

# INDICE

<b>MENU:</b>	3
<b>INIZIO DELLA BATTAGLIA</b>	4
<b>SHOP</b>	6
<b>SEZIONE ASTRONAVI</b>	7
<b>SEZIONE ARMI</b>	11
<b>PARTE DI CODICE E DI FUNZIONAMENTO</b>	16
<b>DatabaseHelper.java</b>	16
<b>Storage.java</b>	19
<b>Costants.java</b>	20
<b>DrawableCostants.java</b>	20
<b>Player.java</b>	21
<b>Enemy.java</b>	22
<b>EnemyLife.java</b>	24
<b>ShieldPlayer.java</b>	25
<b>Shield.java</b>	26
<b>Bullet.java</b>	27
<b>ExplosionPlayer.java</b>	28
<b>ExplosionEnemy.java</b>	31
<b>MainActivity.java</b>	32
<b>Costruttore e inizializzazione gioco</b>	32
<b>START (AVVIO DELLA BATTAGLIA)</b>	33
<b>SHOP (AVVIO DELLO SHOP)</b>	36
<b>Descrizione (Sezione Astronavi)</b>	37
<b>Pulsanti Seleziona e Compra (Astronavi e Armi)</b>	39
<b>BackButton e ExitG (Per uscire dalle sezioni astronavi e armi)</b>	47
<b>GamePanel.java</b>	49
<b>hitCheckBulletEnemy()</b>	52
<b>hitCheckShieldPlayer()</b>	56
<b>hitCheckShieldEnemy()</b>	57
<b>hitCheckEnemyPlayer()</b>	58
<b>SpawnEnemy() DeleteEnemy() DeleteEnemyLife()</b>	60
<b>SpawnBullet() DeleteBullet()</b>	62

# VERBATIAM

Abbiamo realizzato questa applicazione che propone un videogioco dove si ha la possibilità di scegliere diverse **astronavi** per sconfiggere i nemici incontrati nel corso della battaglia.

Il gioco si presenta con un menù in cui è possibile *avviare la partita, entrare nello shop* oppure *uscire dal gioco*.

Cliccando **Start** la partita sarà avviata e la battaglia avrà inizio.

Lo **Shop** ci propone 2 sezioni:

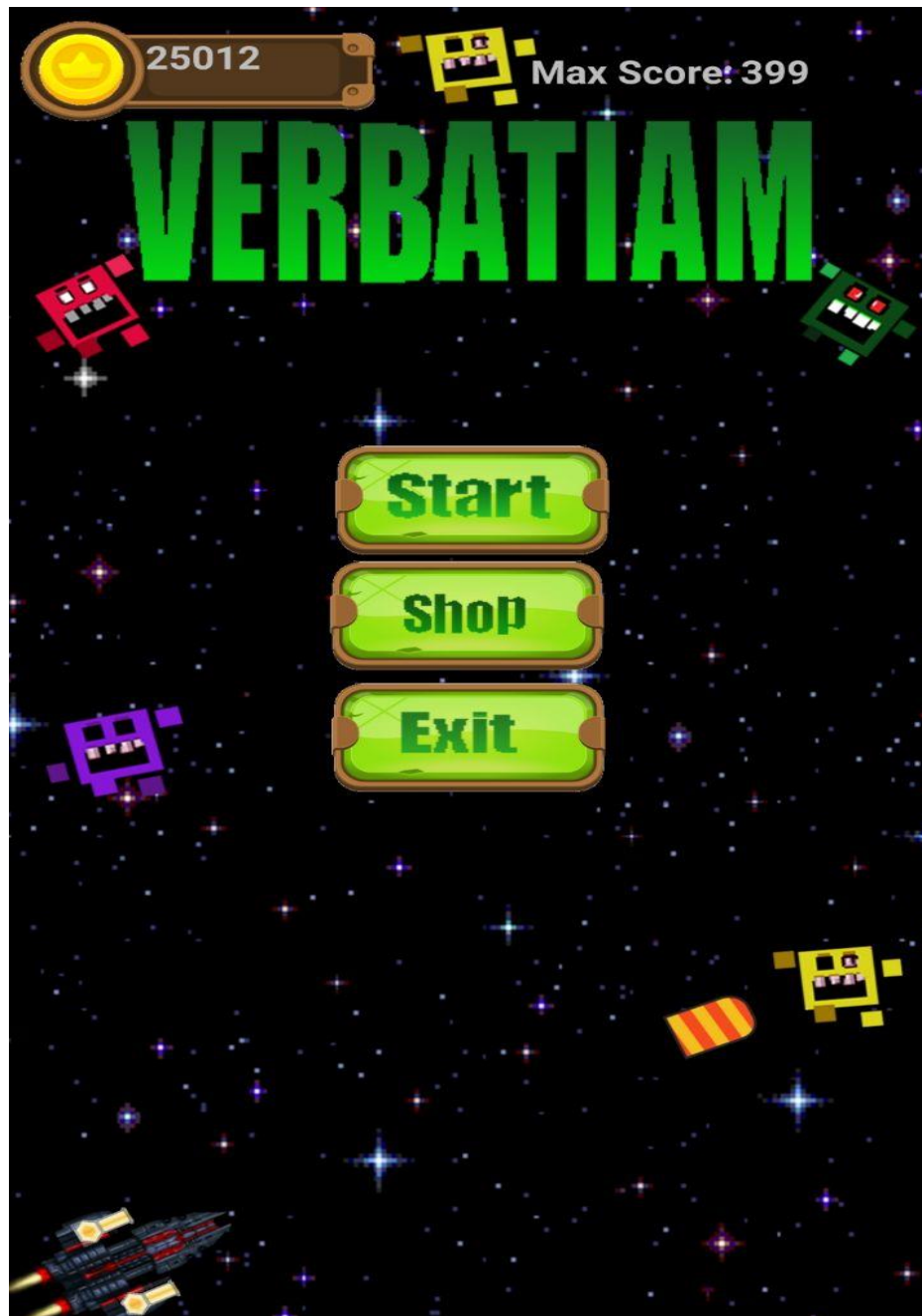
- **La sezione delle astronavi** presenta 3 astronavi: **GreenSpaceship**, **RedSpaceship**, **Ultimate**.  
Qui Il giocatore può acquistare l'astronave desiderata in base ai **Coin** ottenuti nelle battaglie.
- **La sezione armi** presenta i vari tipi di armi che possono essere equipaggiate all'astronave selezionata. Le armi sono predisposte in ordine di potenza.

Cliccando **Exit** si esce dall'applicazione.

Inoltre il **menù principale** offre la possibilità di vedere i **Coin** raccolti e lo **Score** massimo raggiunto.

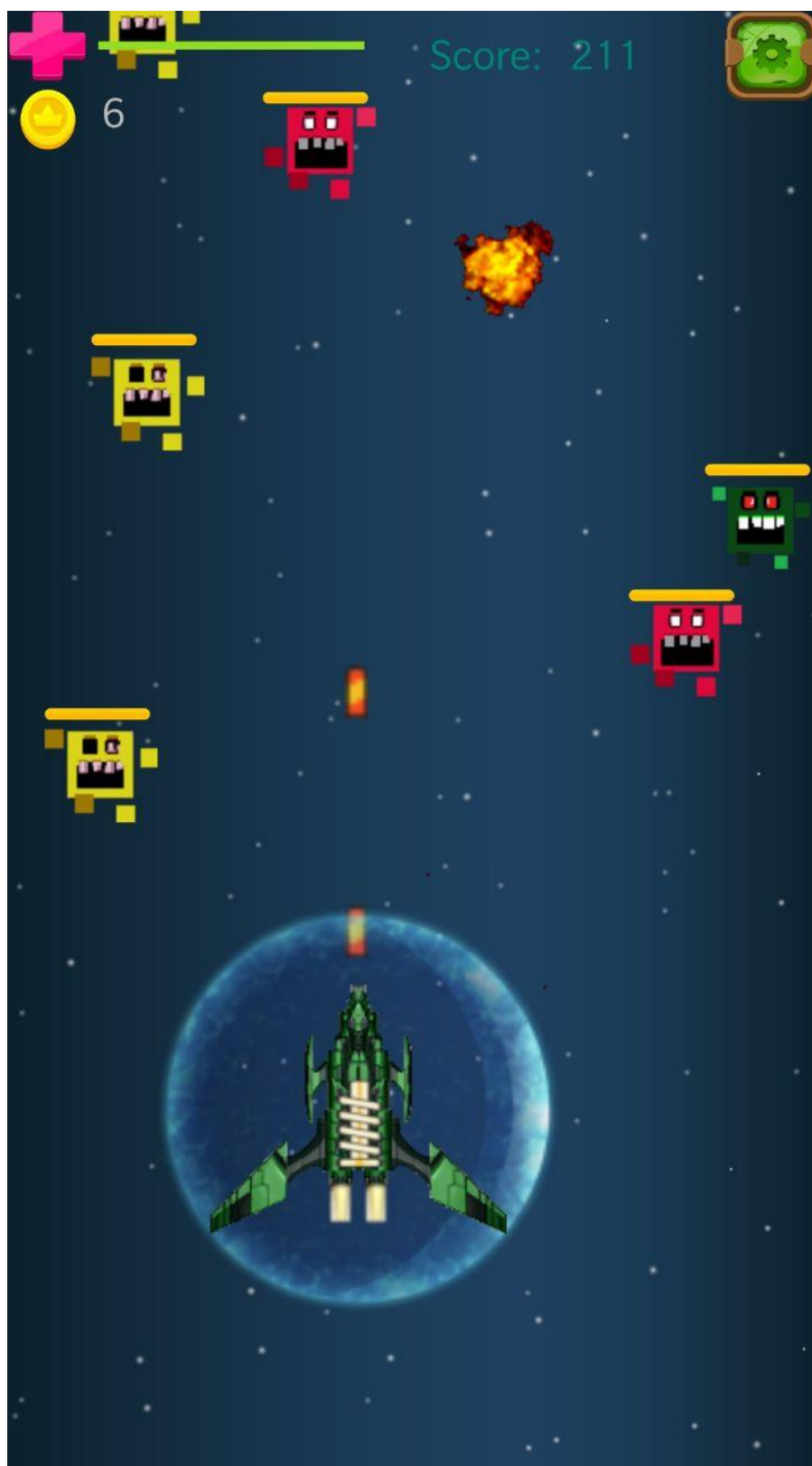
Lo scopo di questo gioco è quello di sopravvivere il più possibile all'invasione dei mostri cercando così di ottenere uno **Score** sempre più alto.

## MENU:



ORA VERRANO SPIEGATE LE VARIE SEZIONI DEL MENU:

## INIZIO DELLA BATTAGLIA



La battaglia offre come scenario lo spazio in cui avviene lo scontro tra l'astronave e il cosiddetto tempestivo attacco degli **Scrool**.

All'inizio della battaglia il giocatore sarà libero di muovere l'astronave selezionata in tutto il display; nel frattempo inizieranno a spawnare gli **Scrool**, di vario colore, sul campo di battaglia. L'astronave è dotata di sparo automatico e le armi equipaggiate continueranno a sparare proiettili in una direzione rettilinea ben precisa. Grazie a questo tipo di sparo il giocatore non si dovrà preoccupare di sparare durante la partita. Ma dovrà prestare attenzione a non essere colpito perché ad ogni collisione con un mostro l'armatura dell'astronave si danneggia. In alto a sinistra troviamo la barra degli **hp** che diminuirà ogni volta che l'astronave colliderà con uno **Scrool**. A seconda degli **hp** che abbiamo durante la partita la barra assume un colore differente. **Verde** quando abbiamo una buona salute o meglio il 100%, **giallo** quando abbiamo il 50% e infine **rosso** quando perdiamo. All'inizio della partita si avvia un timer che ci permetterà di osservare quanto riusciamo a resistere. Lo spawn è composto da 4 tipi di mostri, ognuno con un colore diverso (Rosso, Giallo, Verde, Viola). Ogni **Scrool** ha una propria vita che si incrementa con il passare del tempo. Inoltre ogni 15 secondi viene incrementata la **velocità**, con cui i mostri scendono, e il **danno** causato. Quando il proiettile sparato dall'arma dell'astronave colpisce un mostro la sua vita cala e quando arriva a 0 si verifica un'esplosione. Il mostro viene eliminato e il giocatore avrà la possibilità di accumulare 2 **Coin**. Con il passare del tempo il gioco diventa sempre più difficoltoso perché la velocità dei mostri aumenta sempre di più e anche il danno provocato; oltre a questo la vita di ogni mostro aumenta tanto che l'astronave base non sarebbe più sufficiente per farci continuare a giocare. È necessario così, cercare di guadagnare il maggior numero di **Coin** ad ogni partita per poter potenziare e ottenere nuove astronavi e armi più forti.

Tuttavia nel corso della battaglia il gioco offre alcuni aiuti, man mano che lo score diventa sempre più alto insieme ai nemici spawnano delle palline azzurre con il simbolo di uno shield. Se il giocatore saprà sfruttare al meglio l'occasione e riuscirà a collidere con questa pallina potrà ottenere la possibilità di subire una collisione in più senza danneggiare la propria armatura.

Ogni volta che si perde il gioco ci offre la possibilità di vedere i **Coin** che abbiamo e anche quelli che abbiamo guadagnato nella partita stessa.

Quando si inizia a giocare per la prima volta saremo costretti ad usare l'astronave base (**GreenSpaceship**). Non ci consentirà di ottenere grandi risultati o grandi record ma ci permetterà di ottenere **Coin** sufficienti per acquistare armi e astronavi più forti che ci garantiranno maggior tenuta e di conseguenza di ottenere un record migliore.

