University of Reading

Department of Computer Science – SMPCS

CS2JA16 – JAVA
Coursework – ANDROID GAME

Spring Term

Giorgos Philippou

- Module Code: CS2JA16
- Assignment report Title: Android Game
- Student Number: 28000050
- Date: 19/04/2021
- Actual hrs spent for the assignment: 50+
- Assignment evaluation:
    1. Interesting
    2. Educational
    3. Important

The Spartan

Abstract

The Spartan game is a single player game that allows many users to play and save their high score to get a position on the leader board. There are three different levels, and the state is saved according to user's ID. A sensor interaction is implemented using touch events, to move and to fight with the AI enemies. An improvement on the speed and the memory usage also took place. Some design was both implemented in the game play and in other activities.

1. Introduction and Showcase

On this section there will be an introduction to the application. We will see the activities and their functionality will be briefly explained along with the design. The application is design to be shown on the user's hardware as an icon with the title "The Spartan", as shown in figure 1.

Figure 1: Icon & Name

By entering the application each user must go through three different activities. The first activity, "HomePage", contains a menu section which gives important information to the user about the game. Also, a place to provide an ID is visible, after entering an ID user can request to go to the next activity or see his personal high score, these can be done by tapping the first and the second button, respectively. If the ID is short enough or not entered the user can not do anything of these but a message is shown to inform him. A third button is also available at any time which shows the leader board with the three best scores. The second activity, called "LevelSelector", which shows to the user his score for each individual level. At this point, he can play any "enable" level, level 1 is always enabled, level 2 is enabled by wining level 1 and level 3 is enabled by winning level 2. To save a high score the user must win all levels and the sum of his 3 scores will be his total score, in such case level 2 and level 3 will be disabled and the individual score from the levels is reset. The third activity is the actual game, where the user will face different kind of enemies and try to get to the end of the arena to win the level. He can achieve that using the 4 buttons, on the right-hand side the buttons give you the ability to move on the x-axis and on the left-hand side the yellow button fights the enemy (must be tapped when an enemy is close for collision) and the blue button let the player to jump if he is on the ground. The fight button changes functionality after the user gets a gun. The user can use up to 2 buttons simultaneously, but this must be 1 button from each side. Points are available by killing enemies, getting coins or win a level. During the game play the score and the buttons are visible to the user and the screen display is centred to the player. The map is created by different kind of tiles, some of them are just for design without any functionality, some are blocking tiles (the player cannot go through them), another type is rapidly losing health points to player and the last ends the level with a win case. All of the above are shown in figure 2.

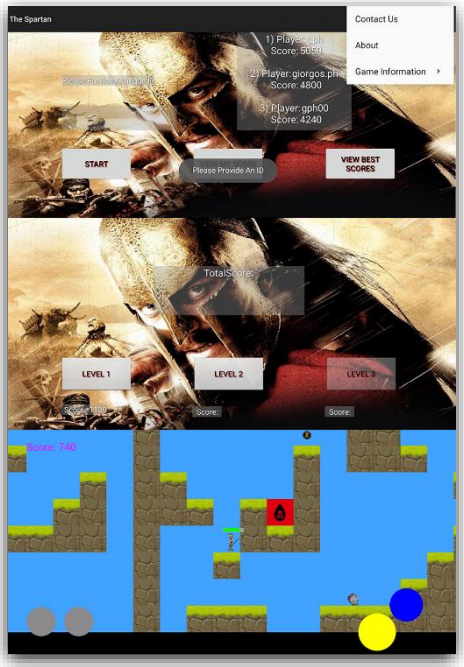


Figure 2: Activities

The design of the game is mostly implemented by me. I tried to make the first two activities

user-friendly by adding “TextViews” with coloured background but faded enough to let the page background visible, and by adding shadow to the text. Also, the menu is easy to be accessed and for each menu item an “AlertDialog” box is opened with the associated

information. Moreover, the user is informed for required actions, on some cases, with a “Toast” message. In the game play the tiles are edited and applied regarding their functionality throughout the game. The button sizes are applied after some trial to make it on an average difficulty and their colour differ based on their functionality again. Furthermore, the characters are designed by me, they were first drawn by hand and then I added some colours from an application on my laptop. The character design is clearly shown in figure 3.

