNeighborCoordinatesByDirection(final HenNeighborDirection direction,
final IHen hen) {
                final PointF result = new PointF();
                final AnimatedSpriteWithAreaToTouch body = hen.getHenAnimatedSprite();
                final float xCenter = body.getX();
                final float yCenter = body.getY();
                switch (direction) {
                case  NW: {
                        result.x = xCenter - Hen.HALF_HEN_SIZE;
                        result.y = yCenter - Hen.HEN_SIZE;
                        return result;
```

```
                }
                case  NE: {
                        result.x = xCenter + Hen.HALF_HEN_SIZE;
                        result.y = yCenter - Hen.HEN_SIZE;
                        return result;
                }
                case  E: {
                        result.x = xCenter + Hen.HEN_SIZE;
                        result.y = yCenter;
                        return result;
                }
                case  SE: {
                        result.x = xCenter + Hen.HALF_HEN_SIZE;
                        result.y = yCenter + Hen.HEN_SIZE;
                        return result;
                }
                case  SW: {
                        result.x = xCenter - Hen.HALF_HEN_SIZE;
                        result.y = yCenter + Hen.HEN_SIZE;

                        return result;
                }
                case  W: {
                        result.x = xCenter - Hen.HEN_SIZE;
                        result.y = yCenter;
                        return result;
                }
                default: {
                        throw new IllegalStateException(UNSUPPORTED_STATE_EXCEPTION);
                }
                }
        }
```

[Code 45] Body of the method which returns coordinates of neighbor by direction

In the last step, the hen is added to the HenTop in the update thread. Its position is corrected to reflect the structure of the HenTop. The Hen's sprite is animated and a physical body is registered. In the end, hens which remain not connected to the HenTop are removed from the scene. The DFS algorithm is used for determining which hens should be removed. It utilizes the following method for finding out which hens are connected to the HenTop.

```
private void createCollectionOfKeyResult(final int key, final SparseArray<List<Integer>>
relationList, final SparseArray<Boolean> visited, final AbstractSet<Integer> keyResult) {
        if (visited.get(key) == null) {
                visited.put(key, true);
```

```
                keyResult.add(key);
                final List<Integer> neighborhoods = relationList.get(key);
                for (final int neighborhood : neighborhoods) {
                        createCollectionOfKeyResult(neighborhood, relationList, visited,
keyResult);
                }
        }
}
```

[Code 46] Method where the DFS algorithm is used

After creating a collection of hens to be connected, another collection is created. This collection contains all the hens which should be removed from the HenTop. A second collection contains all hens which are not attached to the HenTop. Thanks to this solution, all unconnected hens disappear from the game scene. This concept is contained in the method presented below.

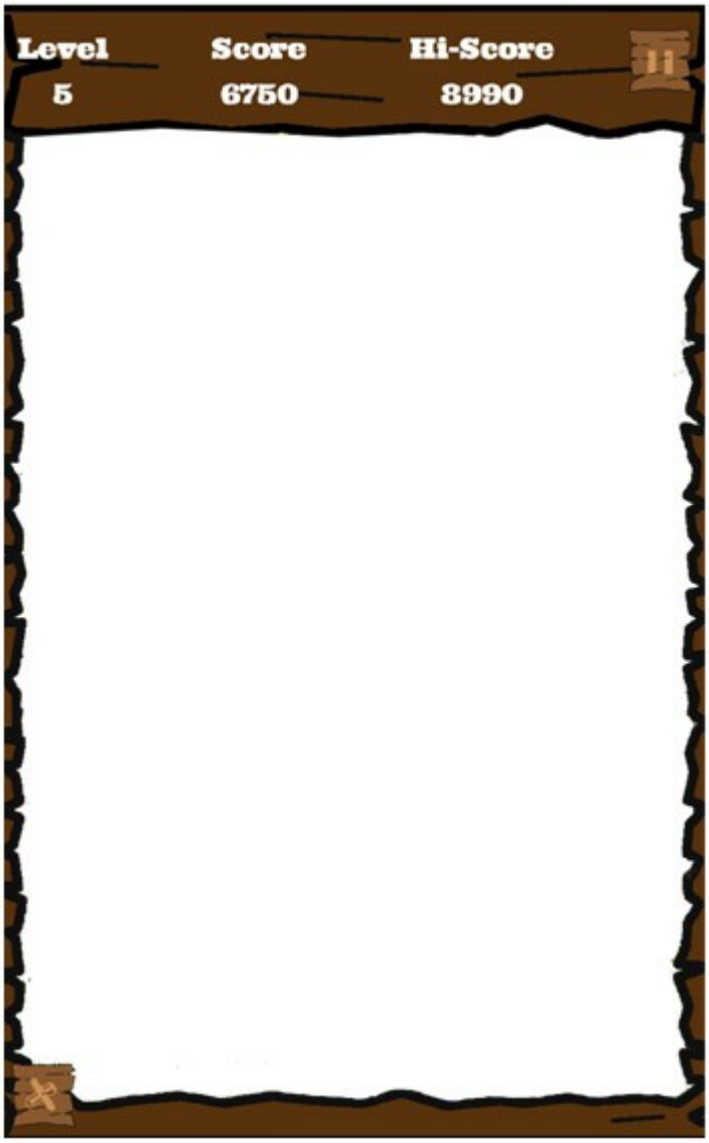
```
public synchronized void removeHenExceptFor(final AbstractSet<integer> key) {
                for (int x = 0; x < HenTop.COL_COUNT; x++) {
                        for (int y = 0; y < HenTop.ROW_COUNT; y++) {
                                final int keyToCheck = y * HenTop.COL_COUNT + x;
                                if (!key.contains(keyToCheck)) {
                                        final IHen hen = mHenTop.getHen(x, y);
                                        if (hen != null) {
                                                if (hen.isConnectedWithTop()) {