Questo gioco non ha una vera e propria fine perché è una continua sfida per migliorare sempre di più e per vedere il numero dello **Score** sempre più alto. Inoltre la curiosità di scoprire le varie abilità delle altre astronavi e il sogno di controllare un'astronave più bella e più forte porterà il giocatore a giocare appassionatamente per raggiungere i propri obiettivi.

## SHOP

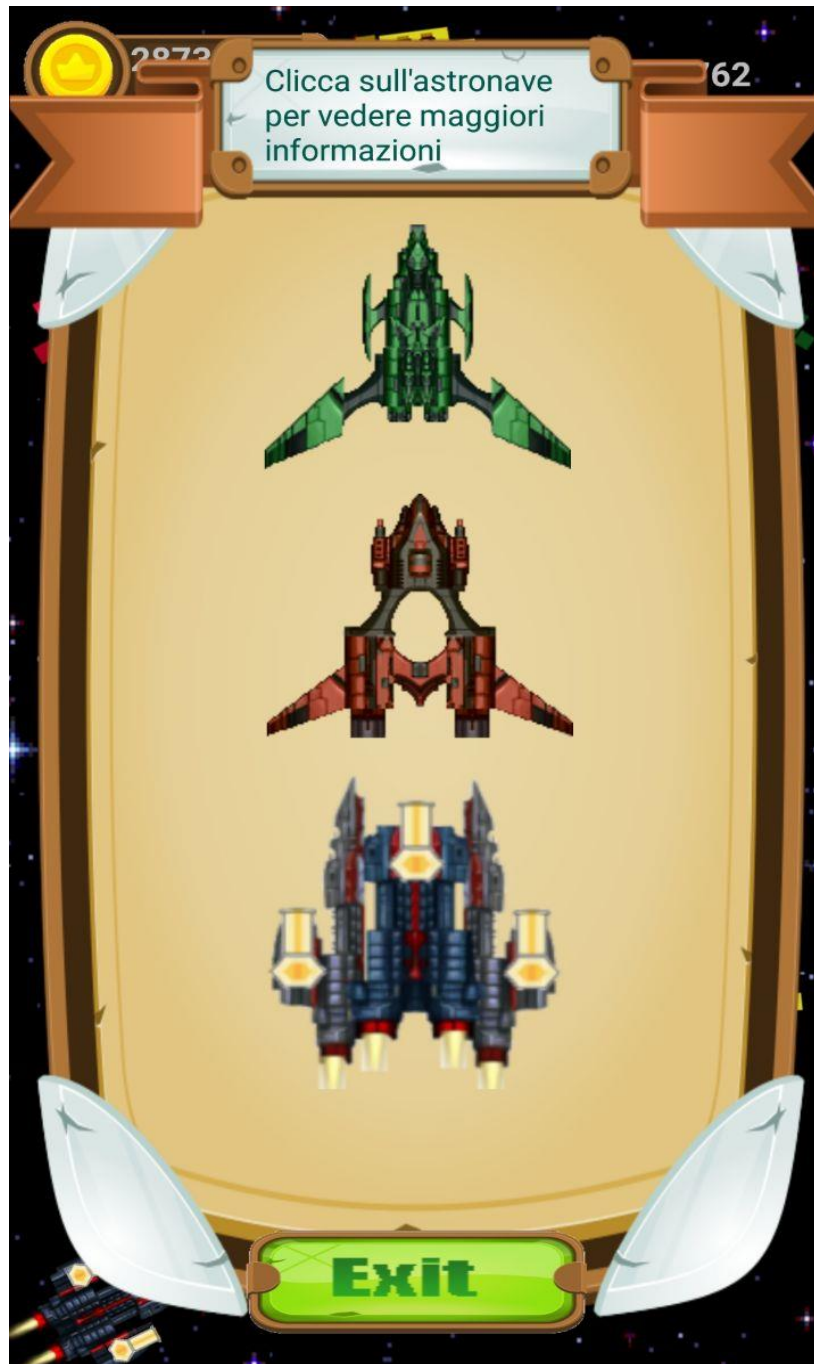
Lo Shop ci propone la possibilità di entrare in 2 sezioni differenti ognuna con uno scopo ben preciso.

La Sezione Astronavi e la Sezione Armi che ora verranno descritte nello specifico.



## SEZIONE ASTRONAVI

La Sezione Astronavi ci mette di fronte a 3 tipi di astronavi (**GreenSpaceShip**, **RedSpaceShip**, **Ultimate**) ognuna di queste con caratteristiche differenti. Per ogni astronave è disponibile una descrizione in cui vengono descritte le sue caratteristiche specifiche, la storia, il valore della vita e il costo per effettuarne l'acquisto.





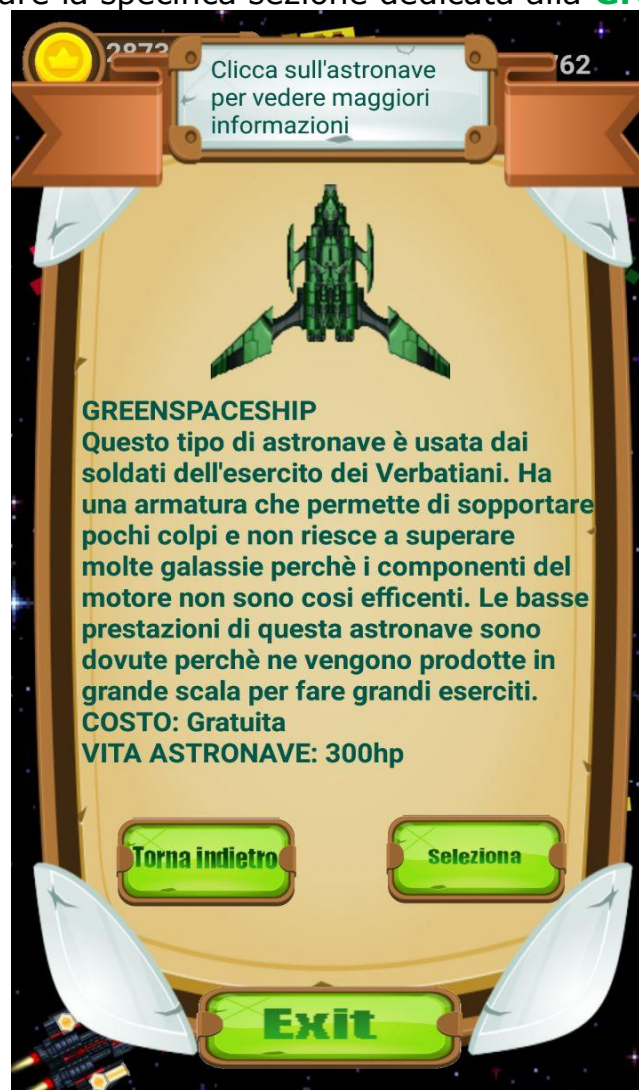
## GREENSPACESHIP



La **GreenSpaceShip** è l'astronave base ed è quella che controlleremo all'inizio. L'astronave ha una vita pari a 300 **hp** e non è necessario comprarla ma quando avviamo il gioco è già selezionata di default. A differenza delle altre astronavi più evolute la **GreenSpaceShip** può equipaggiare una sola arma. Questo particolare ci creerà alcuni svantaggi quando saremo ad uno **Score** piuttosto elevato perché lo sparo dei proiettili in una sola traiettoria non basterà a sconfiggere tutti i nemici.

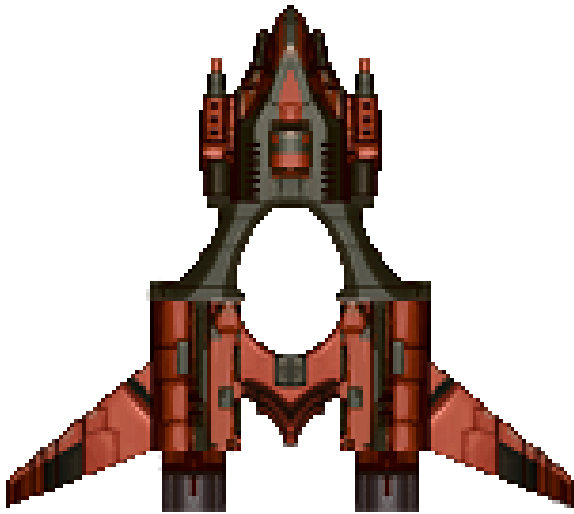
Inoltre la vita non sarà abbastanza sufficiente per i danni sempre maggiori.

Qui possiamo osservare la specifica sezione dedicata alla **GreenSpaceShip**:





## REDSPACESHIP



La **RedSpaceShip** è un'astronave più moderna rispetto alla **GreenSpaceShip**. La sua vita è di 800 **hp**. Il prezzo di questa astronave è di 8000 **Coin**. La particolarità che la distingue dalle altre astronavi è che riesce ad equipaggiare 2 armi. Questo particolare gli consente di sparare proiettili in due direzioni rettilinee e quindi di sconfiggere il doppio dei nemici rispetto alla **GreenSpaceShip**. La vita migliorata insieme alla seguente abilità ci consentirà di ottenere risultati più alti e soddisfacenti. Ma tutto ciò non basta, se vorresti veramente dominare lo spazio l'astronave giusta è l'**Ultimate**.

Qui possiamo osservare la specifica sezione dedicata alla **RedSpaceShip**:



## ULTIMATE



L'**Ultimate** è l'astronave più forte del gioco per le sue sorprendenti caratteristiche. Non è affatto semplice acquistarla perché il suo costo è molto elevato (18000 **Coin**). Quando si acquista è automaticamente equipaggiata con 3 armi; le più forti del gioco. Questa astronave detiene il primato di sparare su 3 traiettorie distinte, la velocità dello sparo è altissima e inoltre causa un danno enorme. Le abilità dell'Ultimate consentiranno al giocatore di dominare letteralmente la galassia.

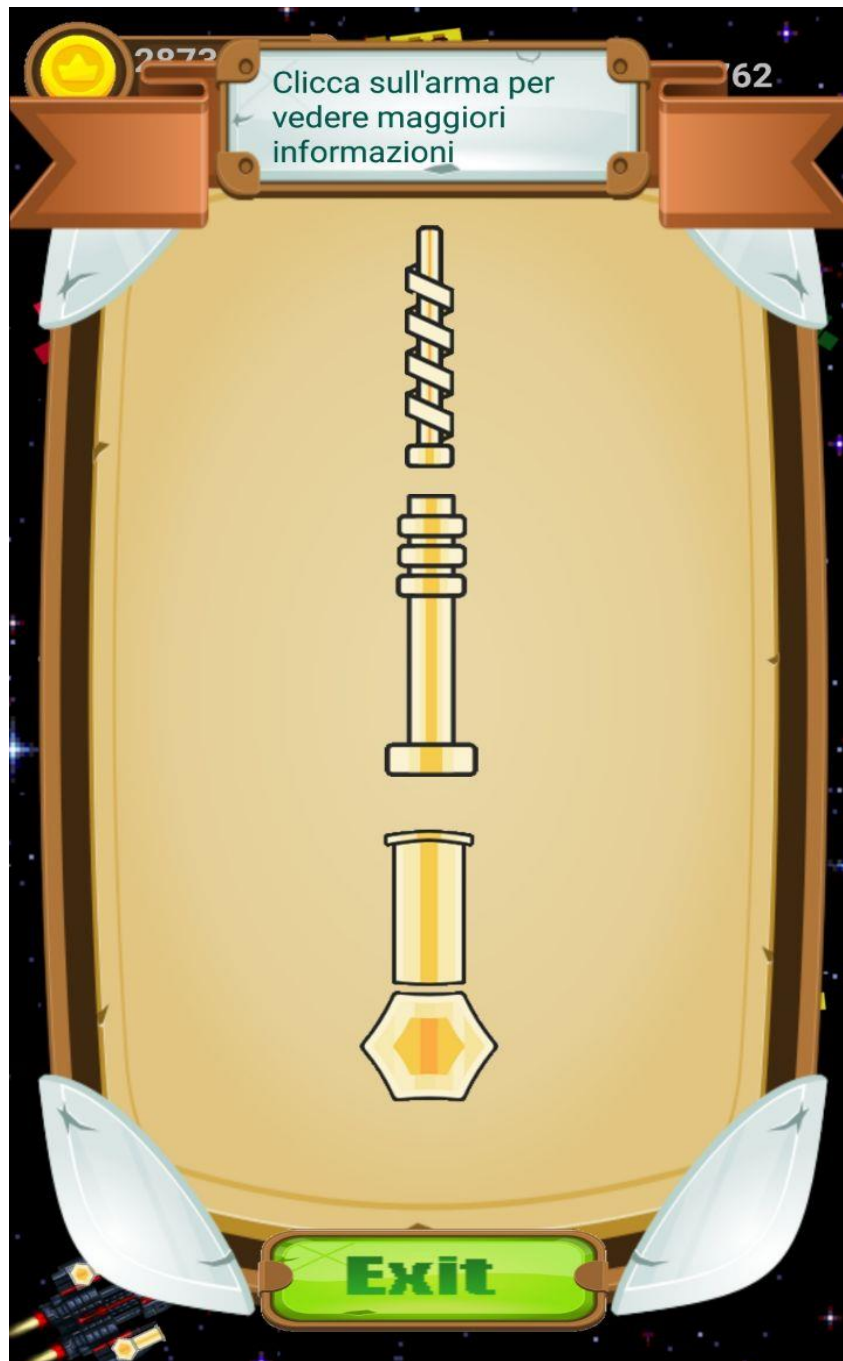
La sua vita è molto alta 16000 **hp**.

