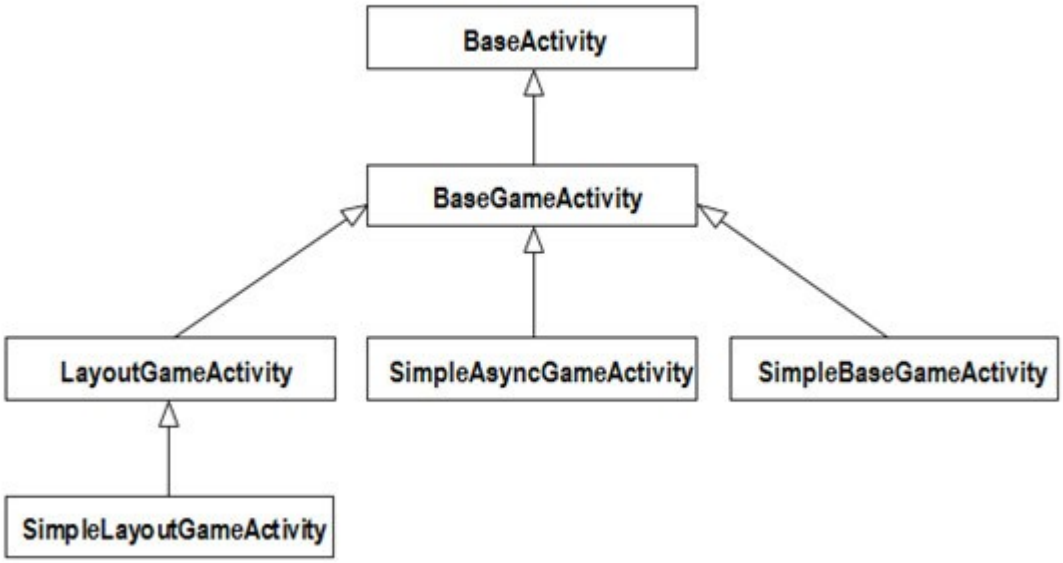
1. Hen Shooter game concept

Hen Shooter is a game in the Bubble Shooter style. The main purpose of the game is shooting hens from the hen top. If the player connects at least three hens of the same color to each other, they are removed from the hen top and the player receives points. Points earned by the player allow him to go to the higher levels. The difficulty of the game increases with each level. On each level the number of available hen colors and the speed of the game are increased. In addition, the player may use the various bonuses, such as bombs, bombs with thunderbolts or delete hens using the S Pen stylus for a limited amount of time. The game takes place in a physical world where forces operate on object's bodies. The body can collide and bounce off other bodies. Wind is also implemented – to influence the flight trajectory. The game ends when the hen top comes to contractual line. If the player beats the highest score, he must enter his name to be displayed on the scoreboard. The player has the possibility to stop the game by pressing the pause button or the back button. The player has no way of saving the game. After leaving the application and re-opening it, the game starts over.

2. Extending BaseGameActivity

The AndEngine framework provides its own subclasses of Android Activity. They are located in the org.andengine.ui.activity package. The main class, which inherits directly from Activity, is called BaseActivity. The rest of the classes are direct or indirect subclasses of the BaseActivity class. The hierarchy of BaseActivity inheritance is shown in the following diagram.



[Image 2] Inheritance tree of the BaseActivity

Selection of the base class is very important and should depend on the nature of the application and planned functionalities. For a simple game the most appropriate one would be SimpleBaseGameActivity. If android UI elements are required, the LayoutGameActivity or SimpleLayoutGameActivity classes should be used. The BaseGameActivity class has been used as the basic activity class for the Hen Shooter game.

2.1 Methods

When extending the BaseGameActivity class, we have to override four fundamental methods: onCreateEngineOptions (), onCreateResources (), onCreateScene () and onPopulateScene() . An example is shown in the following code.

```
public class HenShooterActivity extends BaseGameActivity {
    @Override
    public EngineOptions onCreateEngineOptions() {
        return null;
    }
    @Override
    public void onCreateResources(OnCreateResourcesCallback pOnCreateResourcesCallback)
throws Exception {
    }
    @Override
    public void onCreateScene(OnCreateSceneCallback pOnCreateSceneCallback) throws
Exception {
    }
    @Override
    public void onPopulateScene(Scene pScene, OnPopulateSceneCallback
pOnPopulateSceneCallback) throws Exception {
    }
}
```

[Code 1] Sample game activity which extends BaseGameActivity

The methods presented above are invoked in the following order:

- onCreateEngineOptions() – called in onCreate(Bundle);
- onCreateResources(OnCreateResourcesCallback) – called in onCreateGame();
- onCreateScene(OnCreateSceneCallback) – called after onCreateResources(OnCreateResourcesCallback), in callback;
- onPopulateScene(Scene, OnPopulateSceneCallback) – called in callback, after onCreateScene(OnCreateSceneCallback);

onCreateEngineOptions()

In this method the engine options are created. The creation of the options used in HenShooter is shown in the following code:

```
@Override
public EngineOptions onCreateEngineOptions() {
        mCamera = new Camera(0, 0, SCREEN_WIGHT, SCREEN_HIGHT);
        final EngineOptions engineOptions = new EngineOptions(true,
        ScreenOrientation.PORTRAIT_FIXED,
        new RatioResolutionPolicy(SCREEN_WIGHT, SCREEN_HIGHT), mCamera);
        engineOptions.getAudioOptions().setNeedsSound(true).setNeedsMusic(true);
```

```
        return engineOptions;
}
```

[Code 2] Code for creating engine options.