Figure 1: Characters

2. OOP design

The application is implemented with an Object-Oriented Design and at this point we will go through the overall OOP design and the explanation of the functionality of the classes. We will see some cases of inheritance; how different classes interact; some of the libraries that are used. Starting with the first activity the class HomePage handles the first xml layout that the user will see. In that class we introduce SharedPreferences where are used for saving or loading information, in that case mostly saves information. We must give importance to onCreate() and onCreateOptionsMenu() which creates everything from the code regarding the layout and the menu, correspondingly. These clases are reference from android API, we are also using Intent to move to the next connected activity and sometimes transfer important information. The HomePage is connected to LeverSelector class which displays the second layout of the application. SharedPreferences are used again to save information for each user, like score, and set up a leader board. We call Intent again to move to the third connected activity, MainActivity that calls the game class and starts the actual game. On this class we use Intent to take information like the selected level to set the appropriate layout, or a case of win/loss after finishing a level. The three aforementioned classes extend Activity from android API, and the MainActicity calls Game class to initialize it and set it as the context.

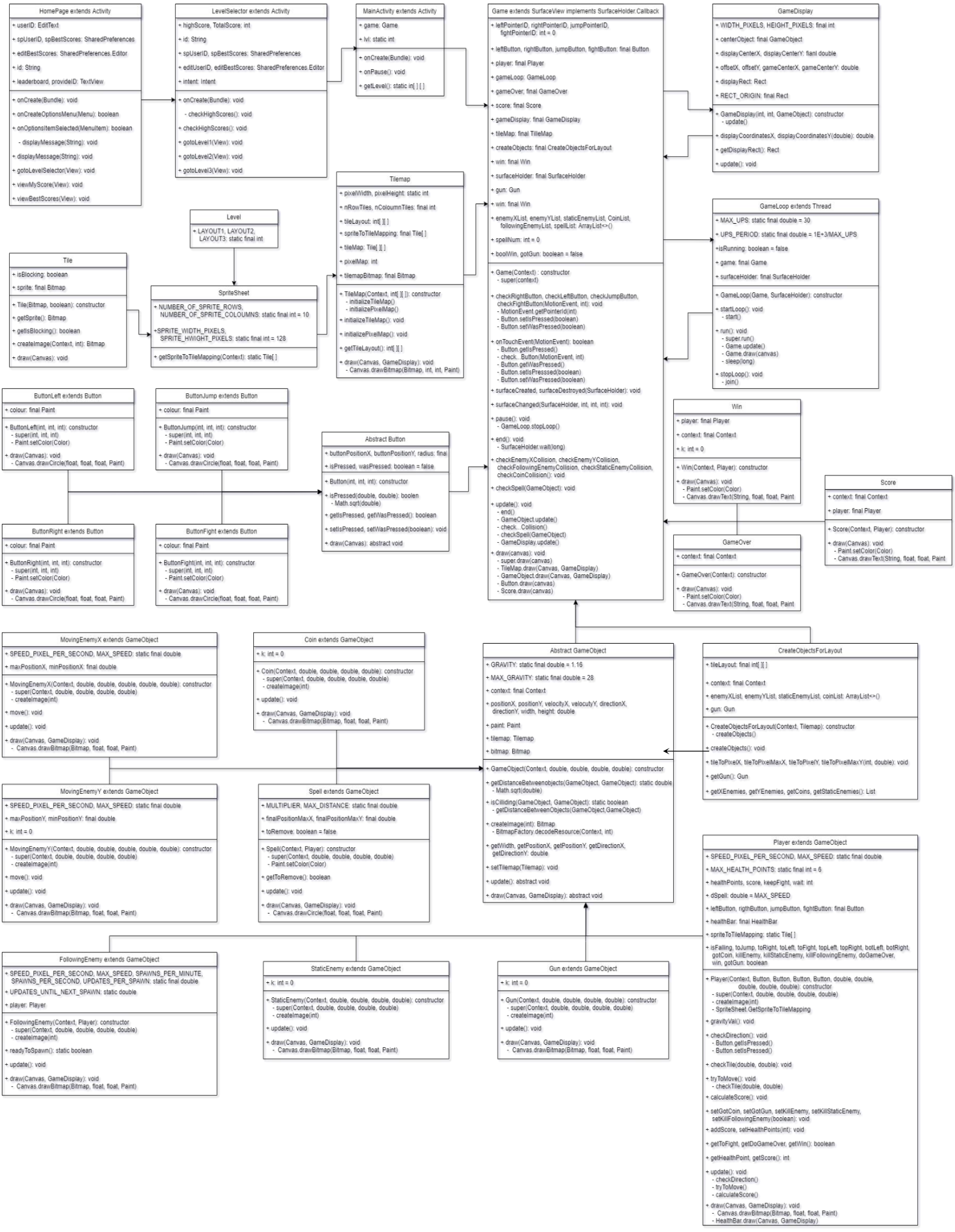
This is how the MainActivity interacts with Game to start our game. The Game class is an extension of the SurfaceView and it implements SurfaceHolder.Callback. This class gathers all the information that will built up the game; for example, game objects, game panels, the map etc. and it interacts with GameLoop and GameDisplay to handle the render and flow of the game. Most of the attributes and methods in the Game class are of private visibility. In the constructor we initialize every class that will be used in the game and we then face update()