Possiamo osservare la specifica sezione dedicata alla **Ultimate**:

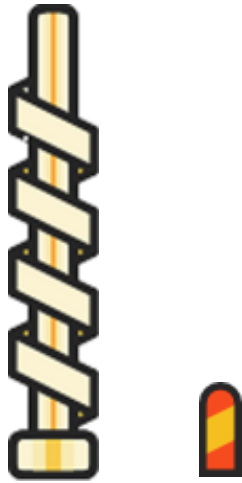


## SEZIONE ARMI

La sezione armi ci presenta 3 tipi di armi che possiamo equipaggiare alle nostre astronavi ad eccezione dell'**Ultimate** che è già integrata con l'arma più forte. Quindi l'equipaggiamento di queste singole armi sarà valido solo per la **GreenSpaceShip** e la **RedSpaceShip**.



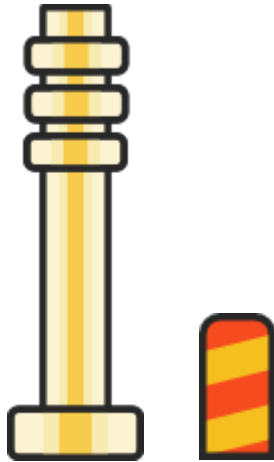
## LIGTH BLUSTER



Questa arma è l'arma base da cui si inizia. Spara proiettili di piccolo spessore e la potenza di fuoco è molto bassa. Questo influenza molto la velocità con cui i proiettili sono sparati. Quando si è ad un elevato **Score** i proiettili non fanno in tempo a colpire i nemici perché sono troppo veloci.



## MEDIUM BLUSTER

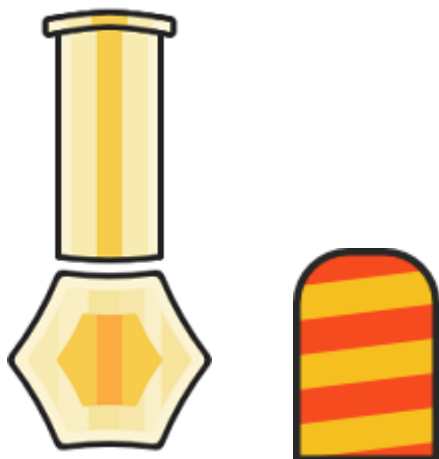


Questa arma è un'arma piuttosto avanzata rispetto a quella precedente. La potenza di fuoco è maggiore e di conseguenza i proiettili verranno sparati con una maggiore potenza e quindi c'è la possibilità di fare più danno.





## HEAVY BLUSTER



Questa è l'arma con potenza di fuoco maggiore ed è anche la più forte di tutto il gioco.

Spara i proiettili ad una velocità notevolmente più alta per la sua straordinaria potenza di fuoco. I proiettili hanno una dimensione gigantesca e causano un danno enorme. Se vuoi ottenere Score impensabili è sicuramente l'arma giusta!





## SCROOL



Sono i nemici che attaccano le astronavi, hanno una vita propria e inizialmente è di 50 **hp**.

Ogni 30 s la vita si incrementa di 25 **hp**

Fanno più danno alle astronavi.

E sono più veloci ad attaccare.

# PARTE DI CODICE E DI FUNZIONAMENTO

Per comprendere al meglio il funzionamento del progetto è necessario introdurre le classi che lo compongono spiegandone il funzionamento. Successivamente verranno spiegate le Activity che costituiscono l'intero progetto in cui verranno istanziati oggetti su cui verranno invocati i metodi delle varie classi.

---

## DatabaseHelper.java

È necessario introdurre il funzionamento di questa classe per la funzione fondamentale che svolge all'interno del nostro progetto. Questa classe ci ha permesso di creare un database interno all'applicazione che contiene una tabella che salva i dati necessari. Senza l'uso di questo database i dati si perderebbero ogni volta che viene chiusa l'applicazione. Ora descriveremo nel dettaglio il codice che contiene questa classe e la tabella che viene creata quando si istanzia un nuovo oggetto DatabaseHelper.

La classe DatabaseHelper estende la classe SQLiteOpenHelper (una classe helper per la gestione di un database; delle varie operazioni e anche delle varie versioni) le variabili dichiarate si riferiscono al nome del Database e ai vari nomi dei campi. Tutti questi dati vengo salvati all'interno della tabella **storage**.

**Id:** numero id del record

**Result:** il numero dei Coin disponibili

**Score:** lo Score massimo ottenuto

Il campo **Green** può contenere 2 interi: 1 nel caso in cui l'astronave non sia stata selezionata per la battaglia, 2 nel caso sia stata selezionata

Il campo **Red** può contenere 3 interi: 0 nel caso in cui l'astronave non sia stata comprata, 1 comprata ma non selezionata, 2 selezionata

Lo stesso vale per l'**Ultimate** e i campi successivi

Il campo **g1** "light bluster" equipaggiata all'astronave green (assume 0, 1,2)

Il campo **g2** "medium bluster" equipaggiata all'astronave green (assume 0, 1,2)

Il campo **g3** "heavy bluster" equipaggiata all'astronave green (assume 0, 1,2)

Il campo **r1, r2, r3** armi equipaggiate all'astronave red (assume 0, 1,2)

Il campo **u2** si riferisce all'arma equipaggiata alla Ultimate che può essere solo 1: la heavy bluster.

CAMPI TABELLA STORAGE DESCRITTI IN PRECEDENZA:

```

public class DatabaseHelper extends SQLiteOpenHelper {
    public static final String DATABASE_NAME="storage.db";
    public static final String COL_1="ID";
    public static final String COL_2="RESULT";
    public static final String COL_3="SCORE";
    public static final String COL_4="GREEN";
    public static final String COL_5="RED";
    public static final String COL_6="ULTIMATE";
    public static final String COL_7="G1";
    public static final String COL_8="G2";
    public static final String COL_9="G3";
    public static final String COL_10="R1";
    public static final String COL_11="R2";
    public static final String COL_12="R3";
    public static final String COL_13="U2";
}

```

COSTRUTTORE:

```

public DatabaseHelper( Context context) {
    super(context, DATABASE_NAME, factory: null, version: 1);
}

@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE storage(ID INTEGER PRIMARY KEY, RESULT LONG DEFAULT 0, " +
        "SCORE INT DEFAULT 0, GREEN INT DEFAULT 1, RED INT, ULTIMATE INT, G1 INT DEFAULT 1, G2 INT, G3 INT, " +
        "R1 INT DEFAULT 1, R2 INT, R3 INT , U2 INT)");
}

@Override
public void onUpgrade(SQLiteDatabase db, int i, int i1) {
    db.execSQL("DROP TABLE IF EXISTS storage");
    onCreate(db);
}

```

Il **costruttore** di questa classe riceve un riferimento al *Context* nonché il nome del database da cercare e la sua versione.

Il metodo **onCreate()** viene invocato solo alla prima richiesta del database nella vita dell'app ossia quando il database ancora non esiste.

Il metodo **onUpgrade()** viene invocato quando si istanzia un oggetto che crea un database uguale a quello già creato. Elimina il database che si sta per creare se esiste già.

```

public boolean insertData(int id2, long result2, int score2, int green, int red, int ultimate,
                          int g1, int g2, int g3, int r1, int r2, int r3, int u2)
{
    SQLiteDatabase db= this.getWritableDatabase();
    ContentValues contentValues=new ContentValues();
    contentValues.put(COL_1,id2);
    contentValues.put(COL_2,result2);
    contentValues.put(COL_3,score2);
    contentValues.put(COL_4,green);
    contentValues.put(COL_5,red);
    contentValues.put(COL_6,ultimate);
    contentValues.put(COL_7,g1);
    contentValues.put(COL_8,g2);
    contentValues.put(COL_9,g3);
    contentValues.put(COL_10,r1);
    contentValues.put(COL_11,r2);
    contentValues.put(COL_12,r3);
    contentValues.put(COL_13,u2);
    long results = db.insert( table: "storage", nullColumnHack: null, contentValues);
    if(results == -1)
    {
        return false;
    }
    else
    {
        return true;
    }
}

```

Il metodo **insertData()** ha come valore di ritorno un booleano che ritorna true se i valori sono stati inseriti correttamente nella tabella. I parametri vengono inseriti in un record della tabella.

```

public Cursor selectData()
{
    SQLiteDatabase db = this.getWritableDatabase();
    Cursor res = db.rawQuery( sql: "SELECT * FROM storage", selectionArgs: null);
    return res;
}

```

Il metodo **selectData()** restituisce il record della tabella con tutti i campi.

Il metodo **getWritableDatabase()** che viene invocato sul riferimento dell'oggetto restituisce un oggetto SQLiteDatabase che fornisce metodi per eseguire query SQL e SQLite.

Sull'oggetto "db" viene invocato il metodo **rawQuery()** che prende come parametro la query "SELECT \* from storage" e la esegue mettendo il risultato dell'occorrenza all'interno dell'oggetto Cursor. Il metodo restituisce l'oggetto Cursor che contiene il record del risultato della query.

```

public void updateData(long result2) {

    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues contentValues = new ContentValues();
    contentValues.put(COL_2,result2);
    db.update( table: "storage", contentValues, whereClause: null, whereArgs: null);
}

```

Successivamente verranno definiti i vari metodi update per avere la possibilità di modificare i vari campi del record della tabella. Ognuno di questi metodi ha la sintassi uguale con la sola differenza che la modifica avviene su un campo differente. Prendendo come esempio il metodo **updateData()**, inizialmente viene definito un oggetto SQLiteDatabase sul quale verrà eseguito il metodo update. Viene definito anche un oggetto ContentValues un insieme vuoto di valori con una dimensione iniziale predefinita. Su questo oggetto viene invocato il metodo put che aggiunge all'insieme vuoto il valore dei Coin aggiornati, in questo caso ci si riferisce alla variabile "result2". L'insieme che ora contiene il valore del campo aggiornato viene passato come parametro al metodo **update()** che esegue la modifica sul campo corrispondente del record della tabella "storage".

Questo principio di modifica vale per tutti i metodi **update()** dei vari campi. Ogni campo del record ha il proprio metodo update.

E' POSSIBILE TROVARE TUTTI I METODI NEL PERCORSO Videate/DataBaseHelper.

---

## Storage.java

Questa classe è stata ideata per salvare temporaneamente i progressi del gioco affinché vengano salvati successivamente nel database.

Per recuperare il valore di una variabile in un'altra classe è possibile scrivere storage.variabile (Es: storage.green)

```

public class Storage {

    long coinStorageI;
    static long coinStorageF;
    int scoreT;
    int green;
    int red;
    int ultimate;
    int g1,g2,g3;
    int r1,r2,r3;
}

```

## Costants.java

La classe Constants invece serve a definire la lunghezza e la larghezza del display di un qualsiasi dispositivo, in modo da renderne l'applicazione compatibile.


---

```
public class Constants {  
    //Constants  
    static int SCREEN_WIDTH;  
    static int SCREEN_HEIGHT;  
}
```

---

## DrawableConstants.java

La classe **DrawableConstants.java** è una classe d'appoggio creata per salvare gli eventi che l'utente fa all'interno dello shop, come ad esempio la selezione delle astronavi.

```
public class DrawableConstants {  
  
     String[] ship = new String[]{"g1", "g2", "g3", "r1", "r2", "r3", "ultimate2"};  
    static int pos;  
  
    public DrawableConstants()  
    {  
  
    }  
  
    public void setPos(int i) { pos = i; }  
  
    public int getPos() { return pos; }  
  
}
```

---



---

## Player.java

La classe **Player** estende `AppCompatActivity`, nel costruttore vengono passati:  
Il *Context* è il riferimento all'oggetto;

La *x* e la *y* sono le coordinate su cui verrà posizionato l'oggetto **Player** nel *Layout*.  
All'interno del costruttore viene definita la variabile "string" con il nome dell'immagine dell'astronave, presente nella cartella *Drawable* di *Android Studio*.  
L'astronave viene scelta dall'utente durante la selezione o l'acquisto all'interno dello shop.

Nella riga successiva la variabile "string" viene convertita in un oggetto *drawable* a cui viene assegnata una risorsa (`context.getPackageName()`). Questo oggetto *drawable* viene decodificato in un *id* che definirà la variabile "id".

Successivamente viene istanziato un oggetto *Drawable* che prende il parametro dell'id definito precedentemente. Poi vengono invocati dei metodi sull'oggetto stesso. L'immagine viene settata con l'oggetto *drawable* e viene definita la posizione sulla quale l'immagine dovrà essere posizionata all'interno del *layout*. Inoltre viene definito anche l'oggetto *Rect* che sarà di fondamentale importanza per controllare la collisione tra l'astronave e i nemici. L'oggetto *Rect* è un rettangolo che ha le stesse dimensioni dell'astronave che si sta controllando.

Il metodo **getRect()** ritorna l'oggetto *Rect*.