In line 93 a camera object is created. Position X, position Y, width and height respectively are passed as constructor parameters. In the line below the constructor, an engine options object is created. The first parameter specifies whether the camera should be in full screen mode – true enables full screen mode, false disables it. The second parameter, which is of ScreenOrientation type, specifies the orientation of screen. ScreenOrientation is an enum which has four fields:

- •LANDSCAPE_FIXED;
- •LANDSCAPE_SENSOR;
- •PORTRAIT_FIXED;
- •PORTRAIT_SENSOR;

The next parameter after screen orientation defines the screen resolution in pixels. The last parameter is the camera object created previously.
Additionally, in line 98 there are modified engine's audio options to enable sound and music, because HenShooter will be using sounds effects.

onCreateResources(OnCreateResourcesCallback)

This method creates and loads game resources. Sample code which shows how to create resource is presented below:

```
@Override
        public void onCreateResources() {
        BitmapTextureAtlasTextureRegionFactory.setAssetBasePath( "gfx/");
        this.mBitmapTextureAtlas = new
        BitmapTextureAtlas( this.getTextureManager(), 256, 128,
        TextureOptions.BILINEAR);
        this.mSampleTextureRegion1 =
        BitmapTextureAtlasTextureRegionFactory.createFromAsset(
        this.mBitmapTextureAtlas, this, "sample_1.png", 0, 0);
        this. mSampleTextureRegion2 =
        BitmapTextureAtlasTextureRegionFactory.createFromAsset(
        this.mBitmapTextureAtlas, this, "sample_2.png", 0, 50, 1, 5);
        this.mBitmapTextureAtlas.load();
}
```

[Code 3] Code for creating resources.

The path to assets is set in line 93. The next step is creating the bitmap texture atlas. This object's constructor takes the following parameters: TextureAtlas, width, height and TextureOptions. After that, the texture regions are created. This object is created with information such as: bitmap texture atlas, context, width, height, and if there are a few frames, information about columns and rows count. In the last step, we load the bitmap texture atlas. It is not necessary that all game resources are fully loaded in this method.

onCreateScene(OnCreateSceneCallback)
In this method the game scene is created. A scene object can be created by calling the constructor without parameters from the Scene class.

onPopulateScene(OnPopulateSceneCallback)
This method is called last. All actions which have to be done after scene creation are implemented in this method. For example in Hen Shooter this method loads the remaining resources and creates a timer which shows the next scene.

3. Scene manager

The scene manager is an object of the SceneManager class which creates and manages scenes. It is constructed using three objects of the following types: HenShooterActivity, Engine and Camera. All scenes used in the application are created in this object inside createXXX() (where XXX is name of scene) methods. Additionally, resources for all scenes are loaded in scene manager. The current scene is set via the setCurrentScene(SceneType) method. This method's body is shown in the following code.

```
public void setCurrentScene(final SceneType pScene) {
        final AudioPlayer audioPlayer = mHenShooterActivity.getAudioPlayer();
        switch (pScene) {
                case SPLASH:
                        break;
                case MENU:
                        audioPlayer.stop(MusicKeys.GAME_AMBIENT);
                        audioPlayer.stop(MusicKeys.GAME_MUSIC);
                        audioPlayer.clearPaused();
                        audioPlayer.play(MusicKeys.MENU_MUSIC, true);
                        if (mMenuScene == null) {
                                mMenuScene = createMenuScenes();
                        }
                        if (mGameScene != null) {
                                mGameScene = null;
                        }
                        mEngine.setScene(mMenuScene);
                        break;
                case GAME:
                        audioPlayer.stop(MusicKeys.MENU_MUSIC);
                        audioPlayer.clearPaused();
                        audioPlayer.play(MusicKeys.GAME_MUSIC, true);
                        audioPlayer.play(MusicKeys.GAME_AMBIENT, true);

                        if (mGameScene == null) {
                                mGameScene = createGameScenes();
                        }
                        if (mGameScene.isInGameMenuOpen()) {
```

```
                                mGameScene.clearChildScene();
                                mGameScene.setInGameMenuIsOpen(false);
                                mInGameMenuScene.reset();
                        } else {
                                mEngine.setScene(mGameScene);
                        }
                        break;
                case GAME_MENU:
                        if (mInGameMenuScene == null) {
                                mInGameMenuScene = createInGameMenuScenes();
                        }
                        mGameScene.setChildScene(mInGameMenuScene, false,
                        true, true);
                        mInGameMenuScene.registerAlphaModifierForAllButtons();
                        mGameScene.setInGameMenuIsOpen(true);
                        break;
                case SCORE:
                        if (mGameScene != null) {
                                mGameScene = null;
                        }
                        mScoreScene = createScoreScenes();
                        mEngine.setScene(mScoreScene);
                        break;
                default: {
                        throw new IllegalStateException();
                        }
        }
        mCurrentScene = pScene;
}
```

[Code 4] Code to create resources

The presented method uses the lazy initialization concept while setting up the current scene. It means that the creation of objects of a particular scene type is deferred to the moment when that scene is set as the current game scene. If some type of scene is set as current for the second time, its object is cleared instead of creating a new instance of the same type. Additionally, this method is responsible for managing resources such as music and sounds for the current scene.

3.1 What is scene?

A scene is an entity that all layers are put into. The AndEngine framework provides the Scene object from which scenes used in Hen Shooter inherit. There are registered update handlers such as the TimerHandler, PhysicsWorld or BaseEntityUpdateHandler on the scene. Additionally, it is possible to set listeners such as IOnAreaTouchListener and IOnSceneTouchListener.

3.2 What is layer?

A layer is an entity where all the sprites are put. It contains all the code responsible for managing its

functionalities. In Hen Shooter, each layer acts as a separate canvas for representing some game functionality. There can be more than one layer on the scene. Layers can be put on the scene or onto another layer. The order of layers is important. Each layer which is added to the scene covers all the layers below.