and draw() methods. These methods are responsible to check for updates/changes before rendering and then render again the new game state. This is done by calling the same two functions, that are implemented in almost every class, on each iteration. Another significant feature is the implementation of onTouchEvent() which gives us the opportunity to interact with the user. On this class we also see the use of ArrayList< >(). The Game class interacts with every class either directly or by their superclass. The GameLoop extends Thread and is rensponsible to handle the iteration, update and draw, and the flow of the game. It starts or stops the game iterations using the thread and the flow is regulated by counting the targeted UPS and pause with Thread.sleep() or skip frames to keep that target. GameDiplay is set to render the game graphics for a specific space (width and height) that is always centred on the player.

The information regarding the map of the game starts with Tile class which is responsible to create the appropriate image for each different type of tiles and save details regarding the functionality of that tile. This class is connected to SpriteSheet where it initializes the tiles in an array type of Tile and is connected to Tilemap along with Level. Tilemap gets data from Level, where it contains the layout for each level, and SpriteSheet to initialize the final map, in both pixels and tiles. The Tilemap is abut with Game by a call of the draw() method. Generally, the visibility of the important methods is public and we face some cases which are also static. We then come across the first inheritance which is connected to the game and have Button as the superclass and RightButton, LeftButton, JumpButton and FightButton as the subclasses. The functionality of the inheritance is to know when each button independently is pressed by the user, and we achieved that using some getters and setters. The draw method is an abstract method in Button class, and it must be implemented by each of its subclasses. Until now the update method is not implemented as these classes have only static objects.

The second inheritance that interacts with Game has GameObject as the superclass and Player, MovingEnemyX, MovingEnemyY, StaticEnemy, FollowingEnemy, Coin, Gun and Spell as subclasses. In GameObject we can see many attributes with protected visibility as they are used in most of the subclasses. Rather than getters and setters, GameObject contains two methods that are responsible to check for collision between two objects; a protected method that creates an image for the object; the public abstract methods update() and draw() that must be implemented by every subclass. MovingEnemyX and MovingEnemyY are AI enemies that move on x and y axis accordingly with a specific starting point and boundaries. The update() method is implemented to set their new position on each frame update and draw() to render the image that will be displayed on the canvas. StaticEnemy, Coin and Gun are static objects where update() is implemented to be used just on the first frame to reposition the objects and draw() is implemented again to render the appropriate image. FollowingEnemy is an AI object which its starting position is random, and his direction is always to the player. The update() method is implemented to set the new position and calculate the direction that leads to the player, the draw() is implemented to render the image that will be displayed. Spell is an object moving to the x axis taking as a starting position the current position of the player and moving to his current x direction having boundaries. In this case the update() method has the same functionality but the draw() is creating a circle to be displayed. The Player subclass is the object which can be controlled by the user. It takes as parameters to its constructor each of the created buttons to recognise the direction it must move. In update() method we check for his current direction/velocity and the tile on that direction to calculated its next position. The tile checking is done by a static call to Tilemap class. We also check if there was a collision to set the new score value and health points. The draw() method have the same functionality with the other subclasses plus a call to the HealthBar.draw().