Il metodo **setRect()** permette di modificare le dimensioni dell'oggetto *Rect*.

```
public class Player extends AppCompatActivity {

    private Constants constants = new Constants();
    private DrawableConstants drawableConstants = new DrawableConstants();
    private Rect playerRect;

    public Player(Context context, float x, float y) {
        super(context);
        String string = drawableConstants.ship[drawableConstants.getPos()];
        int id = getResources().getIdentifier(string, "drawable", context.getPackageName());
        Drawable drawable = getResources().getDrawable(id, null);
        this.setImageDrawable(drawable);
        this.setX(x);
        this.setY(y);
        this.setMaxHeight(77);
        this.setMaxWidth(100);
        playerRect = new Rect((int) this.getX(), (int) this.getY(), (int) this.getX() + 250, (int) this.getY() + 185);
    }

    public Rect getRect() { return playerRect; }

    public Rect setRect(int left, int top, int right, int bottom) {
        playerRect.set(left + 80, top + 80, right + 300, bottom + 184);
        return playerRect;
    }

    public boolean touch() {
        this.setOnTouchListener((v, event) -> { return true; });
        return true;
    }
}
```

---

---

## Enemy.java

La classe Enemy estende la classe AppCompatActivity, oltre alla dichiarazione delle altre variabili viene definito un array di ImageView "images" che contiene al suo interno le 4 immagini degli Enemy.

Il costruttore prende come parametro:

Un *Context*: il riferimento all'oggetto Enemy

L'x e y le coordinate su cui verrà posizionato l'oggetto Enemy nel layout.

Inizialmente viene definita la variabile "path" con un indice casuale dell'array di Enemy definito fuori dal costruttore. Poi l'immagine del nemico generato casualmente viene messa all'interno della variabile "drawable". Viene creato un layout attorno all'immagine in cui viene definita la sua altezza e larghezza.

L'immagine viene settata con l'oggetto drawable, e viene definita la sua posizione nel layout. Successivamente viene definito l'oggetto "boundsEnemy" con le dimensioni dell'oggetto Enemy (questo oggetto servirà per la collisione tra nemico e astronave).

Viene definito un Handler che esegue il metodo **updateY()** ogni 10 millisecondi.

Il metodo **updateY** aumenta la Y dell'immagine e vengono settate anche le dimensioni del rettangolo che circonda l'immagine dell'Enemy. Perché il nemico si muove sul piano e le sue coordinate cambiano. Inizialmente la Y del Enemy viene incrementata di 3. Questo metodo viene eseguito ogni 10 millisecondi all'interno del metodo run del handler.

Il metodo **getSpeed()** ritorna lo speed corrente mentre **setSpeed()** ci permette di modificarlo( per aumentare la velocità con cui il nemico scende).

Il metodo **stopHandler()** stoppa l'handler (questo handler è utile quando si mette in pausa la partita e quando si perde)

### COSTRUTTORE:

```
public Enemy(Context context, float x, float y)
{
    super(context);
    path = random.nextInt(images.length);
    Drawable drawable = getResources().getDrawable(images[path]);
    this.setLayoutParams(new ConstraintLayout.LayoutParams(width: 150, height: 150));
    this.setImageDrawable(drawable);
    this.setX(x);
    this.setY(y);
    boundsEnemy = new Rect((int) this.getX(), (int) this.getY(), right: (int) this.getX()+150, bottom: (int) this.getY()+150);
    handler.post(runnable = (Runnable) () -> {
        updateY();
        handler.postDelayed(r: this, delayMillis: 10);
    });
}
```

## METODI CLASSE ENEMY

```
public void updateY()
{
    this.setY(this.getY() + speedY);
    boundsEnemy.set((int) this.getX(), top: (int) this.getY() + speedY, right: (int) this.getX() + 150,
        bottom: (int) (this.getY() + 150) + speedY);
}

public int getSpeedY()
{
    return speedY;
}

public void setSpeedY(int speedY)
{
    this.speedY = speedY;
}
```

```

public boolean collide(Player player)
{
    if(boundsEnemy.intersect(player.getRect()))
    {
        return true;
    }
    return false;
}

public boolean collideShield(ShieldPlayer shieldPlayer)
{
    if(boundsEnemy.intersect(shieldPlayer.getRect()))
    {
        return true;
    }
    return false;
}

public Rect getBoundsEnemy()
{
    return boundsEnemy;
}

```

Il metodo **collide()** ritorna "true" se il rettangolo delle dimensioni del nemico collide con il rettangolo dell'astronave.

Il metodo **collideShield()** ritorna "true" se il rettangolo delle dimensioni del nemico collide con il rettangolo dello shield (classe ShieldPlayer verrà spiegata successivamente).

Il metodo **getBoundsEnemy()** ritorna l'oggetto boundsEnemy.

Molte classi che verranno spiegate d'ora in poi presentano le stesse caratteristiche della classe Enemy riguardo al costruttore e ai metodi di esemplare.

---

## EnemyLife.java

Questa classe viene utilizzata per la creazione della ProgressBar per gli oggetti Enemy durante lo spawn.

All'interno del costruttore vengono invocati 2 metodi fondamentali per il funzionamento della ProgressBar.

**setMax()** setta il valore massimo che ha la barra quando viene creato l'oggetto.

**setProgress()** inizialmente viene settato con il valore massimo della vita, ma cambia a seconda del danno subito dall'Enemy. Si può definire la vita effettiva dell'Enemy. Mentre il valore della vita massima **setMax()** non cambia mai.

```
public class EnemyLife extends ProgressBar {

    private Handler handler = new Handler();
    private Runnable runnable;

    private int speedY = 3;

}
    public EnemyLife(final Context context, AttributeSet attrs, int defStyleAttr, float x, float y, int maxLife) {
        super(context, attrs, defStyleAttr);
        this.setLayoutParams(new ConstraintLayout.LayoutParams( width: 140, height: 15));
        this.setX(x);
        this.setY(y);
        this.setMax(maxLife);
        this.setProgress(maxLife);
    }
    handler.post(runnable = (Runnable) () -> {
        updateY();
        handler.postDelayed( () this, delayMillis: 10);
    });

}

    public void updateY() { this.setY(this.getY() + speedY); }

    public void setSpeedY(int speedY) { this.speedY = speedY; }

    public void stopHandler() { handler.removeCallbacks(runnable); }

}
```

---

## ShieldPlayer.java

L'oggetto che viene istanziato da questa classe è lo shield che circonda l'astronave quando viene presa la pallina dello shield.

La classe ShieldPlayer è estesa da AppCompatActivity, il suo costruttore ha gli stessi parametri delle classi Player ed Enemy. I metodi e il funzionamento di questa classe sono analoghi a quelli della classe Enemy. Viene definito l'oggetto drawable con l'immagine dello shield e successivamente vengono definite: la sua posizione, le dimensioni, il suo rettangolo (shieldRect).

Il metodo **getRect()** restituisce l'oggetto shieldRect e **setRect()** permette di modificarlo.

```

public class ShieldPlayer extends AppCompatActivity {

    private Rect shieldRect;

}

public ShieldPlayer(Context context, float x, float y) {
    super(context);
    Drawable drawable = getResources().getDrawable(R.drawable.shield);
    this.setLayoutParams(new ConstraintLayout.LayoutParams( width: 550, height: 550));
    this.setImageDrawable(drawable);
    this.setX(x);
    this.setY(y);
    shieldRect = new Rect((int)this.getX(), (int)this.getY(), right: (int)this.getX()+500, bottom: (int)this.getY()+500);
}

}

public Rect getRect() { return shieldRect; }

public Rect setRect(int left, int top, int right, int bottom)
{
    shieldRect.set( left: left + 80 , top: top + 80, right: right + 500, bottom: bottom + 500);
    return shieldRect;
}
}

```



## Shield.java

L'oggetto che viene istanziato da questa classe è lo shield che spawna insieme agli Enemy.

La classe Shield estende la classe AppCompatActivity, il suo costruttore contiene al suo interno gli stessi metodi della classe Enemy. L'immagine dello shield viene posizionata sul layout secondo i parametri x, y del costruttore, viene definito il suo rettangolo per le collisioni con l'astronave, inoltre viene definito anche un Handler che esegue il metodo **updateY()** ogni 10 millisecondi.



```

public class Shield extends AppCompatActivity {

    private Handler handler = new Handler();
    private Runnable runnable;
    private Rect shieldRect;
    private int speedY = 3;

    public Shield(Context context, float x, float y) {
        super(context);
        Drawable drawable = getResources().getDrawable(R.drawable.blu);
        this.setLayoutParams(new ConstraintLayout.LayoutParams(width: 75, height: 75));
        this.setImageDrawable(drawable);
        this.setX(x);
        this.setY(y);
        shieldRect = new Rect((int) this.getX(), (int) this.getY(), right: (int) this.getX() + 75, bottom: (int) this.getY() + 75);
        handler.post(runnable = (Runnable) () -> {
            updateY();
            handler.postDelayed(this, delayMillis: 10);
        });
    }

    public void updateY() {
        this.setY(this.getY() + speedY);
        shieldRect.set((int) this.getX(), top: (int) this.getY() + speedY, right: (int) this.getX() + 75,
            bottom: (int) (this.getY() + 75) + speedY);
    }

    public void setSpeedY(int speedY) {
        this.speedY = speedY;
    }

    public boolean collide(Player player)
    {
        if(shieldRect.intersect(player.getRect()))
        {
            return true;
        }
        return false;
    }
}

```

---

## Bullet.java

L'oggetto che viene istanziato da questa classe e' il proiettile che viene sparato dall'astronave nella battaglia. L'impostazione è sempre la stessa della classe Enemy. La sola differenza è che nel metodo **updateY()** la Y dell'oggetto Bullet viene decrementata perché i proiettili dovranno attraversare il piano dal basso verso l'alto in direzione opposta a quella degli Enemy.

```

public Bullet(Context context, float x, float y) {
    super(context);
    this.setLayoutParams(new ConstraintLayout.LayoutParams( width: 35, height: 70));
    this.setX(x);
    this.setY(y);
    this.setMaxWidth(5);
    this.setMaxHeight(11);
    boundsBullet = new Rect((int)this.getX(), (int)this.getY(), right: (int)this.getX()+35, bottom: (int)this.getY()+70);
    handler.post(runnable = (Runnable) () -> {
        updateY();
        handler.postDelayed( this, millis);
    });
}

```

```

public void updateY() {
    this.setY(this.getY() - speedY);
    boundsBullet.set((int)this.getX(), top: (int)this.getY() - 15, right: (int)this.getX()+35,
        bottom: (int)(this.getY()+70)-15);
}

```

```

public void setMillis(long time) {
    millis = time;
}

```

```

public void setSpeedY(int speedY) {
    this.speedY = speedY;
}

```

---

## ExplosionPlayer.java

Questa esplosione segue gli stessi movimenti dell'astronave inizialmente è invisibile ma quando avviene la collisione diventa visibile.

L'oggetto istanziato da questa classe è usato per l'animazione dell'esplosione quando avviene una collisione. Nel costruttore viene settato il background dell'immagine con il file xml **runExplosion.xml** che contiene al suo interno un'animazione costituita da 16 ImageView. Successivamente vengono definite la

posizione, le dimensioni e infine l'oggetto runExplosion con il background dell'ImageView.

Il metodo **startAnimation()** esegue l'animazione del runExplosion.xml. Ogni ImageView viene mostrata per 30 millisecondi. Questo metodo risulta utile quando l'astronave viene distrutta ed esplode.

```
public class ExplosionPlayer extends AppCompatActivity {

    private Handler handler = new Handler();
    private Runnable runnable;
    final AnimationDrawable runExplosion;
    private Costants costants = new Costants();

    public ExplosionPlayer(Context context, float x, float y) {
        super(context);
        this.setBackgroundResource(R.drawable.run_explosion);
        this.setX(x);
        this.setY(y);
        this.setMaxHeight(77);
        this.setMaxWidth(100);
        this.setVisibility(INVISIBLE);
        runExplosion = (AnimationDrawable) this.getBackground();
    }

    public void startAnimation() { runExplosion.start(); }
}
```

FILE XML RunExplosion

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <animation-list
3      android:oneshot="true"
4      xmlns:android="http://schemas.android.com/apk/res/android">
5
6      <item
7          android:drawable="@drawable/explosion1"
8          android:duration="30" />
9
10     <item
11         android:drawable="@drawable/explosion2"
12         android:duration="30" />
13
14     <item
15         android:drawable="@drawable/explosion3"
16         android:duration="30" />
17
18     <item
19         android:drawable="@drawable/explosion4"
20         android:duration="30" />
21
22     <item
23         android:drawable="@drawable/explosion5"
24         android:duration="30" />
25
26     <item
27         android:drawable="@drawable/explosion6"
28         android:duration="30" />
29
30     <item
31         android:drawable="@drawable/explosion7"
32         android:duration="30" />
33 </animation-list>
```

---

---

## ExplosionEnemy.java

Questa classe è riconducibile alla classe Enemy perché l'esplosione segue i movimenti dell'Enemy con l'unica differenza che all'inizio l'oggetto è invisibile, ma quando un proiettile o il player collide con il nemico, l'oggetto diventa visibile e viene fatta partire l'animazione dell'esplosione sempre attraverso il file RunExplosion.xml.

### CLASSE E METODI