3.3 Scenes used in HenShooter

As mentioned before, Hen Shooter has a few scenes centrally managed by the SceneManager object:

- •Splash Scene
- •Menu Scene
- •Game Scene
- •InGameMenu Scene
- •Score Scene

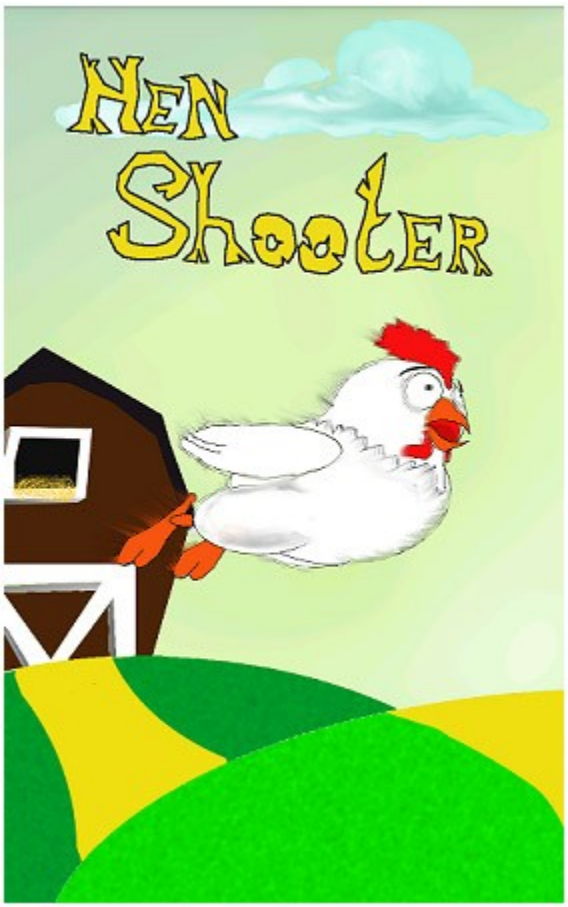
Splash Scene

The splash scene is the simplest of all scenes. It is displayed only once during application startup. There is no way to come back to that scene. The Splash scene is presented for three seconds. When it is visible all game resources are being loaded, the database is created (or just opened if it already exists) and the application cache is preloaded with available data. The SplashScene class is presented in the following code.

```
private final HenShooterActivity mHenShooterActivity;
public SplashScene(final HenShooterActivity pHenShooterActivity) {
         mHenShooterActivity = pHenShooterActivity;
         final ITextureRegion splashBackgorund = mHenShooterActivity
                  .getSplashTextureRegion(SplashTextureKeys.BACKGROUND);
         final SpriteBackground sprbackground = new SpriteBackground(
                  new Sprite(0, 0, splashBackgorund,
                  mHenShooterActivity.getVertexBufferObjectManager()));
                  setBackground(sprbackground);
}
```

[Code 5] Splash scene code

The following picture presents the HenShooter splash scene screen.



[Image 3] Splash scene screen

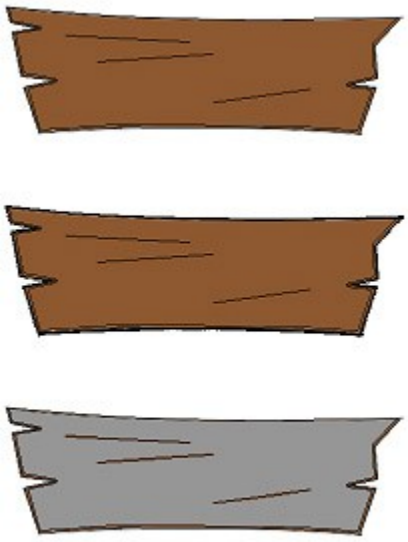
Menu Scene

The next scene is the game menu. This scene contains buttons such as play, score, exit, info about application, help, and buttons to toggle sounds and music on or off. The AndEngine framework provides ButtonSprite objects which can be utilized as buttons. In our application there was created class which extends ButtonSprite by adding a label. The name of this class is ButtonSpriteWithText. Creating such a button with text is shown in the following code.

```
ButtonSpriteWithText  button = new ButtonSpriteWithText(xPos, yPos, -buttonTextureRegion, vbo, label, bmpFont, audioPlayer);
```

[Code 6] Creating ButtonSpriteWithText

ButtonSpritewithText's constructor takes the following parameters positions x and y, TieldTextureRegion, VertexBufferObjectManager, label's text, BitmapFont and AudioPlayer. This button has three states: normal, pressed and disabled. Every state has to be on a separate frame of texture region. The first frame represents the normal state. The second frame represents the pressed state and the last frame represents the disabled button appearance. The following picture shows sample bitmap which can be used as texture for button.



[Image 4] Textures for SpriteButton

The label parameter is a String. It will be put on the button as its label. The BitmapFont object defines the font for this label. Additionally the AudioPlayer plays a sound when a button is clicked.

The next control which is used in the application on the MenuScene is a ToggleButtonSprite. This button serves to switch something on or off. It was used in the application for toggling music and sounds.

The ToggleButtonSprite constructor takes almost the same parameters as ButtonSpriteWithText, only without a label and bitmapFont. The texture region for this button has to have five frames as it is shown in the following picture.

[Image 5] Texture for ToggleButtonSpite

ToggleButtonSprite states are defined as enum ToggleState. The implementation of this enum is shown in the following code.

```
public static enum ToggleState {
        ON(0), ON_PRESSED(1), OFF(2), OFF_PRESSED(3), DISABLE(4);
        private final int mState;
        private ToggleState(final int pSate) {mState = pSate;}
        public int get() {return mState;}
};
```

[Code 7] Enum ToggleState

The ToggleButtonSprite is informed about the click action through an OnToggleClickListener interface. Its implementation is shown below.

```
public interface OnToggleClickListener {
	void onOffClick(ToggleButtonSprite pButtonSprite, float pTouchAreaLocalX, float pTouchAreaLocalY);
	void onOnClick(ToggleButtonSprite pButtonSprite, float pTouchAreaLocalX, float pTouchAreaLocalY);
};
```

[Code 8] Interface OnToggleClickListener

Methods from the listener presented above are called in the onAreaTouched method as it is shown in the following code.

```
@Override
public boolean onAreaTouched(final TouchEvent pSceneTouchEvent, final float pTouchAreaLocalX, final float pTouchAreaLocalY) {
	if (mState == ToggleState.DISABLE) {
		return true;
	}
	if (pSceneTouchEvent.isActionDown()) {
		if (mState == ToggleState.ON) {
			setCurrentTileIndex(ToggleState.OFF_PRESSED.get());
			changeState(ToggleState.ON_PRESSED);
		} else if (mState == ToggleState.OFF) {
			setCurrentTileIndex(ToggleState.ON_PRESSED.get());
			changeState(ToggleState.OFF_PRESSED);
		}
	} else if (pSceneTouchEvent.isActionUp()) {

		if (mAudioPlayer != null) {
			mAudioPlayer.play(SoundKeys.BUTTON_PRESS);
		}
		if (mOnToggleClickListener != null) {
			if (mToogleState) {
				setCurrentTileIndex(ToggleState.OFF.get());
				changeState(ToggleState.ON);
				mOnToggleClickListener.onOnClick(this, pSceneTouchEvent.getX(), pSceneTouchEvent.getY());
			} else {
				setCurrentTileIndex(ToggleState.ON.get());
				changeState(ToggleState.OFF);
```

```
                                    mOnToggleClickListener.onOffClick(this,
pSceneTouchEvent.getX(), pSceneTouchEvent.getY());
                            }
                            mToogleState = !mToogleState;
                    }
            }

            return true;
}
```

[Code 9] Code of onAreaTouched method

As it can be observed in the above excerpt, the code invoked after a button click in ToggleButtonSprite sets the state of this button. A suitable frame is set according to the button's internal state and that state is changed to the opposite one. The menu scene is shown in the following picture.