A class diagram to show the OOP design is provided on the next page.


HomePage extends Activity
+ userID: EditText
+ spUserID, spBestScores: SharedPreferences
+ editBestScores: SharedPreferences.Editor
+ id: String
+ leaderboard, provideID: TextView
+ onCreate(Bundle): void
+ onCreateOptionsMenu(Menu): boolean
+ onOptionsItemSelected(MenuItem): boolean
- displayMessage(String): void
+ displayMessage(String): void
+ gotoLevelSelector(View): void
+ viewMyScore(View): void
+ viewBestScores(View): void
LevelSelector extends Activity
+ highScore, TotalScore: int
+ id: String
+ spUserID, spBestScores: SharedPreferences
+ editUserID, editBestScores: SharedPreferences.Editor
+ intent: Intent
+ onCreate(Bundle): void
- checkHighScores(): void
+ checkHighScores(): void
+ gotoLevel1(View): void
+ gotoLevel2(View): void
+ gotoLevel3(View): void
MainActivity extends Activity
+ game: Game
+ lvl: static int
+ onCreate(Bundle): void
+ onPause(): void
+ getLevel(): static int[ ] [ ]
Game extends SurfaceView implements SurfaceHolder.Callback
+ leftPointerID, rightPointerID, jumpPointerID, fightPointerID: int = 0
+ leftButton, rightButton, jumpButton, fightButton: final Button
+ player: final Player
+ gameLoop: GameLoop
+ gameOver: final GameOver
+ score: final Score
+ gameDisplay: final GameDisplay
+ tileMap: final TileMap
+ createObjects: final CreateObjectsForLayout
+ win: final Win
+ surfaceHolder: final SurfaceHolder
+ gun: Gun
+ win: final Win
+ enemyXList, enemyYList, staticEnemyList, CoinList, followingEnemyList, spellList: ArrayList<>()
+ spellNum: int = 0
+ boolWin, gotGun: boolean = false
+ Game(Context) : constructor
- super(context)
+ checkRightButton, checkLeftButton, checkJumpButton, checkFightButton(MotionEvent, int): void
- MotionEvent.getPointerId(int)
- Button.setIsPressed(boolean)
- Button.setWasPressed(boolean)
+ onTouchEvent(MotionEvent): boolean
- Button.getIsPressed()
- check...Button(MotionEvent, int)
- Button.getWasPressed()
- Button.setIsPresssed(boolean)
- Button.setWasPressed(boolean)
+ surfaceCreated, surfaceDestroyed(SurfaceHolder): void
+ surfaceChanged(SurfaceHolder, int, int, int): void
+ pause(): void
- GameLoop.stopLoop()
+ end(): void
- SurfaceHolder.wait(long)
+ checkEnemyXCollision, checkEnemyYCollision, checkFollowingEnemyCollision, checkStaticEnemyCollision, checkCoinCollision(): void
+ checkSpell(GameObject): void
+ update(): void
- end()
- GameObject.update()
- check...Collision()
- checkSpell(GameObject)
- GameDisplay.update()
+ draw(canvas): void
- super.draw(canvas)
- TileMap.draw(Canvas, GameDisplay)
- GameObject.draw(Canvas, GameDisplay)
- Button.draw(canvas)
- Score.draw(canvas)
GameDisplay
+ WIDTH_PIXELS, HEIGHT_PIXELS: final int
+ centerObject: final GameObject
+ displayCenterX, displayCenterY: fianl double
+ offsetX, offsetY, gameCenterX, gameCenterY: double
+ displayRect: Rect
+ RECT_ORIGIN: final Rect
+ GameDisplay(int, int, GameObject): constructor
- update()
+ displayCoordinatesX, displayCoordinatesY(double): double
+ getDisplayRect(): Rect
+ update(): void
Tilemap
+ pixelWidth, pixelHeight: static int
+ nRowTiles, nColoumnTiles: final int
+ tileLayout: int[ ][ ]
+ spriteToTileMapping: final Tile[ ]
+ tileMap: Tile[ ][ ]
+ pixelMap: int
+ tilemapBitmap: final Bitmap
+ TileMap(Context, int[ ][ ]): constructor
- initializeTileMap()
- initializePixelMap()
+ initializeTileMap(): void
+ initializePixelMap(): void
+ getTileLayout(): int[ ][ ]
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, int, int, Paint)
GameLoop extends Thread
+ MAX_UPS: static final double = 30
