Lab 5

UJI Sea Battle

Introduction to Android Games

UNIVERSITAT JAUME I

Mobile Device Applications

Degree in Video Game Design and Development

1 Introduction and Gameplay

In this project, you will develop *UJI Sea Battle*, a simple graphical strategy guessing game based in the well-known Battleship[1] (or *Sea Battle*) board game. Originally a pencil and paper game which dates from the first World War, it was published as a pad game in the 1930s. Its popularity, particularly among kids and young teenagers, has inspired several versions during the last fifty years, from plastic board games in the 1960s to the popular electronic versions of the 1970s, 1980s and even 1990s —for instance, the game *"Hundir la Flota* as was branded when commercialised in Spain—. In this century you can find video games, smart device apps and even an infamous film[2] loosely based on this game.

The rules of the game[3] are well-known for almost everybody although there are several variations[4]. In your implementation you will consider these basic rules:

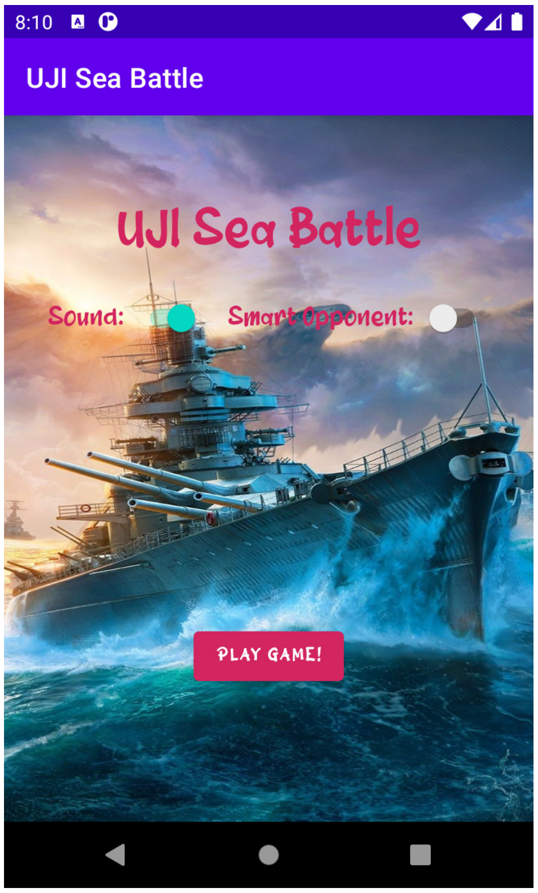
- The goal of the game is to sink all of the opponent's ships before she can sink all of yours. In your game, the opponent will be the computer.
- There will be two boards: one for the player where she will place her fleet and which will be bombed by the computer, and another one where the computer will randomly place its ships and the player's shots will be annotated.
- The boards will be 10 grid spaces both wide and high.
- Each ship must be placed horizontally or vertically across grid spaces, but not diagonally, and the ships can't hang off the grid. Ships *can touch each other*, but they can't occupy the same grid space (not even partially). The position of the ships cannot be changed after the game begins.
- The player and the computer will take alternative turns firing just one shot to attempt to hit the opponent's ships. If in her/its corresponding turn a contender hits a ship then she/it can take another shot. A turn will only be changed when a shot misses to reach a target (hits on "water").
- The fleet is composed of ten ships: **a carrier**, which takes four grid spaces, **two battleships**, which take three grid spaces each, **three destroyers**, which take two grid spaces each, and **four patrol boats**, which take just one grid space each.
- To sink a ship each player must hit all of the grid spaces that it covers. The game ends when the opponent's fleet has been completely sunk.

You can see in Figure 1 five screenshots showing the looks of the game. Figure 1 (a) shows the initial splash screen where some features of the game —sound, smart opponent— can be set. Figure 1 (b) shows the start of the game, where the player is compelled to place her fleet on her board. Figure 1 (c) shows the phase of the game where the player has finished placing her ships but she is still able to change their position on the board until the *Battle!* button is clicked. Figure 1 (d) shows a phase of the game with some ships already sunk and where it is the player's turn to shoot. Finally, Figure 1 (e) shows the end of the game (the computer's fleet has been completely sunk). As you can see, the initial splash screen is in portrait mode while the whole game is played in landscape mode.

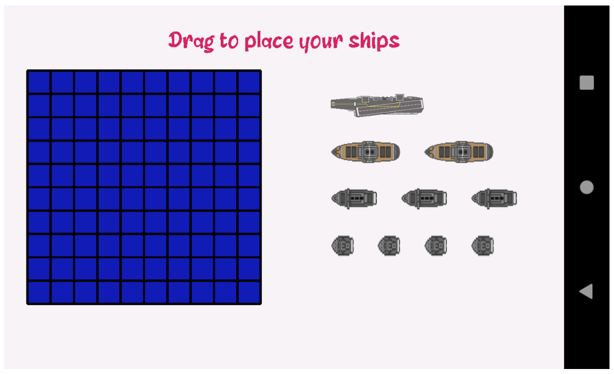
2 Overview of the Implementation's Design

From the point of view of the design, you will follow a pattern similar to the MVC (model, view, controller) one. We will have a single screen in the game, so for us, a game will be simply an `Activity` which will hold a `View`. The implementation of the game will be eased by using a simple framework (VJ1229Framework, see next section). This framework includes an abstract class, `GameActivity`, that takes care of the layout and overall settings and which inherits