[Image 6] Menu scene screen

High-score Scene

The High-score scene is used to show the high-score board. It uses Text objects to represent text and one ButtonSpriteWithText to allow the user to return to the main menu. The text object construction

parameters are positions x and y, bitmap font, label, text options object and VertexBufferManagerObject. Text options are an object of TextOptions type. TextOptions can define the alignment of text by means of passing a HorizontalAlign object. HorizontalAlign is an enum which can have values such as CENTER, LEFT or RIGHT. The Hen Shooter application uses a TextOptions object with one parameter - HorizontalAlign.CENTER. Scores data is taken from the database. Only ten best scores are stored. If there are fewer than 10 records in the database, then the board is supplemented with default labels. The screen which presents the menu scene is shown below.

[Image 7] Menu scene screen

Game Scene

The game scene is the most sophisticated scene in HenShooter. The entire game action is held on it. This scene has the largest number of layers and the most complicated mechanism. It uses the AndEngine framework possibilities to the greatest extent.

In Game Menu Scene

The scene is shown in the game after clicking the back button or after clicking the pause button. This scene is different from the previously described scenes because it extends the MenuScene from the AndEngine framework.
The MenuScene object has special properties. There is a possibility to add items to it which will be managed by the menu scene. Items which can be added to the MenuScene have to implement the IMenuItem interface. Hen Shooter uses the ButtonSpriteWithTextMenuItem class, which extends ButtonSpriteWithText and implements IMenuItem. Its body is shown in the following code.

```
public class ButtonSpriteWithTextMenuItem extends ButtonSpriteWithText implements IMenuItem
{
        private final int mId;
        public ButtonSpriteWithTextMenuItem(final int pId, final TiledTextureRegion
pNormalTextureRegion, final VertexBufferObjectManager pVertexBufferObjectManager, final
String pText, final BitmapFont pBitmapFont, final AudioPlayer pAudioPlayer) {
                super(0, 0, pNormalTextureRegion, pVertexBufferObjectManager, pText,
pBitmapFont, AudioPlayer);
                mId = pId;
        }
        @Override
        public int getID() {return mId;}

        @Override
        public void onSelected() {// Do nothing}

        @Override
        public void onUnselected() {// Do nothing}
}
```

[Code 10] ButtonSpriteWithTextMenuItem

The IMenuItem interface provides three methods:

- •getID();
- •onSelected();
- •onUnselected();

In the above code the last two methods are not used because the “select” action is handled by an OnClickListener from the ButtonSpriteWithText class. Items are added to the MenuScene with addMenuItem(IMenuItem). When the menu is displayed on screen it is possible to use a modifier (for example an AlphaModifier) to change its appearance. To use the animation, the developer has to use the buildAnimations() method. It is possible to set a MenuAnimator via the setMenuAnimator(IMenuAnimator) method. The default MenuAnimator is AlphaMenuAnimator. All menu items will behave in the same way.

Additionally there is a layer located on this scene. It contains buttons such as sound and music switches, help and about. The layer doesn’t implement the IMenuItem interface, so if the Menu will be animated the buttons which are located on layer, will not be animated. It is important to create a mechanism that solves this problem. Therefore in HenShooter there is an alpha modifier registered for all buttons on that layer. The described solution is showed in the following code.

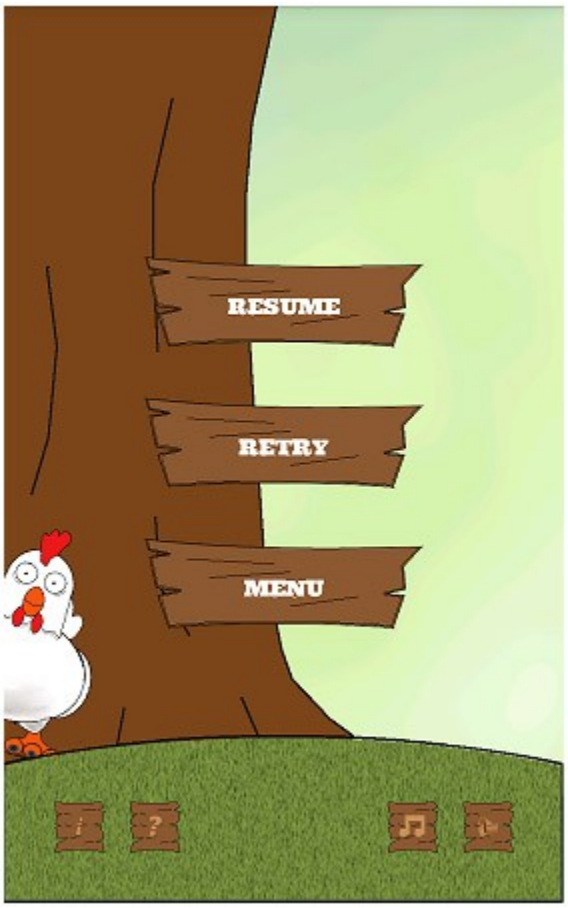
```
public final void registerAlphaModifierForAllButtons() {
        mSoundButton.setAlpha(0);
        mMusicButton.setAlpha(0);
        mHelpButton.setAlpha(0);
```

```
        mInfoButton.setAlpha(0);
        mSoundButton.registerEntityModifier( new AlphaModifier(1.0f, 0, 1));
        mMusicButton.registerEntityModifier( new AlphaModifier(1.0f, 0, 1));
        mHelpButton.registerEntityModifier( new AlphaModifier(1.0f, 0, 1));
        mInfoButton.registerEntityModifier( new AlphaModifier(1.0f, 0, 1));
}
```

[Code 11] Method body which animates buttons on layer

When the MenuScene is displayed on the screen, the scene which shows the MenuScene is paused.