+ UPS_PERIOD: static final double = 1E+3/MAX_UPS
+isRunning: boolean = false
+ game: final Game
+ surfaceHolder: final SurfaceHolder
+ GameLoop(Game, SurfaceHolder): constructor
+ startLoop(): void
- start()
+ run(): void
- super.run()
- Game.update()
- Game.draw(canvas)
- sleep(long)
+ stopLoop(): void
- join()
Level
+ LAYOUT1, LAYOUT2, LAYOUT3: static final int
Tile
+ isBlocking: boolean
+ sprite: final Bitmap
+ Tile(Bitmap, boolean): constructor
+ getSprite(): Bitmap
+ getIsBlocking(): boolean
+ createImage(Context, int): Bitmap
+ draw(Canvas): void
SpriteSheet
+ NUMBER_OF_SPRITE_ROWS, NUMBER_OF_SPRITE_COLOUMNS: static final int = 10
+SPRITE_WIDTH_PIXELS, SPRITE_HWIGHT_PIXELS: static final int = 128
+ getSpriteToTileMapping(Context): static Tile[ ]
ButtonLeft extends Button
+ colour: final Paint
+ ButtonLeft(int, int, int): constructor
- super(int, int, int)
- Paint.setColor(Color)
+ draw(Canvas): void
- Canvas.drawCircle(float, float, float, Paint)
ButtonJump extends Button
+ colour: final Paint
+ ButtonJump(int, int, int): constructor
- super(int, int, int)
- Paint.setColor(Color)
+ draw(Canvas): void
- Canvas.drawCircle(float, float, float, Paint)
Abstract Button
+ buttonPositionX, buttonPositionY, radius: final
+ isPressed, wasPressed: boolean = false
+ Button(int, int, int): constructor
+ isPressed(double, double): boolen
- Math.sqrt(double)
+ getIsPressed, getWasPressed(): boolean
+ setIsPressed, setWasPressed(boolean): void
+ draw(Canvas): abstract void
Win
+ player: final Player
+ context: final Context
+ k: int = 0
+ Win(Context, Player): constructor
+ draw(Canvas): void
- Paint.setColor(Color)
- Canvas.drawText(String, float, float, Paint
Score
+ context: final Context
+ player: final Player
+ Score(Context, Player): constructor
+ draw(Canvas): void
- Paint.setColor(Color)
- Canvas.drawText(String, float, float, Paint
GameOver
+ context: final Context
+ GameOver(Context): constructor
+ draw(Canvas): void
- Paint.setColor(Color)
- Canvas.drawText(String, float, float, Paint
ButtonRight extends Button
+ colour: final Paint
+ ButtonRight(int, int, int): constructor
- super(int, int, int)
- Paint.setColor(Color)
+ draw(Canvas): void
- Canvas.drawCircle(float, float, float, Paint)
ButtonFight extends Button
+ colour: final Paint
+ ButtonFight(int, int, int): constructor
- super(int, int, int)
- Paint.setColor(Color)
+ draw(Canvas): void
- Canvas.drawCircle(float, float, float, Paint)
MovingEnemyX extends GameObject
+ SPEED_PIXEL_PER_SECOND, MAX_SPEED: static final double
+ maxPositionX, minPositionX: final double
+ MovingEnemyX(Context, double, double, double, double, double): constructor
- super(Context, double, double, double, double)
- createImage(int)
+ move(): void
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)
Coin extends GameObject
+ k: int = 0
+ Coin(Context, double, double, double, double): constructor
- super(Context, double, double, double, double)
- createImage(int)
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)
Abstract GameObject
+ GRAVITY: static final double = 1.16
+ MAX_GRAVITY: static final double = 28
+ context: final Context
+ positionX, positionY, velocityX, velocutyY, directionX, directionY, width, height: double
+ paint: Paint
+ tilemap: Tilemap
+ bitmap: Bitmap
+ GameObject(Context, double, double, double, double): constructor
+ getDistanceBetweenobjects(GameObject, GameObject): static double
- Math.sqrt(double)
+ isCilliding(GameObject, GameObject): static boolean
- getDistanceBetweenObjects(GameObject,GameObject)
+ createImage(int): Bitmap