```
public class ExplosionEnemy extends AppCompatActivity {

    private Handler handler = new Handler();
    private Runnable runnable;
    final AnimationDrawable runExplosion;
    private int speedY = 3;

    public ExplosionEnemy(Context context, float x, float y) {
        super(context);
        this.setLayoutParams(new ConstraintLayout.LayoutParams( width: 150, height: 150));
        this.setBackgroundResource(R.drawable.run_explosion);
        this.setX(x);
        this.setY(y);
        this.setVisibility(INVISIBLE);
        runExplosion = (AnimationDrawable) this.getBackground();
        handler.post(runnable = (Runnable) () -> {
            updateY();
            handler.postDelayed( r: this, delayMillis: 10);
        });
    }

    public void updateY() { this.setY(this.getY() + speedY); }

    public void setSpeedY(int speedY) { this.speedY = speedY; }

    public void startAnimation() { runExplosion.start(); }
}
```

---

---

## MainActivity.java

La **MainActivity.java** è usata per il funzionamento e la struttura del menù principale. È una delle classi più importanti.

In questa Activity avviene la gestione:

del tipo di evento sui bottoni START, SHOP, EXIT che sono i fondamenti da cui il gioco svolge le operazioni più importanti;

dalla visibilità dei vari ConstraintLayout in base al bottone che viene premuto;

dell'intero shop con le varie sezioni al suo interno;

e il salvataggio dei dati in un Database interno all'applicazione grazie all'uso della classe DatabaseHelper.java.

Ora vedremo nello specifico le varie funzioni di questa Activity:

### *Costruttore e inizializzazione gioco*

Nel costruttore della classe MainActivity viene definito un oggetto SQLiteOpenHelper (db) che fa riferimento al database contenente la tabella "storage". Quando l'utente fa l'accesso al gioco per la prima volta il metodo **SelectData()** viene invocato sull'oggetto "db". Questo metodo ritorna un oggetto di tipo Cursor: Quest'oggetto fornisce l'accesso in modalità di lettura-scrittura al **result set** restituito dalla query. Attraverso ad un ciclo sull'oggetto Cursor sarà possibile avere accesso ai dati ottenuti. Sull'oggetto Cursor "res" viene invocato il metodo **getCount()** che restituisce il numero di record che sono all'interno della tabella. Con il primo accesso al gioco la tabella non avrà record quindi di conseguenza, il **getCount()** restituirà 0. Se restituisce 0 verrà inserito un record all'interno del database che configurerà il gioco per avere la selezione dell'astronave greenSpaceShip, lo Score e i Coin con valore 0. Se invece il valore restituito è diverso da 0 viene fatto un ciclo while sull'oggetto "res" che ci darà la possibilità di scorrerlo e di accedere a tutti i valori dei campi del record della tabella "storage". Ogni corrispettivo valore viene messo all'interno delle variabili d'appoggio della classe Storage. Il nome di queste variabili è uguale al nome del campo a cui si riferiscono. Queste variabili storage ci consentiranno di usare i valori ricavati dalla tabella "storage", di compiere le operazioni necessarie per il funzionamento del gioco e in caso di modifica di aggiornare il database.



```

Cursor res = db.selectData();
if(res.getCount() == 0) {
    db.insertData( id2: 1, result2: 0, score2: 0, green: 1, red: 0, ultimate: 0, g1: 1,
        g2: 0, g3: 0, r1: 0, r2: 0, r3: 0, u2: 0);
}

if (res.getCount() != 0) {
    while (res.moveToNext()) {
        textcoin.setText((res.getString( columnIndex: 1)));
        textScore.setText((res.getString( columnIndex: 2)));
        storage.green = res.getInt( columnIndex: 3);
        storage.red = res.getInt( columnIndex: 4);
        storage.ultimate = res.getInt( columnIndex: 5);
        storage.g1 = res.getInt( columnIndex: 6);
        storage.g2 = res.getInt( columnIndex: 7);
        storage.g3 = res.getInt( columnIndex: 8);
        storage.r1 = res.getInt( columnIndex: 9);
        storage.r2 = res.getInt( columnIndex: 10);
        storage.r3 = res.getInt( columnIndex: 11);

        System.out.println("green " + storage.green);
        System.out.println("red " + storage.red);
        System.out.println("ultimate " + storage.ultimate);
        System.out.println("g1 " + storage.g1);
        System.out.println("g2 " + storage.g2);
        System.out.println("g3 " + storage.g3);
        System.out.println("r1 " + storage.r1);
        System.out.println("r2 " + storage.r2);
        System.out.println("r3 " + storage.r3);
    }
}

```

## START (AVVIO DELLA BATTAGLIA)

Quando avviene un evento Click sull'immagine Start viene avviato un Handler ovvero un thread che esegue porzioni di codice ogni tot tempo stabilito dal programmatore.

Inizialmente all'interno di questo Handler viene mostrato il livello della barra di progressione. Successivamente viene eseguito un controllo sulla progressione della barra.

Finchè il caricamento è minore di 100 la barra continuerà ad incrementarsi e quando arriverà a 100 il gioco verrà eseguito attraverso l'esecuzione del metodo **StartGame()** inoltre la barra di progressione verrà rimessa a 0.

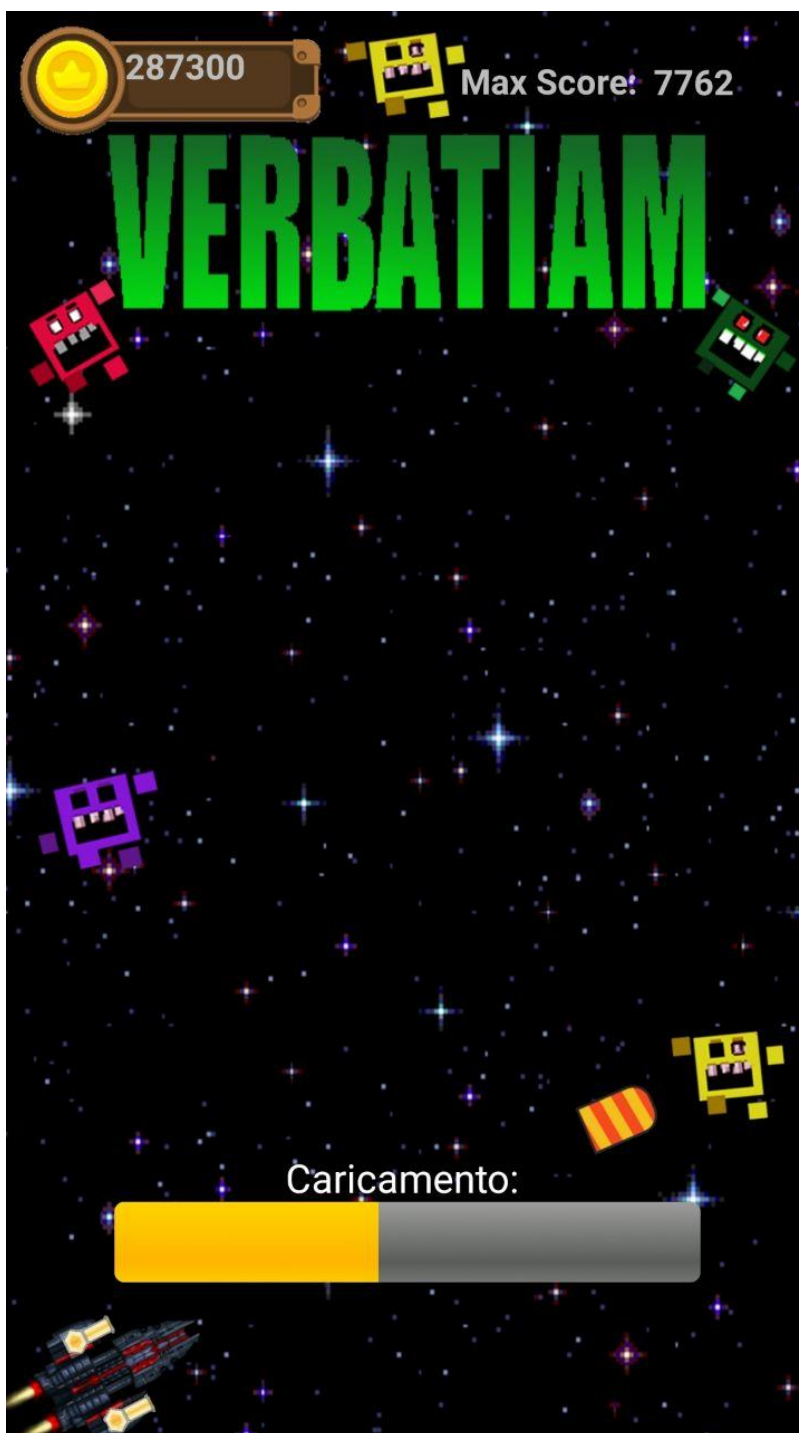
HandlerCount.postDelayed(this,500) permette di ripetere le righe di codice scritta prima.

```

//Fa partire il gioco (menu)
startI.setOnClickListener((v) -> {
    handlerCount.post(runnableCount = (Runnable) () -> {
        if(storage.green == 2 && (storage.g1 == 2 || storage.g2 == 2 && storage.g3 == 2))
        {
            if (handlerFlag) {
                start();
                handlerCount.postDelayed( r: this, delayMillis: 500);
            }
        }
        else if(storage.red == 2 && (storage.r1 == 2 || storage.r2 == 2 || storage.r3 == 2))
        {
            if (handlerFlag) {
                start();
                handlerCount.postDelayed( r: this, delayMillis: 500);
            }
        }
        else if(storage.ultimate == 2)
        {
            if (handlerFlag) {
                start();
                handlerCount.postDelayed( r: this, delayMillis: 500);
            }
        }
        else
        {
            Toast.makeText( context: MainActivity.this, text: "Assicurati di aver selezionato l'arma " +
                "corrispondente all'astronave", Toast.LENGTH_SHORT).show();
        }
    });
});

```

## CARICAMENTO PROGRESS BAR



*Metodi usati nell'evento click dell'immagine Start*

```

//Metodo usato per caricare la seconda activity
public void startGame() {
    Cursor res = db.selectData();
    while (res.moveToNext()) {
        storage.coinStorageF = res.getLong( columnIndex: 1);
    }
    Intent intent = new Intent( packageContext: MainActivity.this, GamePanel.class);
    startActivity(intent);
}

//Metodo usato per incrementare la progress bar
public int count() {
    if (i < 50) {
        i = i + 5;
    } else if (i >= 50) {
        i = i + 2;
    }
    return i;
}

```

## SHOP (AVVIO DELLO SHOP)

```

//Apri la sezione shop (menu)
shopI.setOnClickListener((v) → {
    startI.setVisibility(View.INVISIBLE);
    exitI.setVisibility(View.INVISIBLE);
    shopI.setVisibility(View.INVISIBLE);
    shopL.setVisibility(View.VISIBLE);
});

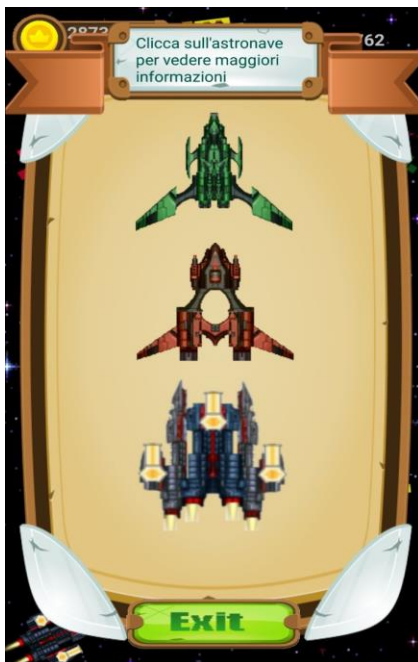
//Esce dalla sezione shop (layout shop)
exitS.setOnClickListener((v) → {
    shopL.setVisibility(View.INVISIBLE);
    startI.setVisibility(View.VISIBLE);
    exitI.setVisibility(View.VISIBLE);
    shopI.setVisibility(View.VISIBLE);
});

```

Cliccando sull'immagine SHOP vengono resi invisibili i pulsanti START, SHOP, EXIT e reso visibile il ConstraintLayout dello shop con le varie sezioni. Cliccando EXIT viene fatta l'operazione contraria.

Cliccando sull'immagine della sezione astronave a sua volta viene mostrato un secondo layout con la possibilità di scegliere tra 3 tipi di astronavi. Il Click sull'immagine exit permetterà l'uscita dalla sezione.

```
//Premendo sull'immagine selezione astonavi,  
//si apre un secondo layout (Ship Layout)  
shipSection.setOnClickListener((v) → {  
    shopL.setVisibility(View.INVISIBLE);  
    shipL.setVisibility(View.VISIBLE);  
});  
  
//Esce dal ship layout con il pulsante exit  
//rendendo visibile lo shop layout e  
//invisibile lo ship layout  
exitShipLayout.setOnClickListener((v) → {  
    shopL.setVisibility(View.VISIBLE);  
    shipL.setVisibility(View.INVISIBLE);  
});
```



A destra: Sezione Shop  
A sinistra: Sezione Astronavi



### Descrizione (Sezione Astronavi)

Ogni astronave ha la sua sezione specifica con la sua descrizione. Qui è presente il codice per l'evento Click sull'immagine di ogni astronave. Quando avviene un Click sull'immagine dell'astronave GreenSpaceShip, si apre un layout con la descrizione dell'astronave e immagini (con funzione da pulsante), **Torna Indietro** e **Seleziona**. **N.B** La GreenSpaceShip è già selezionata di default.

Per quanto riguarda le altre astronavi oltre alla gestione della visibilità del nuovo layout vengono fatti dei controlli sull'acquisto e la selezione della singola astronave.