hen.setState(BulletState.NOT_CONNECTED);

hen.setKey(Hen.INCORRECT_KEY);
                                                        mHensToRemove.add(hen);
                                                }
                                                mHenTop.removeHen(x, y);
                                        }
                                }
                        }
                }
        }
```

[Code 47] Method where DFS algorithm is used

Animated sprites and physical bodies of removed hens are being disposed of in the update thread.

5. HUD

One of the most important elements of the HenShooter game is the HUD. It shows information about the state of the game i.e. the number of points gathered by the user, the high score and the current level. It also has two buttons: one is for in-game pause and the other is for enabling the S Pen feature. The HUD is represented as a layer added to the game scene. Its whole functionality is contained in the HudLayer class which implements the ILayer interface.



[Image 15] Screen with HUD layer visible

5.1 Level, score and high-score

The top part of the HUD layer presents information about the level, current score and high-score. Its content is updated during the game. There is an excerpt from HUDLayer class below presenting the implementation of information updating. It is followed by an explanation of the applied concept.

```
public class HudLayer extends Entity implements ILayer {

        private final GameLayer mGameLayer;
        private final Text mLevel;
        private final Text mScore;
        private final Text mHighScore;
        private final Text mLevelLabel;
```

```
        private final Text mScoreLabel;
        private final Text mHighScoreLabel;

        public HudLayer(final GameLayer pGameScene) {
                mGameLayer = pGameScene;

                AppCache.getInstance().putInt(AppCacheKey.CURRENT_SCORE, 0);
                AppCache.getInstance().putInt(AppCacheKey.CURRENT_LEVEL, 1);
                final int newNextLevelScore = GameUtils.getScoreForLevel(2);
                AppCache.getInstance().putInt(AppCacheKey.NEXT_LEVEL_SCORE,
                        newNextLevelScore);
                AppCache.getInstance().putInt(AppCacheKey.NEXT_LEVEL_SCORE,
                        newNextLevelScore);

                setHighScore();

                //Here is code responsible for HUD elements initialization.
        }

        @Override
        public void onUpdateLayer() {
                final int currentScore =
                        AppCache.getInstance().getInt(AppCacheKey.CURRENT_SCORE);
                final int nextLevelScore =
                        AppCache.getInstance().getInt(AppCacheKey.NEXT_LEVEL_SCORE);
                if (currentScore >= nextLevelScore) {
                        final int currentLevel =
                        AppCache.getInstance().getInt(AppCacheKey.CURRENT_LEVEL) + 1;

                        mGameLayer.showLevelUpText(currentLevel);
                        AppCache.getInstance().putInt(AppCacheKey.CURRENT_LEVEL,
                                currentLevel);
                        final int newNextLevelScore =
                                GameUtils.getScoreForLevel(currentLevel + 1);
                        final float gameSpeed =
                                GameUtils.getSpeedForLevel(currentLevel);
                        mGameLayer.setGameSpeed(gameSpeed);
                        AppCache.getInstance().putInt(AppCacheKey.NEXT_LEVEL_SCORE,
                                newNextLevelScore);
                        if (!
AppCache.getInstance().getBoolean(AppCacheKey.MODE_REMOVE_HEN_BY_PEN))     {
                                mSPenButton.setState(ToggleState.OFF);
                        }
```