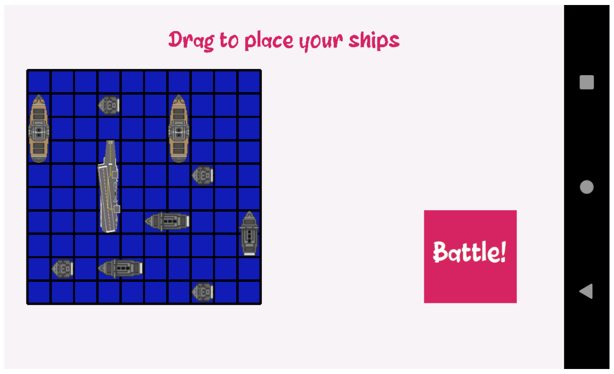
[1] https://en.wikipedia.org/wiki/Battleship_(game)
[2] https://en.wikipedia.org/wiki/Battleship_(film)
[3] https://www.thesprucecrafts.com/the-basic-rules-of-battleship-411069
[4] One of them, for instance, grants three consecutive turns to each player instead of just one.



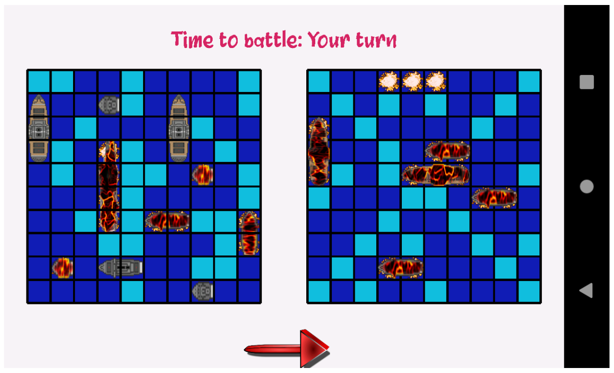
(a) Start screen



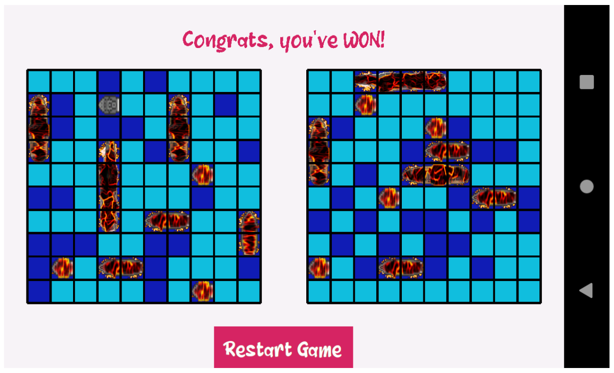
(b) Start of the game



(c) Player has placed her fleet on her board



(d) In the middle of the battle



(e) End of game

Figure 1: Five screenshots from *UJI Sea Battle*.

from the Android's `Activity` class. For instance, this class will set the orientation of the screen (landscape) so that the user will be forced to rotate her phone if she wants to play the game properly.

Its `View` is an specialised version of `SurfaceView` that is in charge of painting the state of the game and processing the touch events (we will not consider other input modes like the accelerometer). Within this framework, the programmer has to extend the `GameActivity` to provide it with a controller, which must implement the `IGameController` interface. This video in YouTube[5] (end of April, 2020) shows the use of the VJ1229 framework to develop games in Android. To ease the task of rendering, another class is used, `Graphics`, that hides some of the complications of the Android graphics module.

This means that your implementation of the game will have three main classes:

- A view that will be a subclass of `GameActivity`. In this view you will need to implement only the code to check the resolution of the display and construct a controller. The management of the `SurfaceView` is done in the base class so you don't have to worry about it.

[5]`https://youtu.be/ZMp_MrkA6mo`

- The controller, that will be a class that will implement the `IGameController` interface. This controller will be in charge of processing the input events (the movement of the fingers of the player in the screen) and drawing the bitmaps that will be sent to the view.
- The model, that will be a class that you will design completely and that will take care of modelling the boards, the ships, their position in the screen and of implementing the gameplay described in the previous section.

But note that in the case of the controller and the model it is useful, although not required, to delegate some of the task to specialised objects. For instance, modelling the boards and the ships can be done with specific classes.

3 The Framework: Initial Preparations

Start a new project with an empty `Activity`. In the project `build.gradle` change the `allprojects` section so that it looks like:

```
allprojects {
    repositories {
        google()
        jcenter()
        maven { url 'https://jitpack.io' }
    }
}
```

Then, in the app `build.gradle`, add in the `dependencies` section the line

```
implementation 'com.github.jvilar:vj1229Framework:v2021.1'
```

After changing this, sync with Gradle, either by clicking in the bar that appears in the upper part of the window ("Sync Now") or by using the option "Sync Project with Gradle Files" in the File menu.

Once the Gradle files have been synchronised, the VJ1229 framework is available to the project. You can access to its javadoc in this address[6]. Also, within *Android Studio*, if you place the cursor over an element of the framework (e.g. a class or method) and press `C-q` you have access to the documentation for that element.

As the number of classes for the model and the controller can grow, it is advisable to create both a `controller` and a `model` packages, however, this is optional.

First of all, you have to create the initial ("splash") activity which will start the `Activity` for the game. Then, this game `Activity`, as explained in theory lectures, will extend `GameActivity`[7] and will implement the method `buildGameController`. Remember that it will need to create an instance of the controller and pass to it the dimensions of the display and the `Context`, which you can recover with `applicationContext`. It is probably safe to say that you will only need to change this class when adding, at least, the switch which controls the audio of the game (in a sound/no sound basis) and pass its corresponding value, as set by the user, to the controller. As you will see later, the other switch ("smart opponent") is optional.