The screen with the InGameMenuScene is presented on the image below.



[Image 8] Screen of in game menu scene

3.4 Resources

HenShooter makes use of numerous amounts of resources like graphics and audio files. The following sections describe the usage of textures and sounds in AndEngine.

Texture

AndEngine provides means for efficient graphics managment and using graphic assets represented as textures. Due to performance issues and the fact that HenShooter is a mobile application run on devices with a limited amount of memory, texture management is a very important part of the game.

The first thing to remember is that images used as assets should have a size which is the power of 2. Otherwise the image size will be rounded to the closest values which are the power of 2 resulting in huge waste of memory resources. AndEngine uses the concept of a texture atlas which is one big image which contains all graphics used in game. Thanks to this approach resources are loaded to memory only once which gives great performance boost. The texture atlas consists of texture regions representing separate textures. Utilizing regions makes it possible to have many sprite objects which are textured with the same texture. As a result big amounts of memory are spared. Loading resources in HenShooter is performed in the Scene Manager's load[scene name]SceneResources() methods for every scene separately as the following code fragment presents.

```
public void loadGameSceneResources() {

        final HenShooterSpriteSheet mainSpriteSheet = HenShooterSpriteSheet.MAIN;
        final BitmapTextureAtlas gameBitmapTextureAtlas = new
BitmapTextureAtlas(mHenShooterActivity.getTextureManager(), mainSpriteSheet.getWidth(),
mainSpriteSheet.getHeight(), TextureOptions.BILINEAR);
        createHenTextureRegions(mainSpriteSheet, gameBitmapTextureAtlas);
        createHenPartsTextureRegions(mainSpriteSheet, gameBitmapTextureAtlas);
        createBulletsTextureRegions(mainSpriteSheet, gameBitmapTextureAtlas);
        createThunderboltsTextureRegions(mainSpriteSheet, gameBitmapTextureAtlas);
        createGunTextureRegions(mainSpriteSheet, gameBitmapTextureAtlas);
        gameBitmapTextureAtlas.load();
        mHenShooterActivity.putBitmapTextureAtlas(BitmapTextureAtlasKeys.GAME,
gameBitmapTextureAtlas);

        final HenShooterSpriteSheet particlesSpriteSheet = HenShooterSpriteSheet.PARTICLES;
        final BitmapTextureAtlas particleBitmapTextureAtlas = new
BitmapTextureAtlas(mHenShooterActivity.getTextureManager(), particlesSpriteSheet.getWidth(),
particlesSpriteSheet.getHeight(), TextureOptions.BILINEAR_PREMULTIPLYALPHA);
        createParticlesTextureRegions(particlesSpriteSheet, particleBitmapTextureAtlas);
        particleBitmapTextureAtlas.load();
        mHenShooterActivity.putBitmapTextureAtlas(BitmapTextureAtlasKeys.PARTICLE,
particleBitmapTextureAtlas);

        final BitmapTextureAtlas backgroundGameTextureAtlas = new
BitmapTextureAtlas(mHenShooterActivity.getTextureManager(), 1024, 2048,
TextureOptions.BILINEAR);
        createGameBackgroundTextureRegion(backgroundGameTextureAtlas);
        backgroundGameTextureAtlas.load();
        mHenShooterActivity.putBitmapTextureAtlas(BitmapTextureAtlasKeys.GAME_BACKGROUND,
backgroundGameTextureAtlas);
}
```

[Code 12] Code of loadGameSceneResources method

The presented code loads resources needed for the game scene. First of all it creates an instance of BitmapTextureAtlas which represents all image resources. Its size corresponds to the size of the main sprite sheet, which currently is 1024x1024 and contains all image resources used for texturing sprites. Then there are calls to methods which create different texture regions on the texture atlas. They take as arguments two parameters: a HenShooterSpriteSheet instance which defines location of all image assets and a BitmapTextureAtlas which represents the sprite sheet in the game's memory. The body of one of these methods is presented below.

```
private void createHenTextureRegions(final HenShooterSpriteSheet pSpriteSheet, final
BitmapTextureAtlas pBmpTxtAtlas) {
        final HenShooterTiledAsset henAsset = HenShooterTextureAssets.getHenAsset();
        Assert.assertTrue(henAsset.getSpriteSheet() == pSpriteSheet);
        createHenTextureRegion(GameTextureKeys.HEN_COLORFUL, henAsset, pBmpTxtAtlas);

        final HenShooterTiledAsset redHenAsset = HenShooterTextureAssets.getRedHenAsset();
        Assert.assertTrue(redHenAsset.getSpriteSheet() == pSpriteSheet);
        createHenTextureRegion(GameTextureKeys.HEN_RED, redHenAsset, pBmpTxtAtlas);
        //Rest of the method's implementation is here.
}
```

[Code 13] Excerpt of createHenTextureRegions method.

The createHenTextureRegions method is responsible for creating regions for the different hen colors. It makes use of the HenShooterTextureAssets class which contains all the information about texture location on the main sprite sheet and returns a HenShooterTiledAsset object. The returned asset is passed to the createHenTextureRegion(...) method which adds it to a particular place on the texture atlas.

Coming back to loadGameSceneResources() there are still some things which need explaining. After creating of the texture regions, the load() method is invoked on gameBitmapTextureAtlas object. A texture atlas is loaded into the game's memory at this point. The rest of the presented code performs identical operations to load textures used for particles.