- BitmapFactory.decodeResource(Context, int)
+ getWidth, getPositionX, getPositionY, getDirectionX, getDirectionY: double
+ setTilemap(Tilemap): void
+ update(): abstract void
+ draw(Canvas, GameDisplay): abstract void
CreateObjectsForLayout
+ tileLayout: final int[ ][ ]
+ context: final Context
+ enemyXList, enemyYList, staticEnemyList, coinList: ArrayList<>()
+ gun: Gun
+ CreateObjectsForLayout(Context, Tilemap): constructor
- createObjects()
+ createObjects(): void
+ tileToPixelX, tileToPixelMaxX, tileToPixelY, tileToPixelMaxY(int, double): void
+ getGun(): Gun
+ getXEnemies, getYEnemies, getCoins, getStaticEnemies(): List
MovingEnemyY extends GameObject
+ SPEED_PIXEL_PER_SECOND, MAX_SPEED: static final double
+ maxPositionY, minPositionY: final double
+ k: int = 0
+ MovingEnemyY(Context, double, double, double, double, double): constructor
- super(Context, double, double, double, double)
- createImage(int)
+ move(): void
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)
Spell extends GameObject
+ MULTIPLIER, MAX_DISTANCE: static final double
+ finalPositionMaxX, finalPositionMaxY: final double
+ toRemove: boolean = false
+ Spell(Context, Player): constructor
- super(Context, double, double, double, double)
- Paint.setColor(Color)
+ getToRemove(): boolean
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawCircle(float, float, float, Paint)
Player extends GameObject
+ SPEED_PIXEL_PER_SECOND, MAX_SPEED: static final double
+ MAX_HEALTH_POINTS: static final int = 6
+ healthPoints, score, keepFight, wait: int
+ dSpell: double = MAX_SPEED
+ leftButton, rigthButton, jumpButton, fightButton: final Button
+ healthBar: final HealthBar
+ spriteToTileMapping: static Tile[ ]
+ isFalling, toJump, toRight, toLeft, toFight, topLeft, topRight, botLeft, botRight, gotCoin, killEnemy, killStaticEnemy, killFollowingEnemy, doGameOver, win, gotGun: boolean
+ Player(Context, Button, Button, Button, Button, double, double, double, double, double): constructor
- super(Context, double, double, double, double)
- createImage(int)
- SpriteSheet.GetSpriteToTileMapping
+ gravityVal(): void
+ checkDirection(): void
- Button.getIsPressed()
- Button.setIsPressed()
+ checkTile(double, double): void
+ tryToMove(): void
- checkTile(double, double)
+ calculateScore(): void
+ setGotCoin, setGotGun, setKillEnemy, setKillStaticEnemy, setKillFollowingEnemy(boolean): void
+ addScore, setHealthPoints(int): void
+ getToFight, getDoGameOver, getWin(): boolean
+ getHealthPoint, getScore(): int
+ update(): void
- checkDirection()
- tryToMove()
- calculateScore()
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)
- HealthBar.draw(Canvas, GameDisplay)
FollowingEnemy extends GameObject
+ SPEED_PIXEL_PER_SECOND, MAX_SPEED, SPAWNS_PER_MINUTE, SPAWNS_PER_SECOND, UPDATES_PER_SPAWN: static final double
+ UPDATES_UNTIL_NEXT_SPAWN: static double
+ player: Player
+ FollowingEnemy(Context, Player): constructor
- super(Context, double, double, double, double)
- createImage(int)
+ readyToSpawn(): static boolean
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)
StaticEnemy extends GameObject
+ k: int = 0
+ StaticEnemy(Context, double, double, double, double): constructor
- super(Context, double, double, double, double)
- createImage(int)
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)
Gun extends GameObject
+ k: int = 0
+ Gun(Context, double, double, double, double): constructor
- super(Context, double, double, double, double)
- createImage(int)
+ update(): void
+ draw(Canvas, GameDisplay): void
- Canvas.drawBitmap(Bitmap, float, float, Paint)