However, bear in mind that from API version 30 the use of `defaultDisplay` is deprecated in *Android Studio*. This is the API version used in the current release of *Android Studio*. Thus, if *Android Studio* warned you about that you should use a suppress deprecation directive. Just like this:

```kotlin
val displayMetrics = DisplayMetrics()
@Suppress("DEPRECATION")
windowManager.defaultDisplay.getMetrics(displayMetrics)

return SeaBattleController(displayMetrics.widthPixels,
                displayMetrics.heightPixels, applicationContext)
```

The controller has to implement the `IGameController` interface. This will compel you to implement the functions `onUpdate` and `onDrawingRequested` which will be used by a `GameView` (VJ1229 framework) in the update—render loop required in games. This is explained in the YouTube video[8] mentioned in the previous section. Observe that the last function has to return a `Bitmap` that can be later draw on the canvas of a `SurfaceView` (`GameView` in our framework). This `Bitmap` is achieved calling the getter `frameBuffer` from a particular instance of the `Graphics` class which is also part of the VJ1229 framework.

Bear in mind that though the initial splash activity works in portrait mode:

[6] https://javadoc.jitpack.io/com/github/jvilar/vj1229Framework/v2021.1/javadoc/
[7] This, like the rest of the classes of the framework, is in the package `es.uji.vj1229.framework`.
[8] https://youtu.be/ZMp_MrkA6mo

```
requestedOrientation = ActivityInfo.SCREEN_ORIENTATION_PORTRAIT
```

the game will be played in landscape. Use:

```
ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE
```

as the parameter for the `GameActivity` constructor.

You will end creating the classes for the Model and the Controller. For the moment, the Model will be empty. Its constructor will receive the boards for the game (player/computer) and a simple strategy —do not worry for now: we will explain this in Sections 4.3.1 and 4.3.5—. For the moment, the constructor of the Controller will receive the width and height of the screen as well as an instance of a `Context`.

4 Implementation guide

4.1 Order of implementation

As in the rest of the projects, you are free to choose the implementation order you desire. However, we feel that implementing, say first the Model and then the Controller is quite difficult because you cannot test anything until the end, unless you use the test facilities of Android, which you can find here[9], but that unfortunately we have no time to cover in the course.

Therefore, we suggest that you implement the game in order of increasing functionality:

- First, a single empty board will be placed in the screen. This will be the player's board and it will located at the left (see Figure 1 (b)).
- Second, the bitmaps for the ten ships will be drawn at the right (see again Figure 1 (b)). This will compel you to create the `Assets` singleton (object declaration) as briefly explained in Section 4.4.3.
- Third, you will start to manage touch events in the screen by letting the player move her ships around the screen (`TOUCH_UP` and `TOUCH_DRAGGED`). As you write code in the controller you will realise that a lot of functions will be required in the Model. This will cause that the code of the model class and its helper classes is completed gradually. You should end this phase implementing the code required to successfully place the player's fleet on her board (see Figure 1 (c)).
- Fourth, the computer's empty board will be placed at the right of the screen. You should now focus on writing the code required in the Model to randomly and properly place the computer's fleet on its board.
- Fifth, you should implement the gameplay but only from the point of view of the player. Do not take turns and focus only on the proper management of the spritesheet animations and the rest of the graphics, and on the hits on the computer's board. This could be a good moment to introduce sound in the game.
- Sixth, complete the game by implementing a simple strategy for the computer to try to sink the player's fleet (see Section 4.3.5). If properly implemented in the previous step you will not have to write any additional code to manage the player's board, the graphics and animations on it, and the sound. Also, you will have to implement the proper alternation of turns and the adequate ending of the game.

4.2 Display Resolution

As you know, the range of devices capable of running Android is overwhelming. And each device comes with its own hardware which means different screen sizes and resolutions. So, how can we develop a game whose graphics looks the same size in every device? Although there are some different options to address this problem, our suggestion would be to set a fixed *virtual resolution* expressed in some "virtual" units and use them to scale and properly place all of the graphical elements of the game. Obviously, these units have to be ultimately scaled to meet the real resolution of the device's display.

Following this approach we suggest that you use a grid of 24×14 (width $\times$ height), which sets an aspect ratio very close to 16:9, and work in *grid space* units. Please, see Figure 2. In this way, you can declare these constants in the controller:

```
private const val TOTAL_CELLS_WIDTH = 24
private const val TOTAL_CELLS_HEIGHT = 14
```

Remember that the real width and height (in pixels) of the device's display are passed to the controller's constructor. Therefore, the size of a grid space, henceforth, the cell side, would be *the minimum* between the quotients $width/cellsWidth$ and $height/cellsHeight$, that is:

[9] https://developer.android.com/training/testing/fundamentals

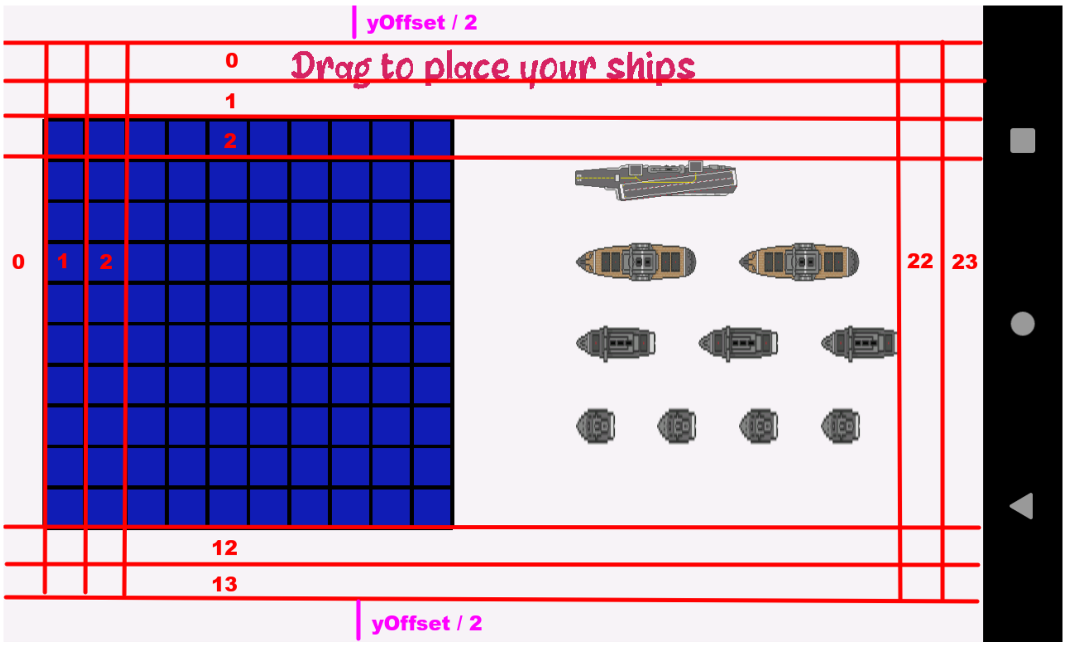


Figure 2: Virtual resolution to display the graphical items of the game.

```
private val cellSide : Float = min(width.toFloat() / TOTAL_CELLS_WIDTH,
        height.toFloat() / TOTAL_CELLS_HEIGHT)
```

However, there are many chances that the resulting size of the cell, as multiplied by the number of cells, does not exactly match the number of pixels in one of the axes. Therefore, we need to compute the (possible) *offsets* in both axes. One of these offsets will be zero —since we are playing in landscape mode, the offset in the X axis has almost every chance to be zero— and the other one will have a positive value. If we divide this non-zero offset between two, the virtual display will be centred on the screen just as Figure 2 shows.

```
private val xOffset : Float = (width - TOTAL_CELLS_WIDTH * cellSide) / 2.0f
private val yOffset : Float = (height - TOTAL_CELLS_HEIGHT * cellSide) / 2.0f
```

So, in your game, you will place your items according to grid cell coordinates. We suggest that you use a Kotlin *data class* to store the (x, y) pair of coordinates of the items in your game (boards or ships, for instance). The values of x and y could be `Int` so that these items are always "aligned" to a cell in the virtual grid. However, if you felt that you need more accuracy —for instance, to place an item just in the middle of a cell— you could use `Float` values instead.

As you can see in Figure 2, using this virtual coordinates system the top left corner of the player's board is placed at $(1, 2)$ and its bottom right corner at $(11, 12)$. To actually draw the items in the real display an inverse transformation is needed. If we had stored the player's board instance of a `Board` class in the variable `board` and we had used the class attribute `coord` to store its top left corner coordinates we would compute its real display placement as:

```
val originX : Float = board.coord.x * cellSide + xOffset
val originY : Float = board.coord.y * cellSide + yOffset
```

This is exactly what it must be done with all the graphical items of the game in order to properly draw them in the canvas provided by `GameActivity` (really a `SurfaceView`).

4.3 The Model

The model has to represent both boards, do the required actions to implement the gameplay and store the corresponding data under the command of the controller. The controller will create both boards because it is the only one that knows the exact placement of both items but their management according to the rules of the game corresponds to the model. Also, the implementation and the management of the computer's strategy to try to win the game corresponds to the model but it is the controller that knows which strategy has set the player in the splash screen.

Therefore, all these data will be passed to the constructor of the model and stored in the corresponding attributes. Obviously, you can (and must) add as many additional attributes as you deem necessary. For instance, the model will need a list or an array to store all of the ships of the player's fleet that she can place on her board. The model will manage this list by removing the ships that are placed inside the board and copying them in the list of ships managed by the player board's instance.

It will also be advisable for the model to store a list of cells (or coordinates) where an animation (a explosion or a water drop, for instance) has to be played by the controller. This information is only known by the model as the gameplay evolves.

These are clear examples of what we mean. In any case, we think that some helper classes will be needed. If this seems right for the `Board` and `Ship` classes it could not be the case for the `Cell` and `Coordinates` classes. We suggest that you implement them but this will depend on your final implementation and the design of your code.

4.3.1 The board

The board is an important object in the game. The player uses a board to place her fleet and another board to annotate her shots. The latter is really the computer's board, i.e. the board where the computer has hidden its fleet.

From a practical point of view, a board will hold a list (or an array) of ships and will have to store the information regarding each cell's state —for instance whether a given cell has been bombed—. In fact, the cells of the board can be seen as a matrix that can be implemented in different ways in Kotlin. Therefore, we strongly encourage the implementation of a `Board` class.

Its constructor could receive the coordinates where the top left corner of the board is placed, the number of cells wide and high, and the size of the cell, that is, the `cellSide` as computed in Section 4.2.