The HenShooter game uses two sprite sheets represented as HenShooterSpriteSheet enum objects. One is called MAIN and contains the location of all images needed for game textures. The second one, called PARTICLES, contains all the graphics needed for showing different particles. The location of each graphic asset on the sprite sheet is defined in the HenShooterTextureAssets class which contains static getters for accessing all resources. Information about each particular asset is stored as an instance of the HenShooter asset class whose code is presented below.

```
public class HenShooterAsset {
        private final String mAssetFileName;
        private final HenShooterSpriteSheet mSpriteSheet;
        private final int mTextureX;
        private final int mTextureY;

        public HenShooterAsset(final String pAssetFileName, final HenShooterSpriteSheet
pSpriteSheet, final int pTextureX, final int pTextureY) {
```

```
                mAssetFileName = pAssetFileName;
                mSpriteSheet = pSpriteSheet;
                mTextureX = pTextureX;
                mTextureY = pTextureY;
        }
        public String getAssetFileName() {
                return mAssetFileName;
        }

        public HenShooterSpriteSheet getSpriteSheet() {
                return mSpriteSheet;
        }

        public int getTextureX() {
                return mTextureX;
        }

        public int getTextureY() {
                return mTextureY;
        }
}
```

[Code 14] Code of HenShooterAsset class.

The HenShooterAsset class stores basic sets of information: X and Y coordinates of asset's location on the sprite sheet, the name of the file which contains asset's image and name of the sprite sheet to which the asset should be added. The existence of a HenShooterTiledAsset class, which inherits from HenShooterAsset, is worth noting. It contains additional information about the number of tiles and their distribution i.e. the number of columns and rows it has. Using this class allows the developer to create a sprite sheet with few tiles and to decide which of them should be currently presented. This approach may be used to create simple animations.

Music and Sounds

Music and sound are indispensable components of almost every game; therefore AndEngine provides a means for playing sound effects. To enable music and sounds it is required to modify the audio options of engine during its creation in the onCreateEngineOptions () method in the following way:

engineOptions.getAudioOptions ().setNeedsSound (true).setNeedsMusic (true);

Having done that it is possible to start using audio effects in game. AndEngine provides two types of effects: music and sounds. The main difference between them is that music usually has longer duration and is used for background effects. There are two factory classes MusicFactory and SoundFactory, which allow creating Music and Sound objects from project assets. It is recommended to use .wav files for sounds and .mp3 files for music. Some of the methods for managing audio playback, which are provided by Music and Sound objects, are play(), pause(), resume() or isPlaying(). The engine also delivers sound and music managers which are accessible with getSoundManager () and getAudioManager () methods. HenShooter uses music in it's game menus and as background during play. It also uses different sounds as a reaction to object's collision. The

HenShooter application contains an AudioPlayer class whose main purpose is to centralize music and sound playback management. An instance of this class is created and stored in the HenShooterActivity object. An excerpt of the AudioPlayer class has been inserted below. It is followed by a short explanation of concepts used in this class.

```
public class AudioPlayer {
	private static final String TAG = "AudioPlayer";
	private final EnumMap<SoundKeys, Sound> mSounds;
	private final EnumMap<MusicKeys, Music> mMusic;
	private final Set<SoundKeys> mPausedSounds;
	private final Set<MusicKeys> mPausedMusic;
	private boolean mMusicOn;
	private boolean mSoundsOn;

	public AudioPlayer(final boolean pMusicOn, final boolean pSoundOn) {
		//Implementation code is here.
	}

	public void loadMusicAndSound(final SoundManager pSoundManager, final MusicManager
pMusicManager, final Context pContext) throws IOException {
		try {
			loadMusic(pMusicManager, pContext);
			loadSoundEffects(pSoundManager, pContext);
		} catch (final IOException e) {
			Log.e(TAG, "Couldn't load audio asset.", e);
			throw e;
		}
	}

	public void play(final MusicKeys pMusic) {
		playAudio(getAudio(pMusic, mMusic), mMusicOn, false);
	}

	public void play(final SoundKeys pSound) {
		playAudio(getAudio(pSound, mSounds), mSoundsOn, false);
	}

	private void loadMusic(final MusicManager musicManager, final Context context)
throws IOException {
		final Music gameMusic = MusicFactory.createMusicFromAsset(musicManager,
context, "game_music.mp3");
		gameMusic.setLooping(true);
		gameMusic.setVolume(0f);
```

```
                mMusic.put(MusicKeys.GAME_MUSIC, gameMusic);
        }

        private void loadSoundEffects(final SoundManager soundManager, final Context
context) throws IOException {
                final Sound henShoot = SoundFactory.createSoundFromAsset(soundManager,
context, "hen_shot.wav");
                mSounds.put(SoundKeys.HEN_SHOT, henShoot);
        }
}
```

[Code 15] Excerpt of AudioPlayer class

The first thing to notice is that AudioPlayer has two separate maps for storing sounds and music. Their values have keys which come from MusicKeys and SoundKeys enums. The player keeps track of the current state of audio playback by storing information such as: if music is on, if sounds are on etc. One of the most important methods of the AudioPlayer class is loadMusicAndSound (...) which invokes loadMusic (...) and loadSoundEffects (...). These methods create different effects from assets utilizing create [Sound, Music] FromAsset (...) methods of previously described factory classes and puts them to described maps. AudioPlayer has methods for playing, pausing and stopping either single or all sound and music effects. Due to the issues in AndEngine connected to stopping sounds (sound stopped with stop() method cannot be played again with play()) AudioPlayer turns down audio volume to 0 to simulate stop behavior. It is also worth mentioning that there are also setSoundsOn (...) and setMusicOn (...) methods which allow to muting all sound and music effects.