3. Memory usage and Speed improvements

The game updates and renders are handled, to run on a stable and smooth iteration, by the GameLoop class which extends Thread. The creation of the thread processes the rendering on each iteration and each time the updates are counted. I made a target value for the FPS/UPS (frames/updates per second) and I set it to 30. Then I calculated the required time to keep the frames at 30 by dividing a second with my FPS/UPS (1E+3/30). Therefore, for every iteration I calculate the elapse time by subtracting the current time by the starting time. Then, I compare the elapse time with the desired time, if the elapsed time is less the thread will be pause using Thread.sleep() to cover the difference, if the elapse time is higher than the desired, I skip some frames to get back to the targeted. For my single player game, the 30 FPS are working smoothly so I just kept this for less CPU consumption too. Also, I used almost everywhere enhanced for loops and that boost the game performance. Moreover, I set every attribute to that was unchanged after initialisation to final and I found that it helped the performance too.

I also tried to limit my memory usage and speed by using ArrayList< > and Iterator< >. The use of inheritance also helped on saving memory as I was using the reference of the superclass to call shared methods, and I did not have to rewrite them. Furthermore, I used the xml files to initialize some attributes. At this point I want to mention a situation where I added pictures for my GameObjects and I made the mistake to create the image in the Game class. This made the game very slow and unstable, but I immediately understood the mistake and I replaced that with a function in the superclass GameObject that creates the image and assigns it to the appropriate object and that fix the problem.

4. Improvements/extensions

Some of the feature and implementations of the game will be mentioned on this section:

- The game contains a leader board that save the best scores along with the users' unique ID using SharedPreferences.
- There is a menu which explains everything to the user.
- There are different types of AI and randomly developed enemies which lose health on collision and give different number of points when they are killed.
- Coins that give a standard number of points.
- Two types of fighting enemies:
  1. Without a gun the user must tap the button when is close to collision to kill an enemy.
  2. With a gun the user can shoot spells and kill the enemies.
- A gravity value that continuously increased until it reaches the max velocity.
- Tiles with different functionalities, win a level; lose health; block movement.
- Sensor interaction using different buttons to move, jump and fight. To jump you need to be on ground. Implemented to use up to 2 buttons simultaneously.
- Draw my own unique characters.
- Implementation of three different level etc.

There are also some features I attempted to add or did not get the time to add:

1. Animation on game objects.
2. Power ups.
3. Possibility for a bonus game that will pause the level and continue when is finished.
4. Multithreading
5. Improve game play and design details in the game and in other activities etc.

5. Conclusions

Taking everything into consideration, the OOP design turned out to be functional and good. The speed and memory usage were handled and resulted on running the game smoothly. Adding some design drawn by me (even though I wanted to spend more time on that and make animation) is an important feature for me and adding a leader board increased the interest of the game and made it competitive. In my own point of view, the project was fun, and I learned many things throughout it, like working on the performance of the game, working with android APIs, new techniques like multithreading and many more. Also, I think that the time of 30 hours was way off the actual time we must spent to develop such a project.

Last thinks to be mentioned are that Win, GameOver and Score classes were created to be implemented with an animation or video or adding some features, but I did not manage to finish that. If you are running the app with an emulator please prefer an XL size and I suggest to use it with a phone as in my case (my laptop is not very good) it cannot handle the graphics.