Apart from that, this class will implement the functions required to manage the board and all of its contents. For instance, it could provide functions to:

- Tell whether a given pair of coordinates lies inside the area of the virtual display covered by the board.
- Get the cells where a given ship is placed.
- Store given data in the cells where a ship is placed.

Of course, and depending on your final implementation, you will need to create more public or private functions (not much more) but we feel that the ones mentioned serve as an example of functionalities that will be required to implement.

4.3.2 The ship

A ship is another important object in the game. Ships are dragged, clicked on, drawn, bombed, destroyed, and so on. You will need to implement the `Ship` class to store its data. For instance, its constructor could receive the coordinates of the ship[10], its length (expressed as the number of cells that covers), and a Boolean telling whether the ship is placed vertically or horizontally in the screen.

However, an important question arises: Which part of the ship do the coordinates reference? We need to set a reference to a point or part of the ship to place it. Our suggestion is that you employ the top left corner of the "box" that bounds the bitmap used to draw it. Therefore, the coordinates of the ship will match those of the cell in the virtual grid (see Figure 2) where the top left corner of its bitmap is drawn. Thus, a ship of length 3 (a battleship) with $(3,5)$ coordinates would cover the cells $(3,5)$, $(4,5)$, and $(5,5)$ if horizontally placed. Otherwise, it would cover $(3,5)$, $(3,6)$, and $(3,7)$.

This reference point (coordinates of the ship) should be used also to change the orientation of the ship. Therefore, the ship would rotate 90 degrees clockwise to change from horizontal to vertical orientation and 90 counterclockwise otherwise.

Instead of using only a bitmap to draw the ship —or actually two, the second being employed to draw a sunk bombed ship— we suggest that you employ four different bitmaps as shown in Figure 3. Proceeding this way you will not need to rotate images drawn in the canvas[11].

If you follow our recommendation then you will need attributes in the `Ship` class to store both this(these) list(s) or array(s) of `Bitmap` and the active bitmap to be drawn, which will be different according to the ship's condition: vertical/horizontal orientation and sunk/not sunk. Also, an attribute —maybe in the constructor— to store the orientation of the ship (vertical/horizontal) will be required.

4.3.3 The cells of the board

There are several strategies that you could follow to implement the storage of the board cell's data:

- its coordinates,
- whether (part of) a ship is placed on it,
- whether it has been bombed,

[10]When on a board, ship's coordinates will always match the coordinates of the cell where its reference part is located.
[11]Something not allowed by the VJ1229 framework by the way.

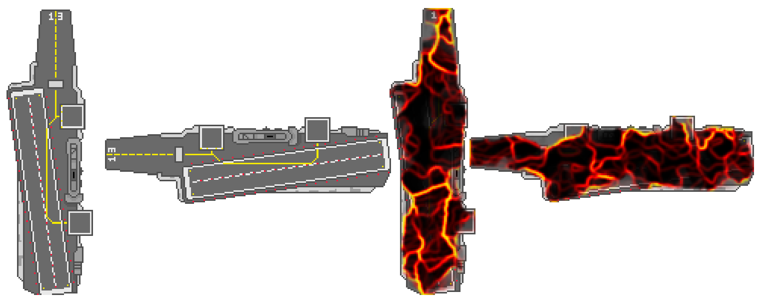


Figure 3: Several bitmaps are stored to draw a carrier but only one is the active, i.e. the currently used to draw the ship.

and maybe more (according to your implementation).

Although not strictly necessary, we recommend that you declare a Kotlin *data class* to easily store these data. As you know from the initial theory lectures on Kotlin, the use of Kotlin's data classes grants some interesting advantages when you have to manipulate, compare or assign instances of them.

4.3.4 The states of the Game

As you can infer from Figure 1, *UJI Sea Battle* has different phases. In this figure you can find at least three —and another one not so obvious—:

- An initial phase where the player drags the ships to place them on her board.
- A second phase where the battle starts. This phase is not really a single phase. It is indeed two phases: the obvious —you can read it in Figure 1(d)— phase for the player's turn and the not so obvious phase for the computer's turn, since somewhat different actions should be carried out in each phase.
- The end of the game.

This reveals that *UJI Sea Battle*, and hence the Model of the game, goes through a series of possible states. Besides the four mentioned above, you will need states when dragging ships inside or into the board (when placing them) or when waiting to complete the animations between turn changes. You can easily check this by watching the video which shows the working of *UJI Sea Battle* in *Aula Virtual*.

Of course, you are free to decide the states of the game that best fit your ideas for the implementation. However, we feel that, at least, the use of these states will ease the implementation of both the controller and the model. In the model, you could implement something like this:

```
  enum class SeaBattleAction {
    PLACE_SHIPS,
    DRAG_INTO_BOARD,
    DRAG_INSIDE_BOARD,
    PLAYER_TURN,
    COMPUTER_TURN,
    WAITING,
    END
}
```

and use an attribute to store the state of the model at any time. This attribute should be public but its value should only be changed by the model:

```
  var state = SeaBattleAction.PLACE_SHIPS
        private set
```

Observe that this attribute of the model is initialised to `PLACE_SHIPS` which is the initial state of *UJI Sea Battle*.

4.3.5 Simple Strategy

For *UJI Sea Battle* to work, you will have to implement a simple strategy so that the computer is able to shot in order to sink all of the player's ships.