4. Engine

The engine class is responsible for managing drawing and updating the current Scene, which contains all currently active game objects. The HenShooter game introduced the EngineWithTextureRendering class which extends Engine and provides the means for rendering many different textures from one base texture during runtime. An example of application of this concept in HenShooter is creating hen textures in many colors from one base texture. Below are presented core concepts of the EngineWithTextureRendering class followed by their short explanations

```
@Override
public void onDrawFrame(final GLState pGlState) throws InterruptedException {
        // Reload hen textures after application resume.
        if (mReloadHenTextures) {
                reloadHenTextures(pGlState);
                mReloadHenTextures = false;
        }
        super.onDrawFrame(pGlState);
        if (!mRenderTextureInitialized) {
                createColoredHenTextures(pGlState);
                mRenderTextureInitialized = true;
        }

}
```

[Code 16] Code presenting onDrawFrame method from EngineWithTextureRendering class

EngineWithTextureRendering overrides the onDrawFrame() method from the Engine class. The OnDrawFrame() method is invoked by AndEngine each time the Scene is updated. An overridden version of this method allows for modifying behavior of texture rendering. In case of this game, first of all it checks if hen textures should be reloaded. If texture reloading is necessary, it calls reloadHenTextures (...). It also checks if textures are initialized and invokes createColoredHenTextures (...) if needed.

```
private TiledTextureRegion createHenTextureRegion(final HenColor pColor, final Sprite
pPartsToColor, final Sprite pPartsNotToColor, final GLState pGlState){
        final RenderTexture renderTexture = new RenderTexture(getTextureManager(),
Hen.HEN_SIZE * HenShooterTextureAssets.HEN_TILE_COUNT, Hen.HEN_SIZE * 4);
        renderTexture.init(pGlState);
        pPartsToColor.setColor(pColor.getColor());
        renderTexture.begin(pGlState, 0.0f, 0.0f, 0.0f, 0.0f);
        pPartsToColor.onDraw(pGlState, getCamera());
        pPartsNotToColor.onDraw(pGlState, getCamera());
        renderTexture.end(pGlState);

        final Bitmap henBitmap = renderTexture.getBitmap(pGlState);
        final Point henTexturePosition =
HenShooterTextureAssets.getHenTexturePosition(pColor);
        final BitmapTextureAtlasSource bitmapTextureAtlasSource = new
BitmapTextureAtlasSource(henBitmap, henTexturePosition.x, henTexturePosition.y, Hen.HEN_SIZE
* HenShooterTextureAssets.HEN_TILE_COUNT,Hen.HEN_SIZE * 4);

        mBitmapTextureAtlasSource.add(bitmapTextureAtlasSource);

        return
updateHenTexture(mHenShooterActivity.getBitmapTextureAtlas(BitmapTextureAtlasKeys.GAME),
bitmapTextureAtlasSource);
}
```

[Code 17] Code presenting the createHenTextureRegion method from EngineWithTextureRendering class

The createHenTextureRegion (…) method renders hen colors by combining hen contours and white parts of the hen colored with some color. It takes four objects as parameters: HenColor indication color of rendered hen, Sprite with hen body, Sprite with hen contour and GLState. At the beginning it creates a new texture which then is colored with the hen color from the pColor parameter using the Sprite's onDraw (…) method. It then creates a BitmapTextureAtlasSource using the bitmap returned by the getBitmap(…) method invoked on a RenderTexture object. Lastly it invokes updateHenTexture (…) which creates a TiledTextureRegion from a colored texture and finally returns that region.

```
private void  createColoredHenTextures(final GLState pGlState) {
        final TiledTextureRegion tiledToColor = (TiledTextureRegion)
```

```
mHenShooterActivity.getGameTextureRegion(GameTextureKeys.HEN_PARTS_TO_COLOR);

        final TextureRegion toColor =
TextureRegionFactory.extractFromTexture(tiledToColor.getTexture(), (int)
tiledToColor.getTextureX(), (int) tiledToColor.getTextureY(), Hen.HEN_SIZE*
HenShooterTextureAssets.HEN_TILE_COUNT, Hen.HEN_SIZE *
HenShooterTextureAssets.HEN_ROW_COUNT);

        final Sprite partsToColor = new Sprite(0, 0, toColor,
mHenShooterActivity.getVertexBufferObjectManager());

        final TiledTextureRegion tiledNotToColor = (TiledTextureRegion)
mHenShooterActivity.getGameTextureRegion(GameTextureKeys.HEN_PARTS_NOT_TO_COLOR);

        final TextureRegion notToColor =
TextureRegionFactory.extractFromTexture(tiledNotToColor.getTexture(), (int)
tiledNotToColor.getTextureX(), (int)tiledNotToColor.getTextureY(), Hen.HEN_SIZE*
HenShooterTextureAssets.HEN_TILE_COUNT, Hen.HEN_SIZE *
HenShooterTextureAssets.HEN_ROW_COUNT);

        final Sprite partsNotToColor = new Sprite(0, 0, notToColor,
mHenShooterActivity.getVertexBufferObjectManager());

        final TiledTextureRegion mHenTextureRegion = createHenTextureRegion(HenColor.WHITE,
partsToColor, partsNotToColor, pGlState);
        mHenShooterActivity.putGameITextureRegion(GameTextureKeys.HEN_COLORFUL,
mHenTextureRegion);

        final TiledTextureRegion mHenBlackTextureRegion =
createHenTextureRegion(HenColor.BLACK, partsToColor, partsNotToColor, pGlState);
        mHenShooterActivity.putGameITextureRegion(GameTextureKeys.HEN_BLACK,
mHenBlackTextureRegion);

        final TiledTextureRegion mHenWhiteTextureRegion =
createHenTextureRegion(HenColor.WHITE, partsToColor, partsNotToColor, pGlState);
        mHenShooterActivity.putGameITextureRegion(GameTextureKeys.HEN_WHITE,
mHenWhiteTextureRegion);
}
```

[Code 18] Code presenting the createColoredHenTextures method from the EngineWithTextureRendering class

The createColoredHenTextures (...) method is used to initialize the colored textures which will be dynamically rendered during runtime. It takes one hen texture and creates textures with different colors using the createHenTextureRegion (...) method described above. At the beginning it extracts

texture regions containing hen parts and creates two Sprites: partsToColor (created on the base of the toColor texture region) and partsNotToColor (created on the base of notToColor textureRegion). It then creates hens with different colors by passing these two Sprites and the desired hen color to createHenTextureRegion (...). The class excerpt located above presents creating only white and black hens, but the full source of the described class initializes 7 different hen colors.

The mechanism described in this section is not so easy and can be cumbersome for developers at the beginning, but it gives great possibilities and a performance boost. Utilizing this method allows a developer to create many different objects on the scene, based upon one texture colored during runtime.