```
/* Premendo sull'astronave GreenSpaceShip
vengono rese invisibile le altre astronavi presenti nel layout
a parte quella selezionata ovvero l'astronave GreenSpaceShip
*/
green.setOnClickListener((v) -> {
    green.setVisibility(View.VISIBLE);
    red.setVisibility(View.INVISIBLE);
    ultimate.setVisibility(View.INVISIBLE);
    backButton.setVisibility(View.VISIBLE);
    selectionGreen.setVisibility(View.VISIBLE);
    descrizioneG.setVisibility(View.VISIBLE);
    descrizioneR.setVisibility(View.INVISIBLE);
    descrizioneU.setVisibility(View.INVISIBLE);
});

/* Premendo sull'astronave RedSpaceShip
vengono rese invisibile le altre astronavi presenti nel layout
a parte quella selezionata ovvero l'astronave RedSpaceShip
*/
red.setOnClickListener((v) -> {
    green.setVisibility(View.INVISIBLE);
    red.setVisibility(View.VISIBLE);
    ultimate.setVisibility(View.INVISIBLE);
    backButton.setVisibility(View.VISIBLE);
    descrizioneG.setVisibility(View.INVISIBLE);
    descrizioneR.setVisibility(View.VISIBLE);
    descrizioneU.setVisibility(View.INVISIBLE);

    if (storage.red >= 1) {
        buyButtonRed.setVisibility(View.INVISIBLE);
        selectionRed.setVisibility(View.VISIBLE);
    } else {
        buyButtonRed.setVisibility(View.VISIBLE);
        selectionRed.setVisibility(View.INVISIBLE);
    }

    //Va alla posizione della astronave verde
    red.setX(336);
    red.setY(300);
});
```

---

```

/* Premendo sull'astronave Ultimate
vengono rese invisibile le altre astronavi presenti nel layout
a parte quella selezionata ovvero l'astronave Ultimate
*/
ultimate.setOnClickListener((v) → {
    green.setVisibility(View.INVISIBLE);
    red.setVisibility(View.INVISIBLE);
    ultimate.setVisibility(View.VISIBLE);
    backButton.setVisibility(View.VISIBLE);
    descrizioneG.setVisibility(View.INVISIBLE);
    descrizioneR.setVisibility(View.INVISIBLE);
    descrizioneU.setVisibility(View.VISIBLE);

    if (storage.ultimate >= 1) {
        buyButtonUltimate.setVisibility(View.INVISIBLE);
        selectionUltimate.setVisibility(View.VISIBLE);
    } else {
        buyButtonUltimate.setVisibility(View.VISIBLE);
        selectionUltimate.setVisibility(View.INVISIBLE);
    }

    //Va alla posizione della astronave verde
    ultimate.setX(336);
    ultimate.setY(285);
});

```

### ***Pulsanti Seleziona e Compra (Astronavi e Armi)***

Ora vedremo il codice dei seguenti eventi sull'immagine **Seleziona** e **Compra** di ciascuna astronave.

La greenSpaceShip è l'astronave che il gioco offre quando si inizia a giocare, di conseguenza la sua sezione ci offrirà soltanto la possibilità di selezionarla. Facendo un Click sul pulsante "Seleziona" verrà rilevato un evento `setOnClickListener` sulla corrispondente immagine che invocherà all'interno del metodo stesso **weapon()** che prende come parametro la stringa "green". (Il metodo **weapon()** verrà spiegato successivamente)



```

477         //Seleziona l'astronave GreenSpaceShip
478         //Invocazione del metodo selezionaGreen()
479         selectionGreen.setOnClickListener(new View.OnClickListener() {
480             @Override
481             public void onClick(View v) {
482                 weapon("green");
483             }
484         });

```

I metodi `setOnClickListener` per la selezione delle altre astronavi sono analoghi. Il metodo **`weapon()`** gestisce con un `switch` tutti i casi per la selezione delle astronavi con l'equipaggiamento della corrispettiva arma. Invoca a seconda dello specifico caso il metodo specifico per la selezione dell'astronave o dell'arma che vogliamo.

```

public void weapon(String s) {
    switch (s) {
        case "g1": g1();
        |         break;
        case "g2": g2();
        |         break;
        case "g3": g3();
        |         break;
        case "r1": r1();
        |         break;
        case "r2": r2();
        |         break;
        case "r3": r3();
        |         break;
        case "ultimate": ultimate();
        |         break;
        case "green": green();
        |         break;
        case "red": red();
    }
}

```

Ad esempio nel caso precedente avevamo selezionato l'astronave "green", il metodo **`weapon()`** riceve il parametro "green" ed invoca il metodo **`green()`**.



Nel metodo **green()** viene effettuato un controllo: Se la variabile d'appoggio della classe storage che fa riferimento al valore del campo "green" nel record della tabella storage è uguale a 1 o maggiore (ciò significa che l'astronave è acquistata ) viene effettuato un controllo sulle possibili armi selezionate con l'astronave redSpaceship e se l'astronave redSpaceShip ed Ultimate sono selezionate. Se uno di questi controlli è vero la variabile che fa riferimento al campo specifico viene rimessa a 1 (acquistato ma non selezionato) e viene aggiornato il campo del database.

```
1328     public void green() {
1329         if(storage.green >= 1) {
1330             if (storage.r1 == 2) { storage.r1 = 1;db.updateR1(storage.r1); }
1331             if (storage.r2 == 2) { storage.r2 = 1;db.updateR2(storage.r2); }
1332             if (storage.r3 == 2) { storage.r3 = 1;db.updateR3(storage.r3); }
1333             if (storage.ultimate == 2) { storage.ultimate = 1;db.updateUltimate(storage.ultimate); }
1334             if (storage.red == 2) { storage.red = 1;db.updateRed(storage.red); }
1335
1336             storage.green = 2;
1337             db.updateGreen(storage.green);
1338             ship404.setImageResource(R.drawable.greenship);
1339             gunSection.setVisibility(View.VISIBLE);
1340         }
1341     }
```

Ora che abbiamo scelto l'astronave vorremo selezionare anche una arma da equipaggiare.

A seconda dell'astronave scelta vengo mostrati sul Layout le immagini che fanno riferimento ad essa. Ad esempio se selezioniamo la greenSpaceShip le immagini delle armi associate ad essa prenderanno il nome di "g1" "g2" "g3" mentre per la redSpaceShip "r1" "r2" "r3". Con la selezione dell'Ultimate non c'è il bisogno di selezionare un'arma perchè viene acquistata direttamente con la heavy Bluster (l'arma 3).

```
//Premendo sull'immagine selezione astonavi,  
//si apre un secondo layout (gun Layout)  
gunSection.setOnClickListener((v) -> {  
    shopL.setVisibility(View.INVISIBLE);  
    gunL.setVisibility(View.VISIBLE);  
    if (storage.green == 2) {  
        r1.setVisibility(View.INVISIBLE);  
        r2.setVisibility(View.INVISIBLE);  
        r3.setVisibility(View.INVISIBLE);  
        g1.setVisibility(View.VISIBLE);  
        g2.setVisibility(View.VISIBLE);  
        g3.setVisibility(View.VISIBLE);  
    } else if (storage.red == 2) {  
        r1.setVisibility(View.VISIBLE);  
        r2.setVisibility(View.VISIBLE);  
        r3.setVisibility(View.VISIBLE);  
        g1.setVisibility(View.INVISIBLE);  
        g2.setVisibility(View.INVISIBLE);  
        g3.setVisibility(View.INVISIBLE);  
    }  
});
```

Qui possiamo vedere un esempio di quando clicchiamo sull'immagine dell'arma specifica per l'astronave greenSpaceShip (in questo caso g1):

```
//Premendo sull'arma, verranno rese invisibili le altre due armi  
g1.setOnClickListener((v) → {  
    g1.setVisibility(View.VISIBLE);  
    g2.setVisibility(View.INVISIBLE);  
    g3.setVisibility(View.INVISIBLE);  
    backButtonG.setVisibility(View.VISIBLE);  
    descrizioneG1.setVisibility(View.VISIBLE);  
    descrizioneG2.setVisibility(View.INVISIBLE);  
    descrizioneG3.setVisibility(View.INVISIBLE);  
    descrizioneR1.setVisibility(View.INVISIBLE);  
    descrizioneR2.setVisibility(View.INVISIBLE);  
    descrizioneR3.setVisibility(View.INVISIBLE);  
  
    if (storage.g1 >= 1) {  
        selectionG1.setVisibility(View.VISIBLE);  
    } else {  
        selectionG1.setVisibility(View.INVISIBLE);  
    }  
});
```

Se all'astronave "green" selezionata in precedenza volessimo equipaggiare l'arma "g1" quando faremo un Click sull'immagine "Seleziona" nella Sezione dell'arma Light Bluster il metodo **weapon()** invocherebbe il metodo g1() .

```
//Serve per selezionare l'arma G1
selectionG1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        weapon( s: "g1");
    }
});
```

Il principio del metodo g1() è simile a quello della selezione dell'astronave. Viene effettuato un controllo: Se la variabile d'appoggio della classe storage che fa riferimento al valore del campo "g1" nel record della tabella storage è uguale a 1 o maggiore (ciò significa che g1 è acquistata viene effettuato un controllo sulle altre armi selezionate con la stessa astronave e anche su quelle selezionate dalla redSpaceship. Vengono controllate anche se le astronavi redSpaceShip ed Ultimate sono selezionate. Se uno di questi controlli è vero la variabile che fa riferimento al campo specifico viene rimessa a 1 (acquistato ma non selezionato) e viene aggiornato il campo del database.

```
public void g1() {
    if (storage.g1 >= 1) {
        if (storage.g2 == 2) { storage.g2 = 1;db.updateG2(storage.g2); }
        if (storage.g3 == 2) { storage.g3 = 1;db.updateG3(storage.g3); }
        if (storage.r1 == 2) { storage.r1 = 1;db.updateR1(storage.r1); }
        if (storage.r2 == 2) { storage.r2 = 1;db.updateR2(storage.r2); }
        if (storage.r3 == 2) { storage.r3 = 1;db.updateR3(storage.r3); }
        if (storage.red == 2) { storage.red = 1;db.updateRed(storage.red); }
        if (storage.ultimate == 2) { storage.ultimate = 1;db.updateUltimate(storage.ultimate); }
        storage.g1 = 2;
        db.updateG1(storage.g1);
        drawableCostants.setPos(0);
        weapon404.setImageResource(R.drawable.barrels);
    }
}
```

Questo è un esempio di Selezione di un'astronave con la corrispettiva arma. Il procedimento è analogo per la selezione delle altre astronavi e armi.

Ora vedremo i metodi che ci permettono di acquistare le astronavi. In particolare la redSpaceShip e L'ultimate.

Per acquistare un'astronave viene estrapolato il dato dei Coin effettivi dal database. Questi Coin sono quelli che possediamo al momento dell'acquisto. Successivamente viene effettuato un controllo se i Coin sono sufficienti per acquistare l'astronave e anche se quell'astronave non è già stata acquistata. Se questo controllo è vero, ai nostri Coin viene sottratto il prezzo dell'astronave e il valore viene aggiornato nel Database. L'astronave redSpaceShip viene acquistata direttamente con i due cannoni della light Bluster mentre l'astronave Ultimate con i cannoni della heavy Bluster ed esclude la possibilità di equipaggiare le armi più deboli.

Esempio:

```
//Metodo usato per la gestione dell'acquisto dell'astronave RedSpaceShip
public void buyRed() {
    long redCoin = 5000L;
    long currentCoin = 0L;

    Cursor res = db.selectData();

    while (res.moveToNext()) {
        currentCoin = res.getLong( columnIndex: 1);
    }

    if (currentCoin >= redCoin && storage.red == 0) {
        currentCoin = currentCoin - redCoin;
        storage.coinStorageF = storage.coinStorageF - redCoin;

        db.updateData(currentCoin);
        textcoin.setText(currentCoin + "");

        storage.red = 2;
        db.updateRed(storage.red);
        if (storage.r2 == 0 && storage.r3 == 0) {
            storage.r1 = 2;
            db.updateR1(storage.r1);
        }
    } else {
        System.out.println("Mi dispiace ma non puoi acquistare questa nave, perchè non hai " +
            "abbastanza soldi. \n L'astronave costa 5000 coin mentre tu ne possiedi" +
            "|" + currentCoin + "\nOppure e' perche' hai gia' acquistato questa astronave");
    }
}
```

Per spiegare l'acquisto di un'arma faremo un esempio. Il procedimento che verrà spiegato sarà valido per l'acquisto di ogni' arma.

Prendiamo come esempio l'acquisto dell'arma "g2".

Quando viene rilevato un evento Click sull'immagine "Compra" all'interno del metodo `setOnClickListener()` viene invocato il metodo `"buyG2()"`.

*//Serve per comprare l'arma G2*

```
buyButtonG2.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        buyG2();  
    }  
});
```

Il procedimento applicato nel metodo `"buyG2()"` è analogo a quello del `"buyRed()"`.

*//Metodo usato per la gestione dell'acquisto dell'arma G2*

```
public void buyG2() {  
    long g2 = 2000L;  
    long currentCoin = 0L;  
  
    Cursor res = db.selectData();  
  
    while (res.moveToNext()) {  
        currentCoin = res.getLong(columnIndex: 1);  
    }  
  
    if (currentCoin >= g2 && storage.g2 == 0) {  
        currentCoin = currentCoin - g2;  
        storage.coinStorageF = storage.coinStorageF - g2;  
  
        db.updateData(currentCoin);  
        textcoin.setText(currentCoin + "");  
  
        storage.g2 = 1;  
        db.updateG2(storage.g2);  
        System.out.println("Acquistato");  
    } else {  
        System.out.println("Mi dispiace ma non puoi acquistare questa arma perchè " +  
            "non hai abbastanza soldi. \n L'arma costa 5000 coin mentre tu ne possiedi" +  
            " " + currentCoin + "\nOppure e' perche' hai gia' acquistato questa arma");  
    }  
}
```

## N.B

E'POSSIBILE TROVARE NEL PERCORSO Videate/MainActivity/ TUTTI I METODI RELATIVI ALL'ACQUISTO E ALLA SELEZIONE DI ARMI E ASTRONAVI.

PER CHIARIRE EVENTUALI DUBBI SUL LORO FUNZIONAMENTO ANALOGO.

## **BackButton e ExitG (Per uscire dalle sezioni astronavi e armi)**

Su ogni sezione dell'astronave specifica, a sinistra del layout troviamo il pulsante "torna indietro" all'evento click su questa immagine viene reso invisibile tutto quello che riguarda il layout corrente (l'astronave specifica sopra la descrizione, la sua descrizione, i pulsanti seleziona o compra) e si tornerà al layout precedente con le varie sezioni delle astronavi.