For our academic purposes it will be enough for you to implement a basic strategy that randomly chooses coordinates of cells in the player's board. In this sense, you could simply create a list or array with the coordinates of all the cells in the player's board and randomly pick one among them —obviously one different at a time— when in computer's turn to shot.

However, there is something that you will have to implement to make this strategy not so basic. In case that a computer's shot hit a ship you should make an educated guess for the next shot instead of a random choice.

This poses an important question regarding not only intelligent but even simple strategies to locate hidden data with specific features: the need for *feedback*. A strategy will not only provide a public function to return the coordinates of a shot but also a public function to get the result of its shot.

Obviously, you could create only a class, let's say `SimpleStrategy` and implement these two functions. However, if you thought of possible enhancements in the intelligence of the game (see Section 4.5) you should be prepared to easily add new classes to your implementation. Therefore, we recommend that you implement a Kotlin interface, `StrategyInterface`:

```kotlin
interface StrategyInterface {
  val playerBoard : Board
  fun computerGuess(...) : Coordinates
  fun shotResults(...)
}
```

Thus, your class `SimpleStrategy`, by implementing this interface, will be compelled to implement both functions at least. Also, the constructor will have to receive the player's board to be able to create the list of valid cell coordinates. Of course this attribute could be omitted in the interface or replaced with something different according to your implementation. Observe that you will have to replace the dots (...) with the parameters that you deem necessary (or even none) in both functions. Therefore, if you decide to implement a *smart* strategy (see Section 4.5), you will create another class which implements this interface.

In any case, bear in mind that the controller will create an instance of the strategy class and pass it as an attribute of the model's constructor since the functions provided by the strategy will only be called by the model.

4.4 The Controller

The controller has two main entry points, the `onUpdate` and `onBitmapRequested` methods, as well as a constructor. Most of the code of its constructor will be dedicated to prepare the variables needed for drawing the bitmap and to create some instances to be passed to the model (for instance, the boards and the strategy).

4.4.1 The `onUpdate` Method

This method receives a float, `deltaTime`, that contains the time (in seconds) elapsed since the last call, and a list of touch events that have happened also since the last call. The controller must then follow these steps:

- Process the touch events according to the model's state to detect either clicks on boards or ships or the dragging of ships.
- Call the corresponding methods of the model either to inform it of the events happened or to update the data which manages.
- Perform the actions required to draw the stage according to the model's state.
- Detect when the game has ended to be able to manage the start of the initial splash activity just in case the player wanted to restart the game.

The third and fourth steps are not complicated to implement, while the second will depend on your ideas for the implementation of the model. But the first probably needs some more detail. The idea is to traverse the list of events and for each event:

- If it is a `TOUCH_UP` event, check the model's state. If the player had been dragging a ship into the board, you should inform the model of the coordinates so that it can check whether the ship lies inside the board or not and carry out the corresponding actions. If the player had been dragging a ship inside the board you should do the same. But if the player had not been dragging any ship, the controller should check:
  - Whether the coordinates lie inside the space covered by the bitmap for the *Battle!* button but only in case the player had been done placing her fleet. If this were the case, the controller should prepare for the battle phase and call the corresponding methods of the model (for instance, to create and place the computer's fleet).

  - If the player had not touched the *Battle!* button, the controller would pass the coordinates to the model so that it can check whether a ship on the player's board has been touched. In this case, the model had to change the orientation of the ship if possible.

  When *UJI Sea Battle* is in the battle phase, and more concretely, the model's state signals the player's turn, the controller will pass the coordinates to the model so that it can perform the required actions for a player's shot.

  Finally, if the game had ended, the controller had only to check whether the player has touched the restart button. In this case, the controller would start the initial splash activity.

- If it is a `TOUCH_DRAGGED` event, the controller has to check whether a ship is already being dragged or not. If not, it will inform the model of the start of a dragging action and pass the coordinates. Otherwise, it will pass the coordinates to the model so that it can monitor the action and take the corresponding actions.

After processing the list of events, the controller should perform the actions required on this update phase of the update-render loop. In short, the controller should check whether the model is in a state that do not require user's actions to which has to respond like the `WAITING` or `COMPUTER_TURN` states. In this case, it has to call the corresponding methods of the model. Also, the controller will update with `deltaTime` the animations currently playing, if any.

Important reminder: observe that the coordinates of the touch events are computed using the *real* display resolution of the device. Therefore these coordinates have to be translated to the game's virtual coordinates (see Figure 2) before being processed by the controller or sent to the model. Just examine the "skeleton" for `onUpdate` below:

```
override fun onUpdate(deltaTime: Float, touchEvents:
                  MutableList<TouchHandler.TouchEvent>?) {
   if (touchEvents != null) {
      for (event in touchEvents) {
         val correctedEventX : Int = ((event.x - xOffset) / cellSide).toInt()
         val correctedEventY : Int = ((event.y - yOffset) / cellSide).toInt()
         [...]
         when (event.type) {
            TOUCH_UP -> {
                     // ACTIONS FOR TOUCH_UP
            }
            TOUCH_DRAGGED -> {
                    // ACTIONS FOR TOUCH_DRAGGED
            }
         }
      }
   }
   // ACTIONS WHICH NOT DEPEND ON USER'S TOUCHES (UPDATES)
}
```

4.4.2 The `onDrawingRequestedMethod`

In this method you will have to draw the message in the header, the cells on the player's board where a ship can land when is being dragged, the player's board, the computer's board if required (according to the model's state), the remaining ships to be placed by the player on her board if required, and the bitmaps for the buttons that move to the battle phase and that let the player restart the game, respectively. To do this, you will have to use a `Graphics` object that you can create in the constructor with the appropriate dimensions (the real width and height of the device's display). The first action then will be to clear this graphics with the background colour.