5. S Pen integration

The S Pen library in HenShooter has been integrated with AndEngine in order to pass S Pen stylus touch events to code dealing directly with AndEngine. Such a solution gives developers the means to write more interactive and create more interesting games which distinguish various ways of input. Thanks to the aforementioned integration it is possible to perform different actions depending on user input e.g. apply forces in physics world which correspond to stylus pressure or open the contextual menu on stylus button click. The HenShooter game contains a SCanvasIntegratorView class – a firm base for handling all S Pen touch events and passing them to AndEngine scenes. The Integrator is implemented as a transparent view inheriting from RelativeLayout which is added on top of AndEngine scenes. The general structure of SCanvasIntegratorView is presented below.

```
public class SCanvasIntegratorView extends RelativeLayout implements SPenTouchListener {

	private ITouchListener mTouchListener;
	private final SPenEventLibrary mSPenEventLibrary;

	public SCanvasIntegratorView(final Context pContext, final int pWidth, final int
pHight) {
		//Method implementation code is here.
	}

	public void setTouchListener(final ITouchListener pTouchListener) {
		mTouchListener = pTouchListener;
	}

	@Override
	public void onTouchButtonDown(final View pView, final MotionEvent pEvent) {
	}

	@Override
	public void onTouchButtonUp(final View pView, final MotionEvent pEvent) {
	}
	@Override
	public boolean onTouchFinger(final View pView, final MotionEvent pEvent) {
		return onTouchAction(pView, pEvent);
```

```
        }

        @Override
        public boolean onTouchPen(final View pView, final MotionEvent pEvent) {
                return onTouchAction(pView, pEvent);
        }

        @Override
        public boolean onTouchPenEraser(final View pView, final MotionEvent pEvent) {
                return onTouchAction(pView, pEvent);
        }

        private boolean onTouchAction(final View pView, final MotionEvent pEvent) {
                if (mTouchListener != null) {
                        mTouchListener.onTouchEvent(pView, pEvent);
                        return true;
                }
                return false;
        }
}
```

[Code 19] Code for S Pen integration

The first important thing about the presented class is that its mTouchListener attribute contains a reference to the ITouchListener object created in the onPopulateScene (...) method of the main activity. The use of ITouchListener makes it possible to pass touch events directly to the AndEngine engine object. The second thing to notice is that SCanvasIntegratorView implements SPenTouchListener which is from the S Pen Library and is responsible for handling S Pen stylus touch events. The following methods come from this listener: onTouchButtonDown(...), onTouchButtonUp(...), onTouchFinger(...), onTouchPen(...), onTouchPenEraser(...). They are invoked by the S Pen library as a reaction to different events. SPenCanvasIntegratorView provides an implementation for some of them. Their implementation passes View and MotionEvent objects provided by S Pen to onTouchAction (final View pView, final MotionEvent pEvent) which in turn passes that object to the onTouchEvent (...) method from the ITouchListener object. MotionEvent objects passed from the S Pen integrator view contain information about the input method e.g. if user touched some particular area with his/her finger or pen, or if the pen button was clicked while performing the touch event. The solution for S Pen integration applied in HenShooter enables developers to create many views which will react on S Pen events. For example it is possible to create part of the game scene which can be used only with the stylus and another part which will react only to finger touch events.

6. Database and application cache

HenShooter like many other games has some state which should be persisted. In case of this particular game it was necessary to store information about game settings (if music and sounds are turned on) and high scores. The Android platform offers two main means for storing information: application properties and a SQLite database. Utilizing the SQLite database was the most natural choice for HenShooter, due to the fact that the application properties don’t allow creating any

relations between stored data, which could be necessary if someone in the future would like to extend the game's functionalities and store more structured information. The application cache was used in addition to the database. Its purpose is preloading necessary data from database at scene creation and as a result minimizing the number of queries.

6.1 Database

The database created for HenShooter's purpose consists of two simple tables: SettingAppTable and ScoreTable. Its structure is reflected in the HenShooterDb class, which contains static inner classes representing the table's structure. SettingAppTable stores information about application settings in the form of a key-value pair, where key is the name of the setting and values is a flag indicating if the particular option is turned on or off. The ScoreTable tabel contains game high scores saved as player's name and the number of points the player gathered during the game. In addition the HenShooterSQLiteOpenHelper class, which extends SQLiteOpenHelper, was created. It provides methods for performing database operations like creating and updating tables or filling the database with default values.

6.2 Application cache

The application cache was created as the database's complement to load application settings once during scene creation and minimize the number of queries performed on the database. The cache of the HenShooter application is implemented in the AppCache class. It makes use of the singleton design pattern, which ensures that only one instance of the application cache's object exists during application runtime. The cache consists of two enum maps, one storing integer values and the second float values. Possible map keys are values which belong to the AppCacheKey enum. The code fragment presented below shows the basic structure of the AppCache class implemented with the singleton pattern.

```
public final class AppCache {
        private static AppCache sInstance;
        private final EnumMap<AppCacheKey, Integer> mSettingInt;

        private AppCache() {
                //Method implementation code is here.
        }

        public static synchronized AppCache getInstance() {
                if (sInstance == null) {
                        sInstance = new AppCache();
                }
                return sInstance;
        }

        public void putInt(final AppCacheKey pKey, final Integer pIntValue) {
                //Method implementation code is here.
        }
```

```
        public int getInt(final AppCacheKey pKey) {
                //Method implementation code is here.

                public void fillAppCacheFromDb(final SQLiteDatabase pDb) {
                        //Method implementation code is here.

                }
        }
}
```

[Code 20] AppCache implemented as singleton

The presented implementation of the singleton pattern is easy and doesn't require too much. First of all AppCache has the sInstance attribute of type AppCache which contains single instance of this class. Furthermore AppCache's no-arg constructor is private which ensures that direct creation of an AppCache objects is not possible. Finally there is the getInstance method which returns an instance of AppCache. It checks if a single instance was already created, if it turns out that there's no instance, it is created with private a no-arg constructor. Thanks to the fact that the getInstance method is static, it is possible to obtain the application cache object from every place of application code. In addition to methods related to the singleton implementation, the AppCache contains fillAppCacheFromDb which loads all the necessary information from the database to cache, and getXXX and putXXX methods (where XXX is Int or Float), which allows storing to and retrieving information from cache.

Ref:http://developer.samsung.com/android/technical-docs/Hen-Shooter-Chapter-1-Core-Game-Concepts