```
//Serve per uscire dalla sezione Ship layout
backButton.setOnClickListener((v) → {
    green.setVisibility(View.VISIBLE);
    red.setVisibility(View.VISIBLE);
    ultimate.setVisibility(View.VISIBLE);
    backButton.setVisibility(View.INVISIBLE);
    selectionGreen.setVisibility(View.INVISIBLE);
    buyButtonUltimate.setVisibility(View.INVISIBLE);
    selectionUltimate.setVisibility(View.INVISIBLE);
    buyButtonRed.setVisibility(View.INVISIBLE);
    selectionRed.setVisibility(View.INVISIBLE);
    descrizioneG.setVisibility(View.INVISIBLE);
    descrizioneR.setVisibility(View.INVISIBLE);
    descrizioneU.setVisibility(View.INVISIBLE);

    //Ritorna alla posizione originale
    green.setX(336);
    green.setY(306);
    //Ritorna alla posizione originale
    red.setX(339);
    red.setY(685);
    //Ritorna alla posizione originale
    ultimate.setX(339);
    ultimate.setY(1083);
});
```



Lo stesso funzionamento è applicato anche all'immagine "torna indietro" posta nel Layout della sezione specifica dell'arma. All'evento Click su questa immagine vengono mostrate le armi che fanno riferimento all'astronave che abbiamo selezionato per la battaglia.

```
//Esce dal ship layout con il pulsante exit
//rendendo visibile lo shop layout e
//invisibile gun layout
exitG.setOnClickListener((v) -> {
    shopL.setVisibility(View.VISIBLE);
    gunL.setVisibility(View.INVISIBLE);
    if (storage.green == 2) {
        r1.setVisibility(View.INVISIBLE);
        r2.setVisibility(View.INVISIBLE);
        r3.setVisibility(View.INVISIBLE);
        g1.setVisibility(View.VISIBLE);
        g2.setVisibility(View.VISIBLE);
        g3.setVisibility(View.VISIBLE);
        backButtonG.setVisibility(View.INVISIBLE);
        buyButtonG3.setVisibility(View.INVISIBLE);
        buyButtonG2.setVisibility(View.INVISIBLE);

        //Ritorna alla posizione originale
        g1.setX(336);
        g1.setY(306);
        //Ritorna alla posizione originale
        g2.setX(339);
        g2.setY(685);
        //Ritorna alla posizione originale
        g3.setX(336);
        g3.setY(1155);
    } else if (storage.red == 2) {
        r1.setVisibility(View.VISIBLE);
        r2.setVisibility(View.VISIBLE);
        r3.setVisibility(View.VISIBLE);
        g1.setVisibility(View.INVISIBLE);
        g2.setVisibility(View.INVISIBLE);
        g3.setVisibility(View.INVISIBLE);
        backButtonG.setVisibility(View.INVISIBLE);
        buyButtonR3.setVisibility(View.INVISIBLE);
        buyButtonR2.setVisibility(View.INVISIBLE);

        //Ritorna alla posizione originale
        r1.setX(336);
        r1.setY(306);
        //Ritorna alla posizione originale
        r2.setX(339);
        r2.setY(685);
        //Ritorna alla posizione originale
        r3.setX(336);
        r3.setY(1155);
    }
});
```

## GamePanel.java

Questa classe è stata creata per gestire l'intera partita del gioco. Si occupa della gestione dei movimenti dell'astronave, la creazione dei nemici, dello shield e dei proiettili.

---

```
db = new DatabaseHelper( context: this);

Cursor res = db.selectData();
if(res.getCount() == 0) {
    db.insertData( id2: 1, storage.coinStorageF, storage.scoreT, green: 1, red: 0, ultimate: 0, g1: 1, g2: 0,
                  g3: 0, r1: 1, r2: 0, r3: 0, u2: 0);
}

if(res.getCount() != 0) {
    while (res.moveToNext()) {
        //storage.coinStorageF = res.getLong(1);
        storage.green = res.getInt( columnIndex: 3);
        storage.red = res.getInt( columnIndex: 4);
        storage.ultimate = res.getInt( columnIndex: 5);
        storage.g1 = res.getInt( columnIndex: 6);
        storage.g2= res.getInt( columnIndex: 7);
        storage.g3= res.getInt( columnIndex: 8);
        storage.r1= res.getInt( columnIndex: 9);
        storage.r2 = res.getInt( columnIndex: 10);
        storage.r3= res.getInt( columnIndex: 11);
    }
}
```

Quando l'utente fa l'accesso al gioco per la prima volta il metodo **SelectData()** viene invocato sull'oggetto "db". Questo metodo ritorna un oggetto di tipo Cursor: Quest'oggetto fornisce l'accesso in modalità di lettura-scrittura al **result set** restituito dalla query. Attraverso ad un ciclo sull'oggetto Cursor sarà possibile avere accesso ai dati ottenuti. Sull'oggetto Cursor "res" viene invocato il metodo **getCount()** che restituisce il numero di record che sono all'interno della tabella. Con il primo accesso al gioco la tabella non avrà record quindi di conseguenza, il **getCount()** restituirà 0. Se restituisce 0 verrà inserito un record all'interno del database che configurerà il gioco per avere la selezione dell'astronave greenSpaceShip, lo Score e i Coin con valore 0. Se invece il valore restituito è diverso da 0 viene fatto un ciclo while sull'oggetto "res" che ci darà la possibilità di scorrerlo e di accedere a tutti i valori dei campi del record della tabella "storage". Ogni corrispettivo valore viene messo all'interno delle variabili d'appoggio della classe Storage. Il nome di queste variabili è uguale al nome del campo a cui si riferiscono. Queste variabili storage ci consentiranno di usare i valori ricavati dalla tabella "storage", di compiere le operazioni necessarie per il funzionamento del gioco e in caso di modifica di aggiornare il database.

---

---

```
explosionSound = MediaPlayer.create( context: this, R.raw.explosion);  
lasershootSound = MediaPlayer.create( context: this, R.raw.lasershoot);  
gameSong = MediaPlayer.create( context: this, R.raw.game);  
gameOver = MediaPlayer.create( context: this, R.raw.gameover);  
lasershootSound.setVolume( leftVolume: 0.0f, rightVolume: 0.1f);  
explosionSound.setVolume( leftVolume: 0.0f, rightVolume: 0.3f);  
gameSong.setVolume( leftVolume: 0.0f, rightVolume: 0.5f);  
gameOver.setVolume( leftVolume: 0.0f, rightVolume: 1f);
```

Qua vengono inizializzate le variabili di tipo MediaPlayer che serviranno per aggiungere musiche o clip audio per l'effetto sonoro del gioco.

---

```
//Set Player
```

```
player = new Player( context: this, x: 400, y: 1500);  
gioco.addView(player);
```

```
//Set Shield
```

```
shieldPlayer = new ShieldPlayer( context: this, x: 325, y: 1410);  
gioco.addView(shieldPlayer);  
shieldPlayer.setVisibility(View.INVISIBLE);
```

```
//Set ExplosionPlayer
```

```
explosionPlayer = new ExplosionPlayer( context: this, x: 800, y: 850);  
gioco.addView(explosionPlayer);
```

Inizializzazione degli oggetti Player, ShieldPlayer ed ExplosionPlayer. Inoltre durante l'inizializzazione viene specificato dove questi oggetti devono essere posizionati all'interno del layout e successivamente vengono aggiunti nel ConstraintLayout di nome **gioco**.

---

---

```
handlerScore.post(runnableScore = new Runnable() {  
    @Override  
    public void run() {  
        //DoSomething()  
        handlerScore.postDelayed( r: this, delayMillis: 30);  
    }  
});
```

All'interno del costruttore GamePanel.java si possono trovare dei metodi Handler che servono per la gestione dei processi concorrenti.

Per permettere l'esecuzione di un metodo più volte, ad esempio per effettuare vari controlli su oggetti come la collisione tra l'oggetto Bullet e l'oggetto Enemy, viene usato il metodo Handler.postDelay.

Questo metodo ci permette di eseguire più volte un metodo dopo una certa frequenza di millisecondi impostata dal programmatore.

---

```
public void backgroundScrool()  
{  
    ImageView backgroundI = (ImageView) findViewById(R.id.img);  
    if(backgroundI.getY() < -6300)  
    {  
        backgroundI.setY(0);  
    }  
    else  
    {  
        backgroundI.setY(backgroundI.getY() - 10);  
    }  
}
```

Viene inizializzato una variabile backgroundI di tipo ImageView.

Successivamente viene effettuato un controllo, se la posizione della coordinata Y dell'immagine è minore di -6300 allora viene impostata la coordinata Y a 0.

Mentre se non fosse così, la coordinata Y dell'immagine viene diminuita di 10 ogni 50 millisecondi. Questo avviene grazie all'handlerBackground (Lo si può trovare nel percorso Videate/GamePanel/Handler/HandlerBackGround.PNG).

---

---

## hitCheckBulletEnemy()

```
ArrayList<Enemy> deleteEnemy = new ArrayList<>();  
ArrayList<Bullet> deleteBullet = new ArrayList<>();  
ArrayList<EnemyLife> deleteEnemyLife = new ArrayList<>();
```

All'inizio del metodo vengono inizializzati 3 ArrayList d'appoggio, ognuno contenente un oggetto differente.

```
for (Bullet bullet: bullets)  
{  
    for (final Enemy enemy: enemies)  
    {  
        for (EnemyLife enemyLife: enemyLives)  
        {  
            if (bullet.collide(enemy))  
            {  
                deleteBullet.add(bullet);  
            }  
        }  
    }  
}
```

Nel primo ciclo foreach viene controllato in continuazione se un determinato oggetto bullet ed enemy collidono. Se ciò avviene, viene aggiunto nell'ArrayList deleteBullet quel determinato oggetto Bullet che ha provocato la collisione con l'oggetto Enemy.

```
if(enemyLife.getX() == enemy.getX() || enemyLife.getY() == enemy.getY())
{
    if(storage.g1 == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletG1);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
    if(storage.g2 == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletG2);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
    if(storage.g3 == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletG3);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
    if(storage.r1 == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletR1);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
    if(storage.r2 == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletR2);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
    if(storage.r3 == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletR3);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
    if(storage.ultimate == 2)
    {
        enemyLife.setProgress(enemyLife.getProgress() - bulletU1);
        deleteEnemyLife.add(enemyLife);
        deleteEnemy.add(enemy);
    }
}
```

In questo pezzo di codice vengono mostrati una serie di controlli necessari per la diminuzione della ProgressBar appartenente ad ogni nemico. La diminuzione della vita del nemico viene gestita in base all'astronave che si sta usando.

Inoltre quando l'oggetto Enemy viene colpito, l'oggetto ProgressBar a lui associato, viene aggiunto all'ArrayList deleteEnemyLife.

```
if(enemyLife.getProgress() <= 0)
{
    coin = coin + 2;
    for(final ExplosionEnemy explosionEnemy : explosionEnemies)
    {
        explosionEnemy.post(new Runnable() {
            @Override
            public void run() {
                if(explosionEnemy.getX() == enemy.getX()
                || explosionEnemy.getY() == enemy.getY())
                {
                    explosionSound.start();
                    explosionEnemy.setVisibility(View.VISIBLE);
                    explosionEnemy.startAnimation();
                }
            }
        });
    }
}
```

Quando la vita di uno specifico nemico raggiunge un valore negativo o zero viene fatta partire l'animazione dell'esplosione.

Inoltre durante questa operazione viene fatta partire anche il clip audio dedicato all'esplosione.