References

https://developer.android.com/reference/android/content/SharedPreferences

https://developer.android.com/reference/android/graphics/Bitmap

https://developer.android.com/guide/topics/ui/menus

https://developer.android.com/guide/topics/ui/notifiers/toasts

https://developer.android.com/reference/android/view/SurfaceView

https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout

Link to GitLab

https://csgitlab.reading.ac.uk/hz000050/androidgame.git

Demo self-assessment sheet

Each tick-box represents one mark unless otherwise specified.		Range
Code style (2 marks each points): Following Java code conventions for all the project ☑ variable and class names ☑ method names ☑ indentation rules ☑ Using inline comments frequency ☑ Javadoc comments for every function (except getters/setters)	Overall OOP design and API usage: ☑ Appropriate use of inheritance (2 marks) ☑ Appropriate use of Abstract classes (2 marks) ☑ Use of Android API classes/methods (Other than in base code) (3 marks) ☑ Greater than 3 original classes (3 marks)	0-20
Functionality: ☑ On-screen menus (1 mark) ☑ Scores (1 mark) ☑ Controllable character (1 mark) ☑ Sensor interaction (touch/ accelerometer/ etc) (1 mark) Game has levels ☑ Works (4 marks) [ ] Attempted (2 marks) Use of standardised data structures (e.g., List, Vector, Collection, Date): ☑ Works (10 marks) [ ] Attempted (5 marks) Randomly/procedurally generated features: ☑ Works (10 marks) [ ] Attempted (5 marks) Opponents (AI): ☑ Works (5 marks) [ ] Attempted (3 marks)	Design quality (both game and other game screens) (1 mark each): ☑ Professional looking ☑ Understandable game flow ☑ Feedback to user instead of crashing, or recover ☑ Runs smoothly without interruptions ☑ Installs without error ☑ Starts/exits without error ☑ Program does not crash ☑ Produced in video form	0-40
Improvements marks: Online high score list ☑ Works (10 marks) [ ] Attempted (5 marks) Multithreading improvements [ ] Works (10 marks) ☑ Attempted (5 marks) Memory optimisation ☑ Works (5 marks) [ ] Attempted (3 marks) User Level Creation [ ] Works (5 marks) [ ] Attempted (3 marks)	Other extension/improvement / innovation ☑ Works (10 marks) [ ] Attempted (5 marks) Note: either the attempted marks or the works marks will be given (so you can only receive 5 marks per attempt max, and there is a max of 10 marks for this section but tick off any extra work done anyway!) For attempted marks to be awarded the feature must be at such a state, that the code could have worked but does not due to bugs or similar.	0-40

Demo self-assessment:

[cite examples where to find evidence for it in your code]

- Appropriate use of inheritance - (Example of Class in your code that inherits other class

  Superclass – GameObject/Button

  Subclasses - Player, FollowingEnemy, Gun etc./ RightButton, JumpButton etc.

- Appropriate use of Abstract classes - (Example of Class in your code that is an abstract class)

  GameObject and Button

- Use of Android API classes/methods - (Example (one or two would fine) of some API classes/methods)

  GameLoop extends Thread, Game extend SurfaceView implements SurfaceHolder.Callback

  Draw(), onTouchEvent(), onCreateOptionsMenu() etc.

- Use of standardised data structures (give some example belonging to one of these e.g., List, Vector, Collection, Date)

  ArrayList< >() – check CreateObjectsForLayout, FollowingEnemy uses vector

- Randomly/procedurally generated features (e.g., randomly adding some obstacles, some sceneries, players, etc)

  FollowingEnemy – randomly generated

- Example of AI Opponents (automatically acting objects, like automatically changing position, chasing other objects, opponent moving with the movement of player, etc.)

  All of the subclasses of GameObject inheritance that are enemies (except StaticEnemy) are AI having different functionality

- Example of Online Scoring

  I first added firebase but due to errors on libraries and gradle scripts Dr Varun advised me to use SharedPreferences for that – check LevelSelector and HomePage

- Multithreading improvements

  At the very first I tried to use multithreading and run my application in parallel but I did not manage to make it work, therefore I tried to implement other features and let it last as the game flow was smooth until now.

- Design

  Draw my own characters and needed more time to develop the animation