We suggest that you implement a function to draw the board that is passed to it. You could use two Boolean arguments to control whether the contents of the cells bombed and the ships are drawn or not, but this is not strictly necessary. Observe that you will have to traverse the board to draw the grid (vertical and horizontal lines). Use two `for` loops, one for the rows, another one for the columns, to this end.

For the buttons, simply use drawables or bitmaps and place them in the graphics using `drawBitmap` or `drawDrawable` as appropriate.

After drawing the boards, do not forget to draw the updated animations (for explosions or splashes) but only in the battle phase. Use `drawBitmap` and `currentFrame`, which belongs to the `AnimatedBitmap` class.

Remember to scale the buttons and the rest of the elements according to the size of the screen and use the ideas explained in the theory lecture for positioning them.

4.4.3 Assets

The sprites (images) used in the game have to be added to the project in *Android Studio*. You could drag and drop your image files (PNG, JPEG. . .) on the `drawable` package in *Android Studio* but we recommend that you create a folder named `drawable-nodpi` inside the `res` folder of the project using the *File Explorer* of your system and copy all of the image files to it. *Android Studio* will be aware of the changes made to the hard disk almost immediately and will add the files to the project inside the `drawable` package (adding the comment `(no dpi)`). However, **it is mandatory** that the names of the files are written in **lowercase letters** only (not uppercase, dashes, underscore or any kind of "special" character).

You can download free assets for your games from OpenGameArt[12], Craftpix.net[13], or Free Game Assets[14], for instance.

As commented in Section 4.1, as part of the Controller (that is, created in the same package) you will create the `Assets` singleton (object declaration).

```
object Assets {
  const val CARRIER_LENGTH = 4
  [...]
  private const val SPLASH_ROWS = 2
  private const val SPLASH_COLUMNS = 4
  private const val SPLASH_WIDTH = 62
  private const val SPLASH_HEIGHT = 33
  [...]
  var horizontalCarrier : Bitmap? = null
  [...]
  private var splash : SpriteSheet? = null
  var waterSplash : AnimatedBitmap? = null
  [...]

  fun loadDrawableAssets(context: Context) {
    val resources : Resources = context.resources
    [...]
    if (splash == null) {
      val sheet = BitmapFactory.decodeResource(resources, R.drawable.splash)
      splash = SpriteSheet(sheet, SPLASH_HEIGHT, SPLASH_WIDTH)
    }
    [...]
  }

  fun createResizedAssets(context: Context, cellSize : Int) {
    val resources : Resources = context.resources
    [...]
    horizontalCarrier?.recycle()
    horizontalCarrier = Bitmap.createScaledBitmap(
       BitmapFactory.decodeResource(resources, R.drawable.shipcarrierhullhoriz),
       cellSize * CARRIER_LENGTH, cellSize, true)
    [...]
    val frames = ArrayList<Bitmap>()
    waterSplash?.recycle()
    for (row in 0 until SPLASH_ROWS)
      splash?.let { frames.addAll(it.getScaledRow(row, SPLASH_COLUMNS, cellSize,
        cellSize)) }
    waterSplash = AnimatedBitmap(2.0f, false, *frames.toTypedArray())
    [...]
  }
}
```

In the code above you can see that a `Bitmap`, `horizontalCarrier`, for the image of a carrier with horizontal orientation is declared, created, and resized according to the size of the cell side of the game's virtual resolution (see

[12] `http://opengameart.org/`
[13] `https://craftpix.net/`
[14] `http://www.gameart2d.com/freebies.html`

Section 4.2).

To complete our example code, we have also declared, created, and properly resized an `AnimatedBitmap` from a `drawable` sprite. The spritesheet for the animation has 8 frames (two rows, four columns) where each frame is 62×33.

First, the `SpriteSheet` is loaded and then an `AnimatedBitmap`, `waterSplash`, is created from its *resized* frames. As you can see, the animation will span 2 seconds and will not keep looping.

Both functions (`loadDrawableAssets` and `createResizedAssets`) will be called from the constructor of the controller (you could do it in the `init` block). From the moment these functions are called the assets of the game will be available through `Assets` attributes.

4.4.4 Restarting the game

To restart the game you will only have to create an `Intent` to start again the initial splash `Activity`. However, you can only create an `Intent` to start an `Activity` inside of other `Activity`. Therefore, the controller can't create the intent to directly start the initial activity.

The solution consists in creating an interface, let's say `StartInterface`, which declares only a function (for instance, let's say `reStartActivity`) and that the class that you created to extend `GameActivity` implements also this interface. Thus, you will have to write the code in that class for `reStartActivity`: simply create and run the `Intent`.

Then, it will be needed to add a new attribute to the constructor of the Controller (let's say `gameStart`, for instance) of type `StartInterface`. The activity that extends `GameActivity` will pass itself (`this`) when creating the instance of the controller class. Thus, the controller will restart the game by calling:

```
gameStart.reStartActivity()
```

4.4.5 Sound

For this functionality you should use the `SoundPool`[15] class as explained in the slides about audio in Android.

The basic modifications you must do are to create an interface in the model, say `SoundPlayer`, that has a method for each of the possible sounds. Then the controller will implement this interface. This means that it will need to have a `SoundPool` as an attribute and in each of the methods, the controller will use that `SoundPool` to play the corresponding sound. Remember to load the sounds in the constructor (in Kotlin, you could use `init`).