```
                }
                final int currentLevel =
AppCache.getInstance().getInt(AppCacheKey.CURRENT_LEVEL);
                setScore(currentScore);
                setLevel(currentLevel);
                final int highScore =
AppCache.getInstance().getInt(AppCacheKey.HIGH_SCORE);
                if (highScore < currentScore) {
                        setHighScore(currentScore);
                }
        }

        private void setHUDNumberValues() {
                setLevel(1);
                setScore(0);
                final int highScore =
                        AppCache.getInstance().getInt(AppCacheKey.HIGH_SCORE);
                setHighScore(highScore);
        }
}
```

[Code 48] HUDLayer's fragment concerning information updating

The code responsible for updating the information resides inside the onUpdateLayer () method. This method is called from inside the GameLayer's onUpdateLayer () which is invoked from the game's update thread. Information about the score and game level are stored in the application cache. They are saved to the currentScore and currentLevel variables. The application cache also contains the score thresholds and game speed for each level. A check is performed on each update which compares the player's score with the next level threshold. If the player gathers an appropriate amount of points, the level is increased and the information presented in the HUD is refreshed. The game speed is also increased with each next level by calling the setGameSpeed () method on the mGameLayer object. If the player manages to beat the high score, his current score is presented as the high score and it is updated each time he gets some points. At the beginning of the game the HUD values are set to default by calling setHudNumberValues ().

5.2 S Pen button

There is a special S Pen button on the bottom of the HUD layer. It is used to enable S Pen mode. When the game is in this mode there is a possibility to delete hens by clicking them with the S Pen stylus. After 10 seconds from button click, the game returns to normal mode. The player is allowed to use this button only once on each level. The player gets the same amount of points for the hen destroyed in that way as if it was destroyed by connecting at least 3 hens of the same color. The S Pen button is an instance of ToggleButtonSprite which was described earlier in this guide. The code fragment below shows the implementation of this button's behavior.

```
mSPenButton.setOnToggleClickListener(new OnToggleClickListener() {
```

```
        @Override
        public void onOnClick(final ToggleButtonSprite pButtonSprite,
                final float pTouchAreaLocalX,
                final float pTouchAreaLocalY) {
                mGameLayer.getAudioPlayer().play(SoundKeys.BUTTON_PRESS);

                AppCache.getInstance().putInt(AppCacheKey.MODE_REMOVE_HEN_BY_PEN, 0);
                mSPenButton.setEnabled( false);
        }

        @Override
        public void onOffClick(final ToggleButtonSprite
                pButtonSprite, final float pTouchAreaLocalX,
                final float pTouchAreaLocalY) {
                mGameLayer.getAudioPlayer().play(SoundKeys.BUTTON_PRESS);

                AppCache.getInstance().putInt(AppCacheKey.MODE_REMOVE_HEN_BY_PEN, 1);
                final TimerHandler th = new TimerHandler(10, false,
                new ITimerCallback() {

                        @Override
                        public void onTimePassed(final TimerHandler
                        pTimerHandler) {
                                AppCache.getInstance()
                                .putInt(AppCacheKey.MODE_REMOVE_HEN_BY_PEN, 0);
                                mSPenButton.setEnabled( false);
                        }
                });
                mGameLayer.registerUpdateHandlerOnScene(th);

        }
});

registerTouchArea(mSPenButton)
```

[Code 49] Code for handling S Pen button clicks

Actions performed on S Pen button click are handled by the OnToggleClickListener which has two methods: onOnClick () and onOffClick (). They are invoked in turns depending on the internal state of the toggle button from the moment when the click occurred. The onOffClick () method is called when the player clicks the S Pen button to go to S Pen mode. It sets the application cache value MODE_REMOVE_HEN_BY_PEN to 1 which allows destroying hens by clicking them with the stylus. A TimerHandler is also created which invokes the onTimePassed () method after 10 seconds from an anonymous callback class. The mentioned callback method simply disables both S Pen mode and the S Pen button.

Ref:http://developer.samsung.com/android/technical-docs/Hen-Shooter-Chapter-2-Game-Implementation