```

for(Bullet bullet: deleteBullet)
{
    for(Enemy enemy: deleteEnemy)
    {
        bullets.remove(bullet);
        gioco.removeView(bullet);
        System.out.println(deleteBullet.size());
        for(EnemyLife enemyLife: deleteEnemyLife)
        {
            if(enemyLife.getProgress() <= 0)
            {
                enemy.stopHandler();
                bullet.stopHandler();
                enemyLife.stopHandler();
                enemies.remove(enemy);
                enemyLifes.remove(enemyLife);
                gioco.removeView(enemyLife);
                gioco.removeView(enemy);
            }
        }
    }
}

```

Nella seconda parte di codice di questo metodo troviamo 3 foreach che servono per l'eliminazione degli oggetti che fanno parte dei vari arrayList "delete" ed oltre a ciò vengono eliminati anche dalla View.

---

---

## hitCheckShieldPlayer()

```
public void hitCheckShieldPlayer() {  
    //Spawn oggetto esplosione  
    ArrayList<Shield> deleteShield = new ArrayList<>();  
  
    for (Shield shield : shields) {  
        if (shield.collide(player))  
        {  
            shield.setVisibility(View.VISIBLE);  
            shieldPlayer.setVisibility(View.VISIBLE);  
            deleteShield.add(shield);  
        }  
    }  
  
    for (Shield shield : deleteShield) {  
        shields.remove(shield);  
        gioco.removeView(shield);  
    }  
}
```

Questo metodo controlla la collisione tra la pallina con il simbolo dello scudo e con l'astronave.

Viene creato un arrayList di nome "deleteShield".

Poi successivamente vengono usati una serie di foreach per ciclare l'arrayList shields e al suo interno controllare se colpisce l'astronave. Se questo avviene l'oggetto Shield che è stato coinvolto nella collisione viene aggiunto all'arrayList "deleteShield". Successivamente con il secondo foreach viene rimosso dalla View la pallina che si è scontrata con il Player.

---

---

## hitCheckShieldEnemy()

```
public void hitCheckShieldEnemy() {
```

```
    //Spawn oggetto esplosione
```

```
    ArrayList<Enemy> deleteEnemy = new ArrayList<>();
```

```
    ArrayList<EnemyLife> deleteEnemyLife = new ArrayList<>();
```

```
    for(final Enemy enemy: enemies)
```

```
    {
```

```
        if (enemy.collideShield(shieldPlayer))
```

```
        {
```

```
            for(EnemyLife enemyLife: enemyLives)
```

```
            {
```

```
                if (enemyLife.getX() == enemy.getX() || enemyLife.getY() == enemy.getY()) {
```

```
                    deleteEnemyLife.add(enemyLife);
```

```
                    shieldPlayer.setVisibility(View.INVISIBLE);
```

```
                    deleteEnemy.add(enemy);
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

```
    for (Enemy enemy : deleteEnemy)
```

```
    {
```

```
        for(EnemyLife enemyLife: deleteEnemyLife)
```

```
        {
```

```
            enemies.remove(enemy);
```

```
            gioco.removeView(enemy);
```

```
            enemyLives.remove(enemyLife);
```

```
            gioco.removeView(enemyLife);
```

```
        }
```

```
    }
```

```
}
```

Questo metodo controlla la collisione tra lo scudo e il nemico. Vengono creati 2 arrayList di nome "deleteEnemy" e "deleteEnemyLife". Poi successivamente vengono usati una serie di foreach per ciclare l'arrayList "enemies" e "enemyLives" per controllare se un nemico colpisce lo scudo che circonda l'astronave. Se questo avviene l'oggetto Enemy che è stato coinvolto nella collisione viene aggiunto all'arrayList "deleteEnemy" e la stessa cosa avviene anche per l'oggetto EnemyLife. Successivamente con il secondo foreach viene rimosso dalla View l'oggetto Enemy che ha colpito lo scudo.

---

---

## hitCheckEnemyPlayer()

```
ArrayList<Enemy> deleteCollision = new ArrayList<>();  
ArrayList<EnemyLife> deleteEnemyLife = new ArrayList<>();
```

All'inizio del metodo vengono inizializzati 2 ArrayList d'appoggio, ognuno contenente un oggetto differente.

```
for(final Enemy enemy: enemies)  
{  
    if(enemy.collide(player))  
    {  
        for(EnemyLife enemyLife: enemyLives)  
        {  
            if (enemyLife.getX() == enemy.getX() || enemyLife.getY() == enemy.getY()) {  
                deleteEnemyLife.add(enemyLife);  
            }  
        }  
        for(final ExplosionEnemy explosionEnemy : explosionEnemies)  
        {  
            explosionEnemy.post(() -> {  
                if(explosionEnemy.getX() == enemy.getX() || explosionEnemy.getY() == enemy.getY())  
                {  
                    explosionSound.start();  
                    explosionEnemy.setVisibility(View.VISIBLE);  
                    explosionEnemy.startAnimation();  
                }  
            });  
        }  
        deleteCollision.add(enemy);  
        life.setProgress(life.getProgress() - damageEnemy);  
    }  
}
```

Nella prima parte di codice possiamo trovare un foreach che cicla per l'oggetto di tipo Enemy e poi invece è presente un controllo che verifica se uno specifico oggetto Enemy collide con il player.

Se questo dovrebbe risultare vero allora viene fatto un altro foreach che cicla per l'oggetto EnemyLife e se quest'oggetto ha la stessa posizione del nemico allora viene aggiunto nell'arrayList di appoggio (deleteEnemyLife).

Poi successivamente viene fatto un ciclo per l'esplosione relativa ai nemici che collidono con l'astronave e anche la sottrazione della vita dell'astronave.

```

for (Enemy enemy : deleteCollision) {
    for (EnemyLife enemyLife: deleteEnemyLife) {
        enemy.stopHandler();
        enemies.remove(enemy);
        gioco.removeView(enemy);
        enemyLives.remove(enemyLife);
        gioco.removeView(enemyLife);
        if (life.getProgress() <= 0) {
            gameSong.stop();
            gameOver.start();
            gioco.removeView(player);
            explosionPlayer.setVisibility(View.VISIBLE);
            explosionPlayer.startAnimation();
            handlerScore.removeCallbacks(runnableScore);
            handler.removeCallbacks(runnable);
            handlerShoot.removeCallbacks(runnableShoot);
            handlerEnemy.removeCallbacks(runnableEnemy);
            handlerCollisionEnemy.removeCallbacks(runnableCollisionEnemy);
            handlerCollisionBullets.removeCallbacks(runnableCollisionBullets);
            gameOver.setVisibility(View.VISIBLE);
            storage.coinStorageI = storage.coinStorageI + coin;
            storage.coinStorageF = storage.coinStorageF + coin;
            storage.scoreT = contatoreTimer;
            Cursor res = db.selectData();
            if (res.getCount() != 0)
            {
                db.updateData( result2: storage.coinStorageF+10000);
                db.updateScore(storage.scoreT);
            }
            String risultato="";
            while (res.moveToNext()) {
                risultato = "Coin Totali " + res.getString( columnIndex: 1)
                    + "\nNe hai ottenuti " + storage.coinStorageI;
            }
            coinViewAll.setText(risultato);
        }
    }
}
}

```

Nella seconda parte di codice vengono ciclati i due ArrayList d'appoggio e per ogni oggetto vengono fatte determinate azione, come la rimozione dell'oggetto dalla View o dall'ArrayList.

Successivamente viene effettuato un controllo se la vita dell'astronave è minore o uguale a zero.

Se così fosse allora vengono messi in pausa tutti gli handler relativi allo spawn dei nemici, dei proiettili, dell'incremento dello score e molti altri.

Inoltre viene resa visibile la scritta "Game Over" e vengono registrati nella classe Storage i Coin ottenuti in battaglia. Infine vengono anche aggiunti i Coin ottenuti e lo score effettuato nel database e viene mostrato a video anche quanti Coin si hanno a disposizione e quanti se ne sono guadagnati in battaglia.

---

## SpawnEnemy() DeleteEnemy() DeleteEnemyLife()

```
public void spawnEnemy()  
{  
    float enemyX = (float) Math.floor(Math.random() * (costants.SCREEN_WIDTH - 150));  
    float enemyY = -100;  
  
    //Spawn oggetto enemy  
    Enemy enemy = new Enemy( context: this, enemyX, enemyY);  
    enemies.add(enemy);  
    gioco.addView(enemy, index: 1);  
  
    //Spawn oggetto progressBar  
    EnemyLife enemyLife = new EnemyLife( context: this, attrs: null, android.R.attr.progressBarStyleHorizontal,  
        enemyX, enemyY, maxLife);  
    enemyLives.add(enemyLife);  
    gioco.addView(enemyLife, index: 1);  
  
    //Spawn oggetto explosionEnemy  
    ExplosionEnemy explosionEnemy = new ExplosionEnemy( context: this, enemyX, enemyY);  
    explosionEnemies.add(explosionEnemy);  
    gioco.addView(explosionEnemy, index: 1);  
}
```

All'interno di questo metodo, grazie all'uso di un handler, vengono creati in continuazione oggetti Enemy, EnemyLife ed ExplosionEnemy, aggiunti negli arrayList opportuni dichiarati ed infine aggiunti al ConstraintLayout.

Lo stesso metodo viene utilizzato per l'oggetto Shield e lo si può trovare in Videate/GamePanel/Shield/SpawnShield.PNG

```

public void deleteEnemy()
{
    ArrayList<Enemy> delete = new ArrayList<>();

    for (Enemy enemy : enemies)
    {
        if (enemy.getY() > constants.SCREEN_HEIGHT)
        {
            delete.add(enemy);
        }
    }

    for (Enemy enemy : delete)
    {
        enemy.stopHandler();
        enemies.remove(enemy);
        gioco.removeView(enemy);
    }
}

```

Viene inizializzato l'arrayList "delete" di tipo Enemy.

Viene usato un foreach che cicla per l'array enemies di tipo Enemy e se un determinato oggetto esce dal layout, allora viene eliminato.

L'eliminazione e la rimozione dell'oggetto Enemy, avviene grazie al secondo foreach.

Lo stesso metodo viene utilizzato per l'oggetto EnemyLife e anche per l'oggetto Shield.

Il primo lo si può trovare in Videate/GamePanel/Enemy/DeleteEnemyLife.PNG mentre il secondo in Videate/GamePanel/Shield/DeleteShield.PNG

---

---

## SpawnBullet() DeleteBullet()

```
if(storage.g1 == 2 || storage.g2 == 2 || storage.g3 == 2)
{
    Bullet bulletG = new Bullet( context: this, x: player.getX()+177, y: player.getY());
    bullets.add(bulletG);
    gioco.addView(bulletG, index: 1);
    if(storage.g1 == 2)
    {
        Drawable drawable = getResources().getDrawable(R.drawable.bullets);
        bulletG.setImageDrawable(drawable);
        bulletG.setMillis(20);
    }
    if(storage.g2 == 2)
    {
        Drawable drawable = getResources().getDrawable(R.drawable.bulletm);
        bulletG.setImageDrawable(drawable);
        bulletG.setMillis(20);
    }
    if(storage.g3 == 2)
    {
        Drawable drawable = getResources().getDrawable(R.drawable.bulleth);
        bulletG.setImageDrawable(drawable);
        bulletG.setMillis(20);
        bulletG.setLayoutParams(new ConstraintLayout.LayoutParams( width: 45, height: 75));
    }
}

else if(storage.r1 == 2 || storage.r2 == 2 || storage.r3 == 2)
{
    Bullet bulletR1 = new Bullet( context: this, x: player.getX()+110, y: player.getY()+20);
    Bullet bulletR2 = new Bullet( context: this, x: player.getX()+250, y: player.getY()+20);
    bullets.add(bulletR1);
    bullets.add(bulletR2);
    gioco.addView(bulletR1, index: 1);
    gioco.addView(bulletR2, index: 1);
    if(storage.r1 == 2)
    {
        Drawable drawable = getResources().getDrawable(R.drawable.bullets);
        bulletR1.setImageDrawable(drawable);
        bulletR2.setImageDrawable(drawable);
        bulletR1.setMillis(15);
        bulletR2.setMillis(15);
    }
    if(storage.r2 == 2)
    {
        Drawable drawable = getResources().getDrawable(R.drawable.bulletm);
        bulletR1.setImageDrawable(drawable);
        bulletR2.setImageDrawable(drawable);
        bulletR1.setMillis(15);
        bulletR2.setMillis(15);
    }
    if(storage.r3 == 2)
    {
        Drawable drawable = getResources().getDrawable(R.drawable.bulleth);
        bulletR1.setImageDrawable(drawable);
        bulletR2.setImageDrawable(drawable);
        bulletR1.setMillis(15);
        bulletR2.setMillis(15);
        bulletR1.setLayoutParams(new ConstraintLayout.LayoutParams( width: 45, height: 75));
        bulletR2.setLayoutParams(new ConstraintLayout.LayoutParams( width: 45, height: 75));
    }
}
```



```

else if(storage.ultimate == 2)
{
    Bullet bulletU1 = new Bullet( context: this, x: player.getX()+10, y: player.getY()+50);
    Bullet bulletU2 = new Bullet( context: this, x: player.getX()+145, y: player.getY());
    Bullet bulletU3 = new Bullet( context: this, x: player.getX()+280, y: player.getY()+50);
    bullets.add(bulletU1);
    bullets.add(bulletU2);
    bullets.add(bulletU3);
    gioco.addView(bulletU1, index: 1);
    gioco.addView(bulletU2, index: 1);
    gioco.addView(bulletU3, index: 1);
    Drawable drawable = getResources().getDrawable(R.drawable.bulleth);
    bulletU1.setImageDrawable(drawable);
    bulletU2.setImageDrawable(drawable);
    bulletU3.setImageDrawable(drawable);
    bulletU1.setLayoutParams(new ConstraintLayout.LayoutParams( width: 45, height: 75));
    bulletU2.setLayoutParams(new ConstraintLayout.LayoutParams( width: 45, height: 75));
    bulletU3.setLayoutParams(new ConstraintLayout.LayoutParams( width: 45, height: 75));
    bulletU1.setMillis(9);
    bulletU2.setMillis(9);
    bulletU3.setMillis(9);
}

```

Per la creazione di oggetti di tipo Bullet, bisogna controllare che tipo di arma si sta usando.

Ogni arma spara un Bullet differente e a seconda dell'arma che si sta usando viene usata una determinata immagine.

Ed inoltre la velocità dello sparo e/o il danno cambia.

```

public void deleteBullet()
{
    ArrayList<Bullet> delete = new ArrayList<>();
    for(Bullet bullet: bullets)
    {
        if(bullet.getY() < - bullet.getHeight())
        {
            delete.add(bullet);
        }
    }
    for(Bullet bullet : delete)
    {
        bullet.stopHandler();
        bullets.remove(bullet);
        gioco.removeView(bullet);
    }
}

```

Viene inizializzato l'arrayList "delete" di tipo Bullet.

Viene usato un foreach che cicla per l'array bullets di tipo Bullet e se un determinato oggetto esce dal layout, allora viene eliminato.

L'eliminazione e la rimozione dell'oggetto Bullet, avviene grazie al secondo foreach.