The model will have an additional attribute (and the corresponding parameter in the constructor) of type `SoundPlayer`. Obviously, that parameter will be the controller. Note that it is important that the type of the parameter is `SoundPlayer` and not `Controller` (or whatever the name you give to the controller).

There are many pages with free sounds, for instance we have used OpenGameArt[16] and Freesound[17]. Bear in mind that the sounds that you finally use in *UJI Sea Battle* will have to be stored as *raw* resources.

4.5 Optional: Smart Opponent

Although it might sound weird, the problem of designing an strategy to maximise the chances of winning a battleship game has been addressed in Computer Science. You can read a full analysis of this question in this blog entry[18] or in this article[19].

Bear in mind that no smart strategy can guarantee to win each game but smart strategies are compared in a great series of games so that the probability of winning is computed. Also, smart strategies are compared in terms of the average[20] number of shots that they take to win the games.

Of course, the problem is slightly different according to the distinct variations in the gameplay. For instance, it is easier to design an smart algorithm if the ships can't be adjacent. Anyway, most of the devised strategies share some ideas in common:

- First, they perform several shots following a pattern that tries to divide the grid space in regions according to the dimensions and number of ships placed in the board. Obviously the shots in this pattern are randomised to avoid being predictable.
- Second, they try to sink the greatest ships first (ships of length one are regarded as random shots). To this end, the consecutive free spaces (horizontal and vertical) of regions are analysed.

[15] `https://developer.android.com/reference/android/media/SoundPool`
[16] `http://opengameart.org/`
[17] `https://freesound.org/`
[18] `https://www.datagenetics.com/blog/december32011/`
[19] `https://paulvanderlaken.com/2019/01/21/beating-battleships-with-algorithms-and-ai/`
[20] And other statistics.

- Third, when a ship has been hit they try to sink it by inspecting the cells around and determining, when possible, which its orientation is so that they can minimise the number of shots required to sink a hit ship. Take into account that the fact that ships can be adjacent hinders the implementation of this approach.

As an optional part for your version of *UJI Sea Battle* we propose that you implement a smart strategy that takes into account these considerations. If you implemented this strategy successfully you could achieve until two extra points (see next section).

5 Submission and Grading

It is **required** to submit the code of the project to get a grade. Those projects whose code is not delivered **will not be graded** although the PDF memory has been submitted. The code of the project **must be built** in *Android Studio* without errors and the app **must** work without significant crashes. If the code can't be built or the app crashes too much your project will be graded 0.

You are expected to store your project in a **git repository**. Therefore, the submission of its code will consist in making it available to us. Use a public service like github[21] or bitbucket[22] to this end. Your repository **must be private** but you will have to grant read access to jcamen@lsi.uji.es, which is valid both in *bitbucket* and *github*, so that we can grade your work. Should you have used a different page to store your repository, please contact with jcamen@uji.es. **Any project whose code is in a public repository will be graded** 0.

This is **the only means to submit the code** of your project. Any other means, like a shared folder in *Google Drive* or a ZIP archive submitted to the task in *Aula Virtual*, will not be considered and thus your project will be graded 0. Please, do not bother us with e-mails or messages at this respect: your time and ours is very valuable.

We will provide a task in *Aula Virtual* to submit a small memory in PDF giving a brief overview of the work done, the design decisions and the difficulties found (four pages at most). It is also **required** to submit this memory, which will be considered the submission of the project. Anyway, keep in mind what we have said above about the code of the *Android Studio* project that you have developed. You should take into account the following points:

- The submission period ends at 23.59.59 of the day before the beginning of the next project, which will be the *final (free) project*. The procedure detailed in the *Assessment Regulations* document will be applied to delayed submissions.
- You can submit your work either individually or by pairs. In the case of pairs, only one member of the pair must submit the project.

Please, take into account the following when writing the memory:

- The design decisions include those aspects that are left open in the implementation like the number of helper classes employed in the model, the number of states of the model, the management of model's state changes, the proper random placement of the computer's fleet, the exact details of the implementation of the computer's simple strategy and so on.
- In case the project has been done by a pair, the memory must include the name of both members of the pair.
- It must also include the id and address of the last commit corresponding to the version of the project submitted for grading.

The submission will be graded with up to:

- One point for the memory. Bear in mind that the use of bad English will be penalised.
- Half point for the splash screen whenever the settings are properly passed to the game activity.
- **Provided that** the part of the game which lets the user place her fleet onto her board **works fine**, three points and a half will be granted.
- Four points for the proper working of the gameplay, i.e., a correct random placement of the computer's fleet, the right alternation of turns, the proper selection of targets by the computer when a player's ship has been hit (but still not sunk), and the employment of adequate graphics and animations.
- One point for the implementation of sound in your game, as described in section 4.4.5. Remember that in the splash screen the sound can be activated or deactivated by the user at her will.

[21]`https://github.com/`
[22]`https://bitbucket.org/product/`

Up to **two extra points** will be granted if you implement also a **smart strategy** (smart opponent switch in the splash screen). In this case, you will have to describe the implemented algorithm in a page of your memory at most. Observe that, in any case, the maximum mark that can be achieved in the project is 10. However we will consider projects with a real mark higher than 10 to grade honour distinctions.

Take special care to avoid that your application crashes anytime. It is better to have less functionality correctly implemented than to have an application that